PKCS #11 Base Functionality v2.30: Cryptoki – Draft 4

RSA Laboratories

10 July 2009

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the basic PKCS#11 token interface and token behavior.

2 Scope

This standard specifies an application programming interface (API), called "Cryptoki," to devices which hold cryptographic information and perform cryptographic functions. Cryptoki 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 typically be provided via C header files by the supplier of a Cryptoki library. Generic ANSI C header files for Cryptoki are available from the PKCS Web page. This document and up-to-date errata for Cryptoki will also be available from the same place.

Additional documents may provide a generic, language-independent Cryptoki interface and/or 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 possibly in a separate document.

Details of cryptographic mechanisms (algorithms) may be found in the associated document PKCS#11 Mechanisms.

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

3 References

ANSI C ANSI/ISO. *American National Standard for Programming Languages – C*. 1990.

CC/PP W3C. *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies*. World Wide Web Consortium, January 2004. URL: http://www.w3.org/TR/CCPP-struct-vocab/

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

FIPS PUB 46–3 NIST. *FIPS 46-3: Data Encryption Standard (DES).* October 25, 1999. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 74 NIST. *FIPS 74: Guidelines for Implementing and Using the NBS Data Encryption Standard.* April 1, 1981. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 81 NIST. *FIPS 81: DES Modes of Operation.* December 1980. URL: http://csrc.nist.gov/publications/fips/index.html

FIPS PUB 113 NIST. *FIPS 113: Computer Data Authentication.* May 30, 1985. URL: http://csrc.nist.gov/publications/fips/index.html

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

ISO/IEC 7816-1 ISO. *Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 1: Physical Characteristics.* 1998.

ISO/IEC 7816-4 ISO. *Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 4: Interindustry Commands for Interchange.* 1995.

ISO/IEC 8824-1 ISO. *Information Technology-- Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.* 2002.

ISO/IEC 8825-1 ISO. *Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).* 2002.

ISO/IEC 9594-1 ISO. *Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services.* 2001.

ISO/IEC 9594-8 ISO. *Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks.* 2001.

ISO/IEC 9796-2 ISO. *Information Technology — Security Techniques — Digital Signature Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms.* 2002.

Java MIDP Java Community Process. *Mobile Information Device Profile for Java 2 Micro Edition.* November 2002. URL: http://jcp.org/jsr/detail/118.jsp

MeT-PTD MeT. *MeT PTD Definition – Personal Trusted Device Definition*, Version 1.0, February 2003. URL: http://www.mobiletransaction.org

PCMCIA Personal Computer Memory Card International Association. *PC Card Standard*, Release 2.1,. July 1993.

PKCS #1 RSA Laboratories. *RSA Cryptography Standard.* v2.1, June 14, 2002.

PKCS #3 RSA Laboratories. *Diffie-Hellman Key-Agreement Standard.* v1.4, November 1993.

PKCS #5 RSA Laboratories. *Password-Based Encryption Standard*. v2.0, March 25, 1999

PKCS #7 RSA Laboratories. *Cryptographic Message Syntax Standard.* v1.5, November 1993

PKCS #8 RSA Laboratories. *Private-Key Information Syntax Standard*. v1.2, November 1993.

PKCS #11-C RSA Laboratories. *PKCS #11: Conformance Profile Specification*, October 2000.

PKCS #11-P RSA Laboratories. *PKCS #11 Profiles for mobile devices*, June 2003.

PKCS #11-M1 RSA Laboratories. *PKCS #11 Mechanisms*, June 2009.

PKCS #11-M2 RSA Laboratories. *PKCS #11 Other Mechanisms*, June 2009.

PKCS #12 RSA Laboratories. *Personal Information Exchange Syntax Standard*. v1.0, June 1999.

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. URL: http://ietf.org/rfc/rfc1421.txt

RFC 2045 Freed, N., and N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. November 1996. URL: http://ietf.org/rfc/rfc2045.txt

RFC 2246 T. Dierks & C. Allen. *RFC 2246: The TLS Protocol Version 1.0*. Certicom, January 1999. URL: http://ietf.org/rfc/rfc2246.txt

RFC 2279 F. Yergeau. *RFC 2279:* UTF-8, a transformation format of ISO 10646 Alis Technologies, January 1998. URL: http://ietf.org/rfc/rfc2279.txt

RFC 2534 Masinter, L., Wing, D., Mutz, A., and K. Holtman. *RFC 2534: Media Features for Display, Print, and Fax.* March 1999. URL: http://ietf.org/rfc/rfc2534.txt

RFC 2630 R. Housley. *RFC 2630: Cryptographic Message Syntax*. June 1999. URL: http://ietf.org/rfc/rfc2630.txt

RFC 2743 J. Linn. *RFC 2743: Generic Security Service Application Program Interface Version 2, Update 1.* RSA Laboratories, January 2000. URL: http://ietf.org/rfc/rfc2743.txt

RFC 2744	J. Wray. *RFC 2744: Generic Security Services API Version 2: C-bindings.* Iris Associates, January 2000. URL: http://ietf.org/rfc/rfc2744.txt
SEC 1	Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography*. Version 1.0, September 20, 2000.
SEC 2	Standards for Efficient Cryptography Group (SECG). Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters. Version 1.0, September 20, 2000.
TLS	IETF. *RFC 2246: The TLS Protocol Version 1.0* . January 1999. URL: http://ietf.org/rfc/rfc2246.txt
WIM	WAP. *Wireless Identity Module. — WAP-260-WIM-20010712-a.* July 2001. URL: http://www.wapforum.org/
WPKI	WAP. *Wireless PKI. — WAP-217-WPKI-20010424-a*. April 2001. URL: http://www.wapforum.org/
WTLS	WAP. *Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a.* April 2001. URL: http://www.wapforum.org/.
X.500	ITU-T. *Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services.* February 2001. Identical to ISO/IEC 9594-1
X.509	ITU-T. *Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks.* March 2000. Identical to ISO/IEC 9594-8
X.680	ITU-T. *Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.* July 2002. Identical to ISO/IEC 8824-1
X.690	ITU-T. *Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).* July 2002. Identical to ISO/IEC 8825-1

4 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.680.
Attribute	A characteristic of an object.
BER	Basic Encoding Rules, as defined in X.690.
CBC	Cipher-Block Chaining mode, as defined in FIPS PUB 81.
Certificate	A signed message binding a subject name and a public key, or a subject name and a set of attributes.
CMS	Cryptographic Message Syntax (see RFC 2630)
Cryptographic Device	A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.
Cryptoki	The Cryptographic Token Interface defined in this standard.
Cryptoki library	A library that implements the functions specified in this standard.
DER	Distinguished Encoding Rules, as defined in X.690.
DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
DSA	Digital Signature Algorithm, as defined in FIPS PUB 186-2.
EC	Elliptic Curve
ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
IV	Initialization Vector.
MAC	Message Authentication Code.
Mechanism	A process for implementing a cryptographic operation.
Object	An item that is stored on a token. May be data, a certificate, or a key.
PIN	Personal Identification Number.
PKCS	Public-Key Cryptography Standards.
PRF	Pseudo random function.
PTD	Personal Trusted Device, as defined in MeT-PTD
RSA	The RSA public-key cryptosystem.

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

Session A logical connection between an application and a token.

Slot A logical reader that potentially contains a token.

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.

TLS Transport Layer Security.

Token The logical view of a cryptographic device defined by Cryptoki.

User The person using an application that interfaces to Cryptoki.

UTF-8 Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.

WIM Wireless Identification Module.

WTLS Wireless Transport Layer Security.

5 Symbols and abbreviations

The following symbols are used in this standard:

Table 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 2, Prefixes

Prefix	Description
C_	Function
CK_	Data type or general constant
CKA_	Attribute

Prefix	Description
CKC_	Certificate type
CKD_	Key derivation function
CKF_	Bit flag
CKG_	Mask generation function
CKH_	Hardware feature type
CKK_	Key type
CKM_	Mechanism type
CKN_	Notification
CKO_	Object class
CKP_	Pseudo-random function
CKS_	Session state
CKR_	Return value
CKU_	User type
CKZ_	Salt/Encoding parameter source
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;

/* an 8-bit UTF-8 character */
typedef CK_BYTE CK_UTF8CHAR;

/* 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 some of these data types, as well as to the type `void`, which are implementation-dependent. These pointer types are:

```
CK_BYTE_PTR      /* Pointer to a CK_BYTE */
CK_CHAR_PTR      /* Pointer to a CK_CHAR */
CK_UTF8CHAR_PTR  /* Pointer to a CK_UTF8CHAR */
CK_ULONG_PTR     /* Pointer to a CK_ULONG */
CK_VOID_PTR      /* Pointer to a void */
```

Cryptoki also defines a pointer to a CK_VOID_PTR, which is implementation-dependent:

```
CK_VOID_PTR_PTR  /* Pointer to a CK_VOID_PTR */
```

In addition, Cryptoki defines a C-style NULL pointer, which is distinct from any valid pointer:

```
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 Cryptoki header files 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 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	' '

The **CK_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-8 allows internationalization while maintaining backward compatibility with the Local String definition of PKCS #11 version 2.01.

In Cryptoki, the **CK_BBOOL** data type is a Boolean type that can be true or false. A zero value means false, and a nonzero value means true. Similarly, an individual bit flag, **CKF_**..., can also be set (true) or unset (false). For convenience, Cryptoki defines the following macros for use with values of type **CK_BBOOL**:

```
#define CK_FALSE 0
#define CK_TRUE  1
```

For backwards compatibility, header files for this version of Cryptoki also defines TRUE and FALSE as (`CK_DISABLE_TRUE_FALSE` may be set by the application vendor):

```
#ifndef CK_DISABLE_TRUE_FALSE
#ifndef FALSE
#define FALSE CK_FALSE
#endif

#ifndef TRUE
#define TRUE CK_TRUE
#endif
#endif
```

6 General overview

6.1 Introduction

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, utilizes 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.

6.2 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 2743 and RFC 2744) and “Generic Cryptographic Service API” (GCS-API) from X/Open.

6.3 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 one or more cryptographic devices, on which some or all of the operations are actually performed. A user may or may not be associated with an application.

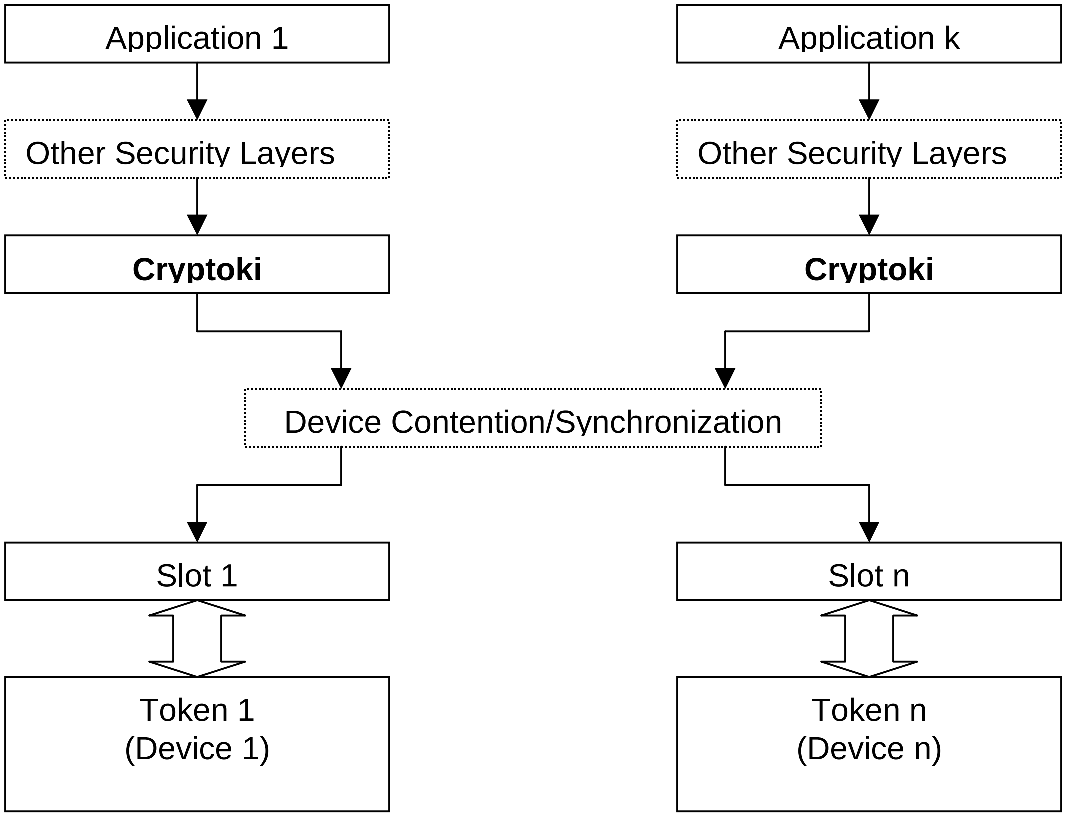

Figure 1, General Cryptoki Model

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 typically "present in the slot" 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 each 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 underlying "device" may be implemented entirely in software (for instance, as a process running on a server)—no special hardware is necessary.

Cryptoki is likely to be implemented as a library supporting the functions in the interface, and applications will be linked to the library. An application may be linked to Cryptoki directly; alternatively, Cryptoki can 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 and OS/2, and can be achieved without too much difficulty in UNIX and DOS systems.

The dynamic approach certainly has advantages as new libraries are made available, but from a security perspective, there are some drawbacks. In particular, if a 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, although code-signing techniques can prevent many of the security risks of dynamic linking. In any case, 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.

6.4 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 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:

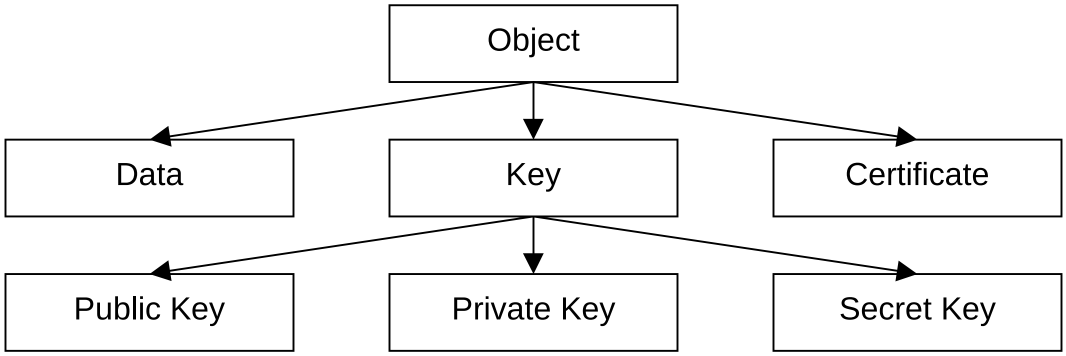

Figure 2, Object Hierarchy

Objects are also classified according to their lifetime and visibility. “Token objects” are visible to all applications connected to the token that have sufficient permission, and remain on 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. In addition, session objects are only visible to the application which created them.

Further classification defines access requirements. Applications are not required to log into the token to view “public objects”; however, to view “private objects”, a user must be authenticated to the token by a PIN or some other token-dependent method (for example, a biometric device).

See Table 6 on page 19 for further clarification on access to objects.

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 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 that are specific to a particular type of object, such as a modulus or exponent for RSA keys.

6.5 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, by some method outside the scope of this version of Cryptoki, how the normal user may be authenticated), and possibly to 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 3. Any translation to the device's requirements is left to the Cryptoki library. The following issues 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 supplied by some means other than through an application (*e.g.*, PINs entered via a PINpad on the token) are even more abstract. Cryptoki knows how to wait (if need be) for such a PIN to be supplied and used, and little more.

6.6 Applications and their use of Cryptoki

To Cryptoki, an application consists of a single address space and all the threads of control running in it. An application becomes a "Cryptoki application" by calling the Cryptoki function **C_Initialize** (see Section 11.4) from one of its threads; after this call is made, the application can call other Cryptoki functions. When the application is done using Cryptoki, it calls the Cryptoki function **C_Finalize** (see Section 11.4) and ceases to be a Cryptoki application.

6.6.1 Applications and processes

In general, on most platforms, the previous paragraph means that an application consists of a single process.

Consider a UNIX process **P** which becomes a Cryptoki application by calling **C_Initialize**, and then uses the `fork()` system call to create a child process **C**. Since **P** and **C** have separate address spaces (or will when one of them performs a write operation, if the operating system follows the copy-on-write paradigm), they are not part of the same application. Therefore, if **C** needs to use Cryptoki, it needs to perform its own **C_Initialize** call. Furthermore, if **C** needs to be logged into the token(s) that it will access via Cryptoki, it needs to log into them *even if **P** already logged in*, since **P** and **C** are completely separate applications.

In this particular case (when **C** is the child of a process which is a Cryptoki application), the behavior of Cryptoki is undefined if **C** tries to use it without its own **C_Initialize** call. Ideally, such an attempt would return the value CKR_CRYPTOKI_NOT_INITIALIZED; however, because of the way `fork()` works, insisting on this return value might have a bad impact on the performance of libraries. Therefore, the behavior of Cryptoki in this situation is left undefined. Applications should definitely *not* attempt to take advantage of any potential "shortcuts" which might (or might not!) be available because of this.

In the scenario specified above, **C** should actually call **C_Initialize** whether or not it needs to use Cryptoki; if it has no need to use Cryptoki, it should then call **C_Finalize** immediately thereafter. This (having the child immediately call **C_Initialize** and then call **C_Finalize** if the parent is using Cryptoki) is considered to be good Cryptoki programming practice, since it can prevent the existence of dangling duplicate resources that were created at the time of the `fork()` call; however, it is not required by Cryptoki.

6.6.2 Applications and threads

Some applications will access a Cryptoki library in a multi-threaded fashion. Cryptoki enables applications to provide information to libraries so that they can give appropriate support for multi-threading. In particular, when an application initializes a Cryptoki library with a call to **C_Initialize**, it can specify one of four possible multi-threading behaviors for the library:

1. The application can specify that it will not be accessing the library concurrently from multiple threads, and so the library need not worry about performing any type of locking for the sake of thread-safety.

2. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must be able to use native operation system synchronization primitives to ensure proper thread-safe behavior.

3. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must use a set of application-supplied synchronization primitives to ensure proper thread-safe behavior.

4. The application can specify that it *will* be accessing the library concurrently from multiple threads, and the library must use either the native operation system synchronization primitives or a set of application-supplied synchronization primitives to ensure proper thread-safe behavior.

The 3rd and 4th types of behavior listed above are appropriate for multi-threaded applications which are not using the native operating system thread model. The application-supplied synchronization primitives consist of four functions for handling mutex (*mut*ual *ex*clusion) objects in the application's threading model. Mutex objects are simple objects which can be in either of two states at any given time: unlocked or locked. If a call is made by a thread to lock a mutex which is already locked, that thread blocks (waits) until the mutex is unlocked; then it locks it and the call returns. If more than one thread is blocking on a particular mutex, and that mutex becomes unlocked, then exactly one of those threads will get the lock on the mutex and return control to the caller (the other blocking threads will continue to block and wait for their turn).

See Section 9.7 for more information on Cryptoki's view of mutex objects.

In addition to providing the above thread-handling information to a Cryptoki library at initialization time, an application can also specify whether or not application threads executing library calls may use native operating system calls to spawn new threads.

6.7 Sessions

Cryptoki requires that an application open one or more sessions with a token to gain 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.

After it opens a session, an application has access to the token's public objects. All threads of a given application have access to exactly the same sessions and the same session 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. That is, if a single application has multiple sessions open with a token, and it uses one of them to create a session object, then that session object is visible through any of that application's sessions. However, as soon as the session that was used to create the object is closed, that object is destroyed.

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 an application to have only a limited number of sessions—or only a limited number of read/write sessions-- 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 6.7.1 and Section 6.7.2.

6.7.1 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 the application has no previously open sessions that are logged in) or the "R/O User Functions" state (if the application already has an open session that is logged in). Note that read-only SO sessions do not exist.

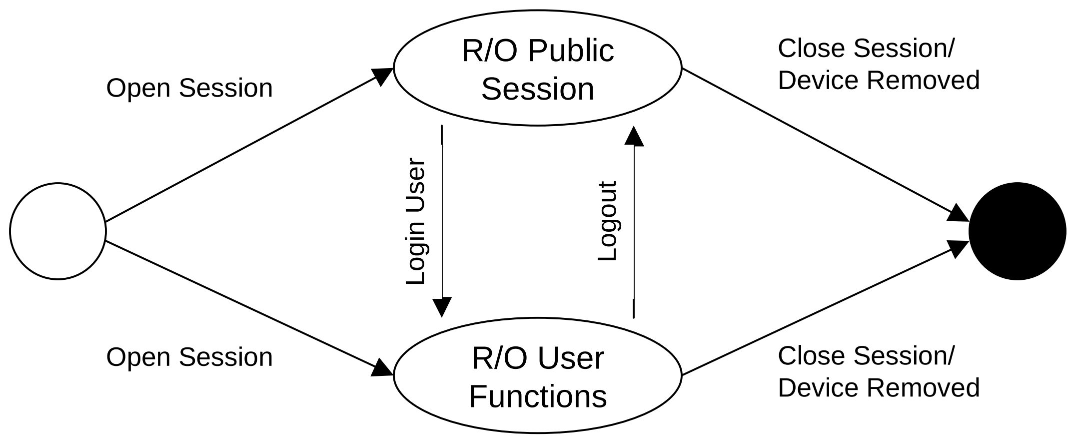

Figure 3, Read-Only Session States

The following table describes the session states:

Table 4, 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).

6.7.2 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 the application has no previously open sessions that are logged in), the “R/W User Functions” state (if the application already has an open session that the normal user is logged into), or the “R/W SO Functions” state (if the application already has an open session that the SO is logged into).

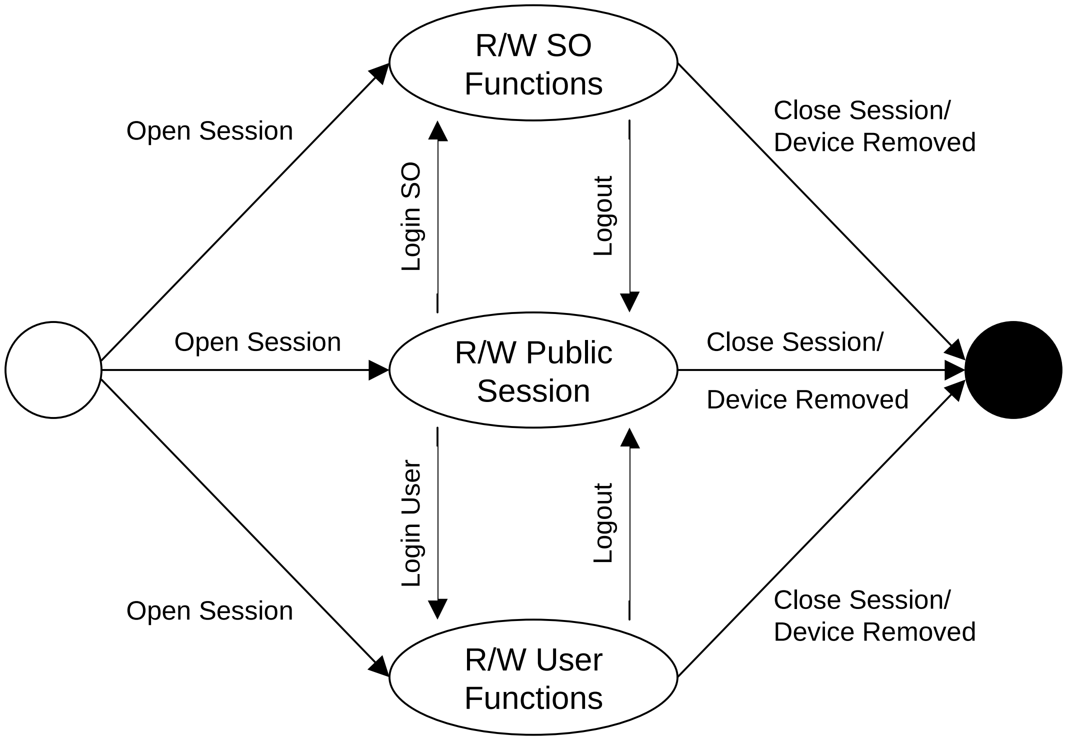

Figure 4, Read/Write Session States

The following table describes the session states:

Table 5, 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.

6.7.3 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 or delete a token object.

Table 6, 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	

As previously indicated, the access to a given session object which is shown in Table 6 is limited to sessions belonging to the application which owns that object (*i.e.*, which created that object).

6.7.4 Session events

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

Table 7, 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 (SO or normal user).
Close Session	the application closes the session or closes all sessions.
Device Removed	the device underlying the token has been removed from its slot.

When the device is removed, all sessions of all applications are automatically logged out. Furthermore, all sessions any applications have with the device are closed (this latter behavior was not present in Version 1.0 of Cryptoki)—an application cannot have a session with a token that is not present. Realistically, Cryptoki may not be constantly monitoring whether or not the token is present, and so the token's absence could conceivably not be noticed until a Cryptoki function is executed. If the token is re-inserted into the slot before that, Cryptoki might never know that it was missing.

In Cryptoki, 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 into 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.

6.7.5 Session handles and object handles

A session handle is a Cryptoki-assigned value that identifies a session. It is in many ways akin to a file handle, and is specified to functions to indicate which session the function should act on. All threads 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 can also be accomplished with that file handle by any other thread of the same application.

Cryptoki also has object handles, which are identifiers used to manipulate Cryptoki objects. Object handles are similar to session handles in the sense that visibility of a given object through an object handle is the same among all threads of a given application. R/O sessions, of course, 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. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

6.7.6 Capabilities of sessions

Very roughly speaking, there are three broad types of operations an open session can be used to perform: administrative operations (such as logging in); object management operations (such as creating or destroying an object on the token); and cryptographic operations (such as computing a message digest). Cryptographic operations sometimes require more than one function call to the Cryptoki API to complete. In general, a single session can perform only one operation at a time; for this reason, it may be desirable for a single application to open multiple sessions with a single token. For efficiency's sake, however, a single session on some tokens 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 are provided in Section 11.13.

A consequence of the fact that a single session can, in general, perform only one operation at a time is that *an application should never make multiple simultaneous function calls to Cryptoki which use a common session*. If multiple threads of an application attempt to use a common session concurrently in this fashion, Cryptoki does not define what happens. This means that if multiple threads of an application all need to use Cryptoki to access a particular token, it might be appropriate for each thread to have its own session with the token, unless the application can ensure by some other means (*e.g.*, by some locking mechanism) that no sessions are ever used by multiple threads simultaneously. This is true regardless of whether or not the Cryptoki library was initialized in a fashion which permits safe multi-threaded access to it. Even if it is safe to access the library from multiple threads simultaneously, it is still not necessarily safe to use *a particular session* from multiple threads simultaneously.

6.7.7 Example of use of sessions

We give here a detailed and lengthy example of how multiple applications can make use of sessions in a Cryptoki library. Despite the somewhat painful level of detail, we highly recommend reading through this example carefully to understand session handles and object handles.

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 Cryptoki library to access a single token **T**. Each application has two threads running: **A** has threads **A1** and **A2**, and **B** has threads **B1** and **B2**. We assume in what follows that there are no instances where multiple threads of a single application simultaneously use the same session, and that the events of our example occur in the order specified, without overlapping each other in time.

1. **A1** and **B1** each initialize the Cryptoki library by calling **C_Initialize** (the specifics of Cryptoki functions will be explained in Section 10.12). Note that exactly one call to **C_Initialize** should be made for each application (as opposed to one call for every thread, 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 into 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 code indicating that the existence of a R/O session has blocked this attempt to log in (CKR_SESSION_READ_ONLY_EXISTS).

5. **A2** logs the normal user into 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 (CKR_SESSION_HANDLE_INVALID).

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 (*i.e.*, **B2** receives a CKR_SESSION_HANDLE_INVALID error code).

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 into [**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 (CKR_SESSION_READ_WRITE_SO_EXISTS).

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, the attempt would fail with error code CKR_USER_NOT_LOGGED_IN or CKR_TEMPLATE_INCONSISTENT).

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 (CKR_SESSION_READ_ONLY).

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 (CKR_OBJECT_HANDLE_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. It returns a CKR_OBJECT_HANDLE_INVALID.

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 (*i.e.*, it would be a public session).

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.

6.8 Secondary authentication (Deprecated)

Note: This support may be present for backwards compatibility. Refer to PKCS11 V 2.11 for details.

6.9 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 8, 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_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur
	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
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

Category	Function	Description
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 functions	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 simultaneous multiple-part digesting and encryption operations
	C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations
	C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations
	C_DecryptVerifyUpdate	continues simultaneous multiple-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

Category	Function	Description
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	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
	C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
Callback function		application-supplied function to process notifications from Cryptoki

7 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 and/or certificates on the token as well, 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 “unextractable”. Sensitive keys cannot be revealed in plaintext off the token, and unextractable 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 unextractable objects 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.

An application may also want to be sure that the token is “legitimate” in some sense (for a variety of reasons, including export restrictions and basic security). 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. Some tokens may not even require any type of authentication to make use of its cryptographic functions.

8 Platform- and compiler-dependent directives for C or C++

There is a large array of Cryptoki-related data types which are defined in the Cryptoki header files. Certain packing- and pointer-related aspects of these types are platform- and compiler-dependent; these aspects are therefore resolved on a platform-by-platform (or compiler-by-compiler) basis outside of the Cryptoki header files by means of preprocessor directives.

This means that when writing C or C++ code, certain preprocessor directives must be issued before including a Cryptoki header file. These directives are described in the remainder of Section 8.

8.1 Structure packing

Cryptoki structures are packed to occupy as little space as is possible. In particular, on the Windows platforms, Cryptoki structures should be packed with 1-byte alignment. In a UNIX environment, it may or may not be necessary (or even possible) to alter the byte-alignment of structures.

8.2 Pointer-related macros

Because different platforms and compilers have different ways of dealing with different types of pointers, Cryptoki requires the following 6 macros to be set outside the scope of Cryptoki:

♦ **CK_PTR**

`CK_PTR` is the "indirection string" a given platform and compiler uses to make a pointer to an object. It is used in the following fashion:

```
typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

♦ **CK_DEFINE_FUNCTION**

`CK_DEFINE_FUNCTION(returnType, name)`, when followed by a parentheses-enclosed list of arguments and a function definition, defines a Cryptoki API function in a Cryptoki library. `returnType` is the return type of the function, and `name` is its name. It is used in the following fashion:

```
CK_DEFINE_FUNCTION(CK_RV, C_Initialize)(
  CK_VOID_PTR pReserved
)
{
  ...
}
```

♦ **CK_DECLARE_FUNCTION**

`CK_DECLARE_FUNCTION(returnType, name)`, when followed by a parentheses-enclosed list of arguments and a semicolon, declares a Cryptoki API function in a Cryptoki library. `returnType` is the return type of the function, and `name` is its name. It is used in the following fashion:

```
CK_DECLARE_FUNCTION(CK_RV, C_Initialize)(
  CK_VOID_PTR pReserved
);
```

♦ **CK_DECLARE_FUNCTION_POINTER**

`CK_DECLARE_FUNCTION_POINTER(returnType, name)`, when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to a Cryptoki API function in a Cryptoki library. `returnType` is the return type of the function, and `name` is its name. It can be used in either of the following fashions to define a function pointer variable, `myC_Initialize`, which can point to a **C_Initialize** function in a Cryptoki library (note that neither of the following code snippets actually *assigns* a value to `myC_Initialize`):


```
CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_Initialize)(
  CK_VOID_PTR pReserved
);
```

or:

```
typedef CK_DECLARE_FUNCTION_POINTER(CK_RV,
        myC_InitializeType)(
  CK_VOID_PTR pReserved
);
myC_InitializeType myC_Initialize;
```

♦ **CK_CALLBACK_FUNCTION**

`CK_CALLBACK_FUNCTION(returnType, name)`, when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to an application callback function that can be used by a Cryptoki API function in a Cryptoki library. `returnType` is the return type of the function, and `name` is its name. It can be used in either of the following fashions to define a function pointer variable, `myCallback`, which can point to an application callback which takes arguments `args` and returns a **CK_RV** (note that neither of the following code snippets actually *assigns* a value to `myCallback`):

```
CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);
```

or:

```
typedef CK_CALLBACK_FUNCTION(CK_RV,
        myCallbackType)(args);
myCallbackType myCallback;
```

♦ **NULL_PTR**

`NULL_PTR` is the value of a NULL pointer. In any ANSI C environment—and in many others as well—`NULL_PTR` should be defined simply as 0.

8.3 Sample platform- and compiler-dependent code

8.3.1 Win32

Developers using Microsoft Developer Studio 5.0 to produce C or C++ code which implements or makes use of a Win32 Cryptoki .dll might issue the following directives before including any Cryptoki header files:

```
#pragma pack(push, cryptoki, 1)

#define CK_IMPORT_SPEC __declspec(dllimport)
```

```
/* Define CRYPTOKI_EXPORTS during the build of cryptoki
 * libraries. Do not define it in applications.
 */
#ifdef CRYPTOKI_EXPORTS
#define CK_EXPORT_SPEC __declspec(dllexport)
#else
#define CK_EXPORT_SPEC CK_IMPORT_SPEC
#endif

/* Ensures the calling convention for Win32 builds */
#define CK_CALL_SPEC __cdecl

#define CK_PTR *

#define CK_DEFINE_FUNCTION(returnType, name) \
  returnType CK_EXPORT_SPEC CK_CALL_SPEC name

#define CK_DECLARE_FUNCTION(returnType, name) \
  returnType CK_EXPORT_SPEC CK_CALL_SPEC name

#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
  returnType CK_IMPORT_SPEC (CK_CALL_SPEC CK_PTR name)

#define CK_CALLBACK_FUNCTION(returnType, name) \
  returnType (CK_CALL_SPEC CK_PTR name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif
```

Hence the calling convention for all C_xxx functions should correspond to "cdecl" where function parameters are passed from right to left and the caller removes parameters from the stack when the call returns.

After including any Cryptoki header files, they might issue the following directives to reset the structure packing to its earlier value:

```
#pragma pack(pop, cryptoki)
```

8.3.2 Win16

Developers using a pre-5.0 version of Microsoft Developer Studio to produce C or C++ code which implements or makes use of a Win16 Cryptoki .dll might issue the following directives before including any Cryptoki header files:

```
#pragma pack(1)

#define CK_PTR far *
```

```
#define CK_DEFINE_FUNCTION(returnType, name) \
  returnType __export _far _pascal name

#define CK_DECLARE_FUNCTION(returnType, name) \
  returnType __export _far _pascal name

#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
  returnType __export _far _pascal (* name)

#define CK_CALLBACK_FUNCTION(returnType, name) \
  returnType _far _pascal (* name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif
```

8.3.3 Generic UNIX

Developers performing generic UNIX development might issue the following directives before including any Cryptoki header files:

```
#define CK_PTR *

#define CK_DEFINE_FUNCTION(returnType, name) \
  returnType name

#define CK_DECLARE_FUNCTION(returnType, name) \
  returnType name

#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
  returnType (* name)

#define CK_CALLBACK_FUNCTION(returnType, name) \
  returnType (* name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif
```

9 General data types

The general Cryptoki data types are described in the following subsections. 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 12.

A C or C++ source file in a Cryptoki application or library can define all these types (the types described here and the types that are specifically used for particular mechanism parameters) by including the top-level Cryptoki include file, `pkcs11.h`. `pkcs11.h`, in turn, includes the other Cryptoki include files, `pkcs11t.h` and `pkcs11f.h`. A source file can also include just `pkcs11t.h` (instead of `pkcs11.h`); this defines most (but not all) of the types specified here.

When including either of these header files, a source file must specify the preprocessor directives indicated in Section 8.

9.1 General information

Cryptoki represents general information with the following types:

- **CK_VERSION; CK_VERSION_PTR**

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or 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)

Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10. Table 9 below lists the major and minor version values for the officially published Cryptoki specifications.

Table 9, Major and minor version values for published Cryptoki specifications

Version	major	minor
1.0	0x01	0x00
2.01	0x02	0x01
2.10	0x02	0x0a
2.11	0x02	0x0b
2.20	0x02	0x14
2.30	0x02	0x1e

Minor revisions of the Cryptoki standard are always upwardly compatible within the same major version number.

CK_VERSION_PTR is a pointer to a **CK_VERSION**.

- ♦ **CK_INFO; CK_INFO_PTR**

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

```
typedef struct CK_INFO {
  CK_VERSION cryptokiVersion;
  CK_UTF8CHAR manufacturerID[32];
  CK_FLAGS flags;
  CK_UTF8CHAR 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 (‘ ‘). Should *not* be null-terminated.

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 (‘ ‘). Should *not* be null-terminated.

libraryVersion Cryptoki library version number

For libraries written to this document, the value of *cryptokiVersion* should match the version of this specification; the value of *libraryVersion* is the version number of the library software itself.

CK_INFO_PTR is a pointer to a **CK_INFO**.

♦ 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:

```
CKN_SURRENDER
```

The notifications have the following meanings:

CKN_SURRENDER	Cryptoki is surrendering the execution of a function executing in a session 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 (see Section 11.17.1).

9.2 Slot and token types

Cryptoki represents slot and token information with the following types:

♦ CK_SLOT_ID; CK_SLOT_ID_PTR

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

```
typedef CK_ULONG CK_SLOT_ID;
```

A list of **CK_SLOT_ID**s is returned by **C_GetSlotList**. A priori, *any* value of **CK_SLOT_ID** can be a valid slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a slot, however.

CK_SLOT_ID_PTR is a pointer to a **CK_SLOT_ID**.

♦ CK_SLOT_INFO; CK_SLOT_INFO_PTR

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

```
typedef struct CK_SLOT_INFO {
  CK_UTF8CHAR slotDescription[64];
  CK_UTF8CHAR 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 (‘ ‘). Should *not* be null-terminated.
manufacturerID	ID of the slot manufacturer. Must be padded with the blank character (‘ ‘). Should *not* be null-terminated.
flags	bits flags that provide capabilities of the slot. The flags are defined below
hardwareVersion	version number of the slot’s hardware
firmwareVersion	version number of the slot’s firmware

The following table defines the *flags* field:

Table 10, 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”

For a given slot, the value of the **CKF_REMOVABLE_DEVICE** flag *never changes*. In addition, if this flag is not set for a given slot, then the **CKF_TOKEN_PRESENT** flag for that slot is *always* set. That is, if a slot does not support a removable device, then that slot always has a token in it.

CK_SLOT_INFO_PTR is a pointer to a **CK_SLOT_INFO**.

♦ CK_TOKEN_INFO; CK_TOKEN_INFO_PTR

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

```
typedef struct CK_TOKEN_INFO {
  CK_UTF8CHAR label[32];
  CK_UTF8CHAR manufacturerID[32];
  CK_UTF8CHAR 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 ulFreePrivateMemory;
  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 (‘ ‘). Should *not* be null-terminated.
manufacturerID	ID of the device manufacturer. Must be padded with the blank character (‘ ‘). Should *not* be null-terminated.
model	model of the device. Must be padded with the blank character (‘ ‘). Should *not* be null-terminated.
serialNumber	character-string serial number of the device. Must be padded with the blank character (‘ ‘). Should *not* be null-terminated.
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 by a single application (see note below)

ulSessionCount	number of sessions that this application currently has open with the token (see note below)
ulMaxRwSessionCount	maximum number of read/write sessions that can be opened with the token at one time by a single application (see note below)
ulRwSessionCount	number of read/write sessions that this application currently has open with the token (see note below)
ulMaxPinLen	maximum length in bytes of the PIN
ulMinPinLen	minimum length in bytes of the PIN
ulTotalPublicMemory	the total amount of memory on the token in bytes in which public objects may be stored (see note below)
ulFreePublicMemory	the amount of free (unused) memory on the token in bytes for public objects (see note below)
ulTotalPrivateMemory	the total amount of memory on the token in bytes in which private objects may be stored (see note below)
ulFreePrivateMemory	the amount of free (unused) memory on the token in bytes for private objects (see note below)
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* field:

Table 11, 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 (see below)
CKF_LOGIN_REQUIRED	0x00000004	True if there are some cryptographic functions that a user must be logged in to perform
CKF_USER_PIN_INITIALIZED	0x00000008	True if the normal user’s PIN has been initialized
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_PROTECTED_AUTHENTICATION_PATH	0x00000100	True if token has a “protected authentication path”, whereby a user can log into 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 11.13)
CKF_TOKEN_INITIALIZED	0x00000400	True if the token has been initialized using C_InitToken or an equivalent mechanism outside the scope of this standard. Calling C_InitToken when this flag is set will cause the token to be reinitialized.

Bit Flag	Mask	Meaning
CKF_SECONDARY_AUTHENTICATION	0x00000800	True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag)
CKF_USER_PIN_COUNT_LOW	0x00010000	True if an incorrect user login PIN has been entered at least once since the last successful authentication.
CKF_USER_PIN_FINAL_TRY	0x00020000	True if supplying an incorrect user PIN will cause it to become locked.
CKF_USER_PIN_LOCKED	0x00040000	True if the user PIN has been locked. User login to the token is not possible.
CKF_USER_PIN_TO_BE_CHANGED	0x00080000	True if the user PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_SO_PIN_COUNT_LOW	0x00100000	True if an incorrect SO login PIN has been entered at least once since the last successful authentication.
CKF_SO_PIN_FINAL_TRY	0x00200000	True if supplying an incorrect SO PIN will cause it to become locked.
CKF_SO_PIN_LOCKED	0x00400000	True if the SO PIN has been locked. SO login to the token is not possible.
CKF_SO_PIN_TO_BE_CHANGED	0x00800000	True if the SO PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_ERROR_STATE	0x01000000	True if the token failed a FIPS 140-2 self-test and entered an error state.

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 others:

- Creating/modifying/deleting any object on the token.
- Creating/modifying/deleting a token object on the token.
- Changing the SO's PIN.
- Changing the normal user's PIN.

The token may change the value of the **CKF_WRITE_PROTECTED** flag depending on the session state to implement its object management policy. For instance, the token may set the **CKF_WRITE_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy that does not allow any objects, public or private, to be created, modified, or deleted unless the user has successfully called C_Login.

The **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_FINAL_TRY**, and **CKF_SO_PIN_FINAL_TRY** flags may always be set to false if the token does not support the functionality or will not reveal the information because of its security policy.

The **CKF_USER_PIN_TO_BE_CHANGED** and **CKF_SO_PIN_TO_BE_CHANGED** flags may always be set to false if the token does not support the functionality. If a PIN is set to the default value, or has expired, the appropriate **CKF_USER_PIN_TO_BE_CHANGED** or **CKF_SO_PIN_TO_BE_CHANGED** flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed, but only the C_SetPIN function can be called. Calling any other function that required the user to be logged in will cause CKR_PIN_EXPIRED to be returned until C_SetPIN is called successfully.

Note: The fields *ulMaxSessionCount*, *ulSessionCount*, *ulMaxRwSessionCount*, *ulRwSessionCount*, *ulTotalPublicMemory*, *ulFreePublicMemory*, *ulTotalPrivateMemory*, and *ulFreePrivateMemory* can have the special value CK_UNAVAILABLE_INFORMATION, which means that the token and/or library is unable or unwilling to provide that information. In addition, the fields *ulMaxSessionCount* and *ulMaxRwSessionCount* can have the special value CK_EFFECTIVELY_INFINITE, which means that there is no practical limit on the number of sessions (resp. R/W sessions) an application can have open with the token.

It is important to check these fields for these special values. This is particularly true for CK_EFFECTIVELY_INFINITE, since an application seeing this value in the *ulMaxSessionCount* or *ulMaxRwSessionCount* field would otherwise conclude that it can't open *any* sessions with the token, which is far from being the case.

The upshot of all this is that the correct way to interpret (for example) the *ulMaxSessionCount* field is something along the lines of the following:


```
CK_TOKEN_INFO info;

.
.
if ((CK_LONG) info.ulMaxSessionCount
    == CK_UNAVAILABLE_INFORMATION) {
  /* Token refuses to give value of ulMaxSessionCount */
  .
  .
} else if (info.ulMaxSessionCount ==
        CK_EFFECTIVELY_INFINITE) {
  /* Application can open as many sessions as it wants */
  .
  .
} else {
  /* ulMaxSessionCount really does contain what it should
        */
  .
  .
}
```

CK_TOKEN_INFO_PTR is a pointer to a **CK_TOKEN_INFO**.

9.3 Session types

Cryptoki represents session information with the following types:

♦ CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR

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

```
typedef CK_ULONG CK_SESSION_HANDLE;
```

Valid session handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

CK_SESSION_HANDLE_PTR is a pointer to a **CK_SESSION_HANDLE**.

♦ CK_USER_TYPE

CK_USER_TYPE holds the types of Cryptoki users described in Section 6.5, and, in addition, a context-specific type described in Section 10.9. It is defined as follows:

```
typedef CK_ULONG CK_USER_TYPE;
```

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

```
CKU_SO
CKU_USER
CKU_CONTEXT_SPECIFIC
```

♦ CK_STATE

CK_STATE holds the session state, as described in Sections 6.7.1 and 6.7.2. It is defined as follows:

```
typedef CK_ULONG CK_STATE;
```

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

```
CKS_RO_PUBLIC_SESSION
CKS_RO_USER_FUNCTIONS
CKS_RW_PUBLIC_SESSION
CKS_RW_USER_FUNCTIONS
CKS_RW_SO_FUNCTIONS
```

♦ CK_SESSION_INFO; CK_SESSION_INFO_PTR

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* field:

Table 12, Session Information Flags

Bit Flag	**Mask**	**Meaning**
CKF_RW_SESSION	0x00000002	True if the session is read/write; false if the session is read-only
CKF_SERIAL_SESSION	0x00000004	This flag is provided for backward compatibility, and should always be set to true

CK_SESSION_INFO_PTR is a pointer to a **CK_SESSION_INFO**.

9.4 Object types

Cryptoki represents object information with the following types:

♦ CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR

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 by an application, Cryptoki assigns it an object handle for that application's 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 access 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.

Valid object handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

CK_OBJECT_HANDLE_PTR is a pointer to a **CK_OBJECT_HANDLE**.

♦ CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR

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;
```

Object classes are defined with the objects that use them. The type is specified on an object through the CKA_CLASS attribute of the object.

Vendor defined values for this type may also be specified.

```
CKO_VENDOR_ DEFINED
```

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 is a pointer to a **CK_OBJECT_CLASS**.

♦ CK_HW_FEATURE_TYPE

CK_HW_FEATURE_TYPE is a value that identifies a hardware feature type of a device. It is defined as follows:

```
typedef CK_ULONG CK_HW_FEATURE_TYPE;
```

Hardware feature types are defined with the objects that use them. The type is specified on an object through the CKA_HW_FEATURE_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

```
CKH_VENDOR_DEFINED
```

Feature types **CKH_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their feature types through the PKCS process.

♦ 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;
```

Key types are defined with the objects and mechanisms that use them. The key type is specified on an object through the CKA_KEY_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

```
CKK_VENDOR_DEFINED
```

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;
```

Certificate types are defined with the objects and mechanisms that use them. The certificate type is specified on an object through the CKA_CERTIFICATE_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

```
CKC_VENDOR_DEFINED
```

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;
```

Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an object as a list of type, length value items. These are often specified as an attribute template.

Vendor defined values for this type may also be specified.

```
CKA_VENDOR_DEFINED
```

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_PTR

CK_ATTRIBUTE is a structure that includes the type, value, and length 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 *ulValueLen* = 0, and the value of *pValue* is irrelevant. An array of **CK_ATTRIBUTE**s is called a "template" and is used for creating, manipulating and searching for objects. The order of the attributes in a template *never* matters, even if the template contains vendor-specific attributes. Note that *pValue* 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 (*i.e.*, without word-alignment errors).

CK_ATTRIBUTE_PTR is a pointer to a **CK_ATTRIBUTE**.

♦ 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 3, not the literal byte values.

When a Cryptoki object carries an attribute of this type, and the default value of the attribute is specified to be "empty," then Cryptoki libraries shall set the attribute's *ulValueLen* to 0.

Note that implementations of previous versions of Cryptoki may have used other methods to identify an "empty" attribute of type CK_DATE, and that applications that needs to interoperate with these libraries therefore have to be flexible in what they accept as an empty value.

9.5 Data types for mechanisms

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

♦ CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR

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

```
typedef CK_ULONG CK_MECHANISM_TYPE;
```

Mechanism types are defined with the objects and mechanism descriptions that use them.

Vendor defined values for this type may also be specified.

```
CKM_VENDOR_DEFINED
```

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 is a pointer to a **CK_MECHANISM_TYPE**.

♦ CK_MECHANISM; CK_MECHANISM_PTR

CK_MECHANISM is a structure that specifies a particular mechanism and any parameters it requires. 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
ulParameterLen	length in bytes of the parameter

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

CK_MECHANISM_PTR is a pointer to a **CK_MECHANISM**.

♦ CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR

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 (whether this is measured in bits or in bytes is mechanism-dependent)
ulMaxKeySize	the maximum size of the key for the mechanism (whether this is measured in bits or in bytes is mechanism-dependent)
flags	bit flags specifying mechanism capabilities

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

The following table defines the *flags* field:

Table 13, 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 there is an extension to the flags; false if no extensions. Must be false for this version.

CK_MECHANISM_INFO_PTR is a pointer to a **CK_MECHANISM_INFO**.

9.6 Function types

Cryptoki represents information about functions with the following data types:

♦ 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;
```

Vendor defined values for this type may also be specified.

```
CKR_VENDOR_DEFINED
```

Section 11.1 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 defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, 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

♦ CK_C_XXX

Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the Cryptoki API (see Section 10.12 for detailed information about each of them), Cryptoki defines a type **CK_C_XXX**, which is a pointer to a function with the same arguments and return value as **C_XXX** has. An appropriately-set variable of type **CK_C_XXX** may be used by an application to call the Cryptoki function **C_XXX**.

- **CK_FUNCTION_LIST; CK_FUNCTION_LIST_PTR; CK_FUNCTION_LIST_PTR_PTR**

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_GetOperationState;
  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_EncryptUpdate 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_SignRecover;
    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_C_WaitForSlotEvent C_WaitForSlotEvent;
  } CK_FUNCTION_LIST;
```

Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section 11.2). The value that this pointer points to can be used by an application to quickly find out where the executable code for each function in the Cryptoki API is located. *Every function in the Cryptoki API must have an entry point defined in the Cryptoki library's* ***CK_FUNCTION_LIST*** *structure*. If a particular function in the Cryptoki API is not supported by a library, then the function pointer for that function in the library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns CKR_FUNCTION_NOT_SUPPORTED.

In this structure 'version' is the cryptoki specification version number. It should match the value of 'cryptokiVersion' returned in the CK_INFO structure.

An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST** structure. Whether or not it can, it should never attempt to do so.

CK_FUNCTION_LIST_PTR is a pointer to a **CK_FUNCTION_LIST**.

CK_FUNCTION_LIST_PTR_PTR is a pointer to a **CK_FUNCTION_LIST_PTR**.

9.7 Locking-related types

The types in this section are provided solely for applications which need to access Cryptoki from multiple threads simultaneously. *Applications which will not do this need not use any of these types.*

♦ CK_CREATEMUTEX

CK_CREATEMUTEX is the type of a pointer to an application-supplied function which creates a new mutex object and returns a pointer to it. It is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX)(
  CK_VOID_PTR_PTR ppMutex
);
```

Calling a **CK_CREATEMUTEX** function returns the pointer to the new mutex object in the location pointed to by *ppMutex*. Such a function should return one of the following values: CKR_OK, CKR_GENERAL_ERROR, CKR_HOST_MEMORY.

♦ CK_DESTROYMUTEX

CK_DESTROYMUTEX is the type of a pointer to an application-supplied function which destroys an existing mutex object. It is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX)(
  CK_VOID_PTR pMutex
);
```

The argument to a **CK_DESTROYMUTEX** function is a pointer to the mutex object to be destroyed. Such a function should return one of the following values: CKR_OK, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MUTEX_BAD.

♦ CK_LOCKMUTEX and CK_UNLOCKMUTEX

CK_LOCKMUTEX is the type of a pointer to an application-supplied function which locks an existing mutex object. **CK_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- If a **CK_LOCKMUTEX** function is called on a mutex which is not locked, the calling thread obtains a lock on that mutex and returns.
- If a **CK_LOCKMUTEX** function is called on a mutex which is locked by some thread other than the calling thread, the calling thread blocks and waits for that mutex to be unlocked.

- If a **CK_LOCKMUTEX** function is called on a mutex which is locked by the calling thread, the behavior of the function call is undefined.

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:

  - If exactly one thread was blocking on that particular mutex, then that thread stops blocking, obtains a lock on that mutex, and its **CK_LOCKMUTEX** call returns.

  - If more than one thread was blocking on that particular mutex, then exactly one of the blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on the mutex, and its **CK_LOCKMUTEX** call returns. All other threads blocking on that particular mutex continue to block.

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is not locked, then the function call returns the error code CKR_MUTEX_NOT_LOCKED.

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by some thread other than the calling thread, the behavior of the function call is undefined.

CK_LOCKMUTEX is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX)(
  CK_VOID_PTR pMutex
);
```

The argument to a **CK_LOCKMUTEX** function is a pointer to the mutex object to be locked. Such a function should return one of the following values: CKR_OK, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MUTEX_BAD.

CK_UNLOCKMUTEX is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX)(
  CK_VOID_PTR pMutex
);
```

The argument to a **CK_UNLOCKMUTEX** function is a pointer to the mutex object to be unlocked. Such a function should return one of the following values: CKR_OK, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MUTEX_BAD, CKR_MUTEX_NOT_LOCKED.

♦ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

CK_C_INITIALIZE_ARGS is a structure containing the optional arguments for the **C_Initialize** function. For this version of Cryptoki, these optional arguments are all concerned with the way the library deals with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

```
typedef struct CK_C_INITIALIZE_ARGS {
  CK_CREATEMUTEX CreateMutex;
  CK_DESTROYMUTEX DestroyMutex;
  CK_LOCKMUTEX LockMutex;
  CK_UNLOCKMUTEX UnlockMutex;
  CK_FLAGS flags;
  CK_VOID_PTR pReserved;
} CK_C_INITIALIZE_ARGS;
```

The fields of the structure have the following meanings:

CreateMutex	pointer to a function to use for creating mutex objects
DestroyMutex	pointer to a function to use for destroying mutex objects
LockMutex	pointer to a function to use for locking mutex objects
UnlockMutex	pointer to a function to use for unlocking mutex objects
flags	bit flags specifying options for **C_Initialize**; the flags are defined below
pReserved	reserved for future use. Should be NULL_PTR for this version of Cryptoki

The following table defines the *flags* field:

Table 14, C_Initialize Parameter Flags

Bit Flag	**Mask**	**Meaning**
CKF_LIBRARY_CANT_CREATE_OS_THREADS	0x00000001	True if application threads which are executing calls to the library may *not* use native operating system calls to spawn new threads; false if they may
CKF_OS_LOCKING_OK	0x00000002	True if the library can use the native operation system threading model for locking; false otherwise

CK_C_INITIALIZE_ARGS_PTR is a pointer to a **CK_C_INITIALIZE_ARGS**.

10 Objects

Cryptoki recognizes a number of classes of objects, as defined in the **CK_OBJECT_CLASS** data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and some of the attributes they support:

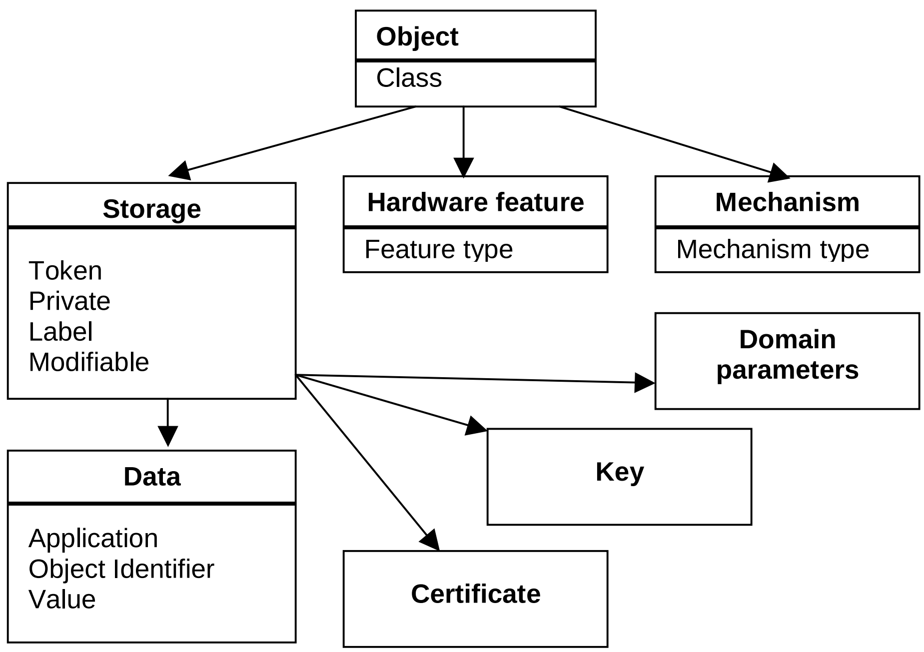

Figure 5, Object Attribute Hierarchy

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

Objects are always "well-formed" in Cryptoki—that is, an object always contains all required attributes, and the attributes are always consistent with one another from the time the object is created. This contrasts 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.

Tables throughout most of Section 10 define each Cryptoki attribute 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**). Attribute values 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	an unpadded string of **CK_CHAR**s (see Table 3) with no null-termination
RFC2279 string	an unpadded string of **CK_UTF8CHARs** with no null-termination

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.

In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of Cryptoki attributes. 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. A given object has a single value for each attribute it possesses, even if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes whose meanings and values are not specified by Cryptoki.

10.1 Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 11.14) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see Section 12). In any case, all the required 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 itself.

10.1.1 Creating objects

Objects may be created with the Cryptoki functions **C_CreateObject** (see Section 11.7), **C_GenerateKey**, **C_GenerateKeyPair**, **C_UnwrapKey**, and **C_DeriveKey** (see Section 11.14). In addition, copying an existing object (with the function **C_CopyObject**) also creates a new object, but we consider this type of object creation separately in Section 10.1.3.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the error code CKR_ATTRIBUTE_TYPE_INVALID. An attribute is valid if it is either one of the attributes described in the Cryptoki specification or an additional vendor-specific attribute supported by the library and token.

2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with the error code CKR_ATTRIBUTE_VALUE_INVALID. The valid values for Cryptoki attributes are described in the Cryptoki specification.

3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the error code CKR_ATTRIBUTE_READ_ONLY. Whether or not a given Cryptoki attribute is read-only is explicitly stated in the Cryptoki specification; however, a particular library and token may be even more restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-only is obviously outside the scope of Cryptoki.

4. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to fully specify the object to create, then the attempt should fail with the error code CKR_TEMPLATE_INCOMPLETE.

5. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are inconsistent, then the attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT. A set of attribute values is inconsistent if not all of its members can be satisfied simultaneously *by the token*, although each value individually is valid in Cryptoki. One example of an inconsistent template would be using a template which specifies two different values for the same attribute. Another example would be trying to create a secret key object with an attribute which is appropriate for various types of public keys or private keys, but not for secret keys. A final example would be a template with an attribute that violates some token specific requirement. Note that this final example of an inconsistent template is token-dependent—on a different token, such a template might *not* be inconsistent.

6. If the supplied template specifies the same value for a particular attribute more than once (or the template specifies the same value for a particular attribute that the object-creation function itself contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to create an object can either succeed—thereby creating the same object that would have been created if the multiply-specified attribute had only appeared once—or it can fail with error code CKR_TEMPLATE_INCONSISTENT. Library developers are encouraged to make

their libraries behave as though the attribute had only appeared once in the template; application developers are strongly encouraged never to put a particular attribute into a particular template more than once.

If more than one of the situations listed above applies to an attempt to create an object, then the error code returned from the attempt can be any of the error codes from above that applies.

10.1.2 Modifying objects

Objects may be modified with the Cryptoki function **C_SetAttributeValue** (see Section 11.7). The template supplied to **C_SetAttributeValue** can contain new values for attributes which the object already possesses; values for attributes which the object does not yet possess; or both.

Some attributes of an object may be modified after the object has been created, and some may not. In addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable might depend on the values of certain attributes of the object. For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE, but not the other way around.

All the scenarios in Section 10.1.1—and the error codes they return—apply to modifying objects with **C_SetAttributeValue**, except for the possibility of a template being incomplete.

10.1.3 Copying objects

`Unless an object's CKA_COPYABLE (see table 21) attribute is set to CK_FALSE, it` may be copied with the Cryptoki function **C_CopyObject** (see Section 11.7). In the process of copying an object, **C_CopyObject** also modifies the attributes of the newly-created copy according to an application-supplied template.

The Cryptoki attributes which can be modified during the course of a **C_CopyObject** operation are the same as the Cryptoki attributes which are described as being modifiable, plus the three special attributes **CKA_TOKEN**, **CKA_PRIVATE**, and **CKA_MODIFIABLE**. To be more precise, these attributes are modifiable during the course of a **C_CopyObject** operation *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes during the course of a **C_CopyObject** operation. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable during the course of a **C_CopyObject** operation might depend on the values of certain attributes of the object.

For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE during the course of a **C_CopyObject** operation, but not the other way around.

```
If the CKA_COPYABLE attribute of the object to be copied is set to
CK_FALSE, C_CopyObject returns CKR_COPY_PROHIBITED.  Otherwise, the
scenarios described in 10.1.1 - and the error codes they return - apply
to copying objects with C_CopyObject, except for the possibility of a
template being incomplete.
```

10.2 Common attributes

Table 15, Common footnotes for object attribute tables

[1] Must be specified when object is created with **C_CreateObject**.
[2] Must *not* be specified when object is created with **C_CreateObject**.
[3] Must be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
[4] Must *not* be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
[5] Must be specified when object is unwrapped with **C_UnwrapKey**.
[6] Must *not* be specified when object is unwrapped with **C_UnwrapKey**.
[7] Cannot be revealed if object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.
[8] May be modified after object is created with a **C_SetAttributeValue** call, or in the process of copying object with a **C_CopyObject** call. However, it is possible that a particular token may not permit modification of the attribute during the course of a **C_CopyObject** call.
[9] Default value is token-specific, and may depend on the values of other attributes.
[10] Can only be set to CK_TRUE by the SO user.
[11] Attribute cannot be changed once set to CK_TRUE. It becomes a read only attribute.
[12] Attribute cannot be changed once set to CK_FALSE. It becomes a read only attribute.

Table 16, Common Object Attributes

Attribute	Data Type	Meaning
CKA_CLASS[1]	CK_OBJECT_CLASS	Object class (type)

- Refer to table Table 15 for footnotes

The above table defines the attributes common to all objects.

10.3 Hardware Feature Objects

10.3.1 Definitions

This section defines the object class CKO_HW_FEATURE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.3.2 Overview

Hardware feature objects (**CKO_HW_FEATURE**) represent features of the device. They provide an easily expandable method for introducing new value-based features to the cryptoki interface.

When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, hardware feature objects are not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_HW_FEATURE**. This protects applications written to previous versions of cryptoki from finding objects that they do not understand.

Table 17, Hardware Feature Common Attributes

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE[1]	CK_HW_FEATURE	Hardware feature (type)

- Refer to table Table 15 for footnotes

10.3.3 Clock

10.3.3.1 Definition

The **CKA_HW_FEATURE_TYPE** attribute takes the value **CKH_CLOCK** of type **CK_HW_FEATURE.**

10.3.3.2 Description

Clock objects represent real-time clocks that exist on the device. This represents the same clock source as the **utcTime** field in the **CK_TOKEN_INFO** structure.

Table 18, Clock Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE	CK_CHAR[16]	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 **CKA_VALUE** attribute may be set using the **C_SetAttributeValue** function if permitted by the device. The session used to set the time must be logged in. The device may require the SO to be the user logged in to modify the time value. **C_SetAttributeValue** will return the error CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.

10.3.4 Monotonic Counter Objects

10.3.4.1 Definition

The **CKA_HW_FEATURE_TYPE** attribute takes the value **CKH_MONOTONIC_COUNTER** of type **CK_HW_FEATURE.**

10.3.4.2 Description

Monotonic counter objects represent hardware counters that exist on the device. The counter is guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an application for generating serial numbers to get some assurance of uniqueness per token.

Table 19, Monotonic Counter Attributes

Attribute	Data Type	Meaning
CKA_RESET_ON_INIT[1]	CK_BBOOL	The value of the counter will reset to a previously returned value if the token is initialized using **C_InitToken**.
CKA_HAS_RESET[1]	CK_BBOOL	The value of the counter has been reset at least once at some point in time.
CKA_VALUE[1]	Byte Array	The current version of the monotonic counter. The value is returned in big endian order.

[1]Read Only

The **CKA_VALUE** attribute may not be set by the client.

10.3.5 User Interface Objects

10.3.5.1 Definition

The **CKA_HW_FEATURE_TYPE** attribute takes the value **CKH_USER_INTERFACE** of type **CK_HW_FEATURE.**

10.3.5.2 Description

User interface objects represent the presentation capabilities of the device.

Table 20, User Interface Object Attributes

Attribute	Data type	Meaning
CKA_PIXEL_X	CK_ULONG	Screen resolution (in pixels) in X-axis (e.g. 1280)
CKA_PIXEL_Y	CK_ULONG	Screen resolution (in pixels) in Y-axis (e.g. 1024)
CKA_RESOLUTION	CK_ULONG	DPI, pixels per inch
CKA_CHAR_ROWS	CK_ULONG	For character-oriented displays; number of character rows (e.g. 24)
CKA_CHAR_COLUMNS	CK_ULONG	For character-oriented displays: number of character columns (e.g. 80). If display is of proportional-font type, this is the width of the display in "em"-s (letter "M"), see CC/PP Struct.
CKA_COLOR	CK_BBOOL	Color support
CKA_BITS_PER_PIXEL	CK_ULONG	The number of bits of color or grayscale information per pixel.
CKA_CHAR_SETS	RFC 2279 string	String indicating supported character sets, as defined by IANA MIBenum sets (www.iana.org). Supported character sets are separated with ";". E.g. a token supporting iso-8859-1 and us-ascii would set the attribute value to "`4;3`".
CKA_ENCODING_METHODS	RFC 2279 string	String indicating supported content transfer encoding methods, as defined by IANA (www.iana.org). Supported methods are separated with ";". E.g. a token supporting 7bit, 8bit and base64 could set the attribute value to "`7bit;8bit;base64`".
CKA_MIME_TYPES	RFC 2279 string	String indicating supported (presentable) MIME-types, as defined by IANA (www.iana.org). Supported types are separated with ";". E.g. a token supporting MIME types "a/b", "a/c" and "a/d" would set the attribute value to "`a/b;a/c;a/d`".

The selection of attributes, and associated data types, has been done in an attempt to stay as aligned with RFC 2534 and CC/PP Struct as possible. The special value

CK_UNAVAILABLE_INFORMATION may be used for CK_ULONG-based attributes when information is not available or applicable.

None of the attribute values may be set by an application.

The value of the **CKA_ENCODING_METHODS** attribute may be used when the application needs to send MIME objects with encoded content to the token.

10.4 Storage Objects

This is not an object class, hence no CKO_ definition is required. It is a category of object classes with common attributes for the object classes that follow.

Table 21, Common Storage Object Attributes

Attribute	Data Type	Meaning
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_COPYABLE	CK_BBOOL	CK_TRUE if object can be copied using C_CopyObject. Defaults to CK_TRUE. Can't be set to TRUE once it is set to FALSE.

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, however.)

The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

When the **CKA_PRIVATE** attribute is CK_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.

The value of the CKA_COPYABLE attribute determines whether or not an object can be copied. This attribute can be used in conjunction with CKA_MODIFIABLE to prevent changes to the permitted usages of keys and other objects.

10.5 Data objects

10.5.1 Definitions

This section defines the object class CKO_DATA for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.5.2 Overview

Data objects (object class **CKO_DATA**) hold information defined by an application. Other than providing access to it, 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 defined for this object class:

Table 22, Data Object Attributes

Attribute	Data type	Meaning
CKA_APPLICATION	RFC2279 string	Description of the application that manages the object (default empty)
CKA_OBJECT_ID	Byte Array	DER-encoding of the object identifier indicating the data object type (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)

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

The **CKA_OBJECT_ID** attribute provides an application independent and expandable way to indicate the type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier matches the data value.

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

```
CK_OBJECT_CLASS class = CKO_DATA;
CK_UTF8CHAR label[] = “A data object”;
CK_UTF8CHAR application[] = “An application”;
CK_BYTE data[] = “Sample data”;
CK_BBOOL true = CK_TRUE;
```

```
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &class, sizeof(class)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_APPLICATION, application, sizeof(application)-1},
  {CKA_VALUE, data, sizeof(data)}
};
```

10.6 Certificate objects

10.6.1 Definitions

This section defines the object class CKO_CERTIFICATE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.6.2 Overview

Certificate objects (object class **CKO_CERTIFICATE**) hold public-key or attribute 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 defined for this object class:

Table 23, Common Certificate Object Attributes

Attribute	**Data type**	**Meaning**
CKA_CERTIFICATE_TYPE[1]	CK_CERTIFICATE_TYPE	Type of certificate
CKA_TRUSTED[10]	CK_BBOOL	The certificate can be trusted for the application that it was created.
CKA_CERTIFICATE_CATEGORY	CK_ULONG	Categorization of the certificate: 0 = unspecified (default value), 1 = token user, 2 = authority, 3 = other entity
CKA_CHECK_VALUE	Byte array	Checksum
CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)
CKA_END_DATE	CK_DATE	End date for the certificate (default empty)

- Refer to table Table 15 for footnotes

The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of Cryptoki supports the following certificate types:

- X.509 public key certificate

- WTLS public key certificate
- X.509 attribute certificate

The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It must be set by a token initialization application or by the token's SO. Trusted certificates cannot be modified.

The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user certificate for which the corresponding private key is available on the token ("token user"), a CA certificate ("authority"), or an other end-entity certificate ("other entity"). This attribute may not be modified after an object is created.

The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to the categorization of the certificates. A certificate in the certificates CDF will be marked with category "token user". A certificate in the trustedCertificates CDF or in the usefulCertificates CDF will be marked with category "authority" or "other entity" depending on the CommonCertificateAttribute.authority attribute and the **CKA_TRUSTED** attribute indicates if it belongs to the trustedCertificates or usefulCertificates CDF.

CKA_CHECK_VALUE: The value of this attribute is derived from the certificate by taking the first three bytes of the SHA-1 hash of the certificate object's CKA_VALUE attribute.

The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not attach any special meaning to them. When present, the application is responsible to set them to values that match the certificate's encoded "not before" and "not after" fields (if any).

10.6.3 X.509 public key certificate objects

X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes defined for this object class:

Table 24, 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[2]	Byte array	BER-encoding of the certificate
CKA_URL[3]	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
CKA_HASH_OF_SUBJECT_PUBLIC_KEY[4]	Byte array	Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY[4]	Byte array	Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_JAVA_MIDP_SECURITY_DOMAIN	CK_ULONG	Java MIDP security domain: 0 = unspecified (default value), 1 = manufacturer, 2 = operator, 3 = third party
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

[1]Must be specified when the object is created.
[2]Must be specified when the object is created. Must be non-empty if CKA_URL is empty.
[3]Must be non-empty if CKA_VALUE is empty.
[4]Can only be empty if CKA_URL is empty.

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 **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA_NAME_HASH_ALGORITHM.

The **CKA_JAVA_MIDP_SECURITY_DOMAIN** attribute associates a certificate with a Java MIDP security domain.

The following is a sample template for creating an X.509 certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_UTF8CHAR label[] = “A certificate object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_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)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_VALUE, certificate, sizeof(certificate)}
```

```
};
```

10.6.4 WTLS public key certificate objects

WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes defined for this object class.

Table 25: WTLS Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[1]	Byte array	WTLS-encoding (Identifier type) of the certificate subject
CKA_ISSUER	Byte array	WTLS-encoding (Identifier type) of the certificate issuer (default empty)
CKA_VALUE[2]	Byte array	WTLS-encoding of the certificate
CKA_URL[3]	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained
CKA_HASH_OF_SUBJECT_PUBLIC_KEY[4]	Byte array	SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY[4]	Byte array	SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

[1]Must be specified when the object is created. Can only be empty if CKA_VALUE is empty.
[2]Must be specified when the object is created. Must be non-empty if CKA_URL is empty.
[3]Must be non-empty if CKA_VALUE is empty.
[4]Can only be empty if CKA_URL is empty.

Only the **CKA_ISSUER** attribute may be modified after the object has been created.

The encoding for the **CKA_SUBJECT**, **CKA_ISSUER**, and **CKA_VALUE** attributes can be found in [WTLS] (see References).

The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA_NAME_HASH_ALGORITHM.

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

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_WTLS;
CK_UTF8CHAR label[] = “A certificate object”;
CK_BYTE subject[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_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)-1},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_VALUE, certificate, sizeof(certificate)}
};
```

10.6.5 X.509 attribute certificate objects

X.509 attribute certificate objects (certificate type **CKC_X_509_ATTR_CERT**) hold X.509 attribute certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the common attributes defined for this object class:

Table 26, X.509 Attribute Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_OWNER[1]	Byte Array	DER-encoding of the attribute certificate's subject field. This is distinct from the CKA_SUBJECT attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different.
CKA_AC_ISSUER	Byte Array	DER-encoding of the attribute certificate's issuer field. This is distinct from the CKA_ISSUER attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different. (default empty)
CKA_SERIAL_NUMBER	Byte Array	DER-encoding of the certificate serial number. (default empty)
CKA_ATTR_TYPES	Byte Array	BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself. (default empty)
CKA_VALUE[1]	Byte Array	BER-encoding of the certificate.

[1]Must be specified when the object is created

Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be modified after the object is created.

The following is a sample template for creating an X.509 attribute certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
CK_UTF8CHAR label[] = "An attribute certificate object";
CK_BYTE owner[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_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)-1},
  {CKA_OWNER, owner, sizeof(owner)},
  {CKA_VALUE, certificate, sizeof(certificate)}
};
```

10.7 Key objects

10.7.1 Definitions

There is no CKO_ definition for the base key object class, only for the key types derived from it.

This section defines the object class CKO_PUBLIC_KEY, CKO_PRIVATE_KEY and CKO_SECRET_KEY for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.7.2 Overview

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:

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes defined for this object class:

Table 27, Common Key Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE[1,5]	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	CK_TRUE if key supports key derivation (*i.e.*, if other keys can be derived from this one (default CK_FALSE)
CKA_LOCAL[2,4,6]	CK_BBOOL	CK_TRUE only if key was either • generated locally (*i.e.*, on the token) with a **C_GenerateKey** or **C_GenerateKeyPair** call • created with a **C_CopyObject** call as a copy of a key which had its **CKA_LOCAL** attribute set to CK_TRUE
CKA_KEY_GEN_ MECHANISM[2,4,6]	CK_MECHANISM _TYPE	Identifier of the mechanism used to generate the key material.

Attribute	Data Type	Meaning
CKA_ALLOWED_MECHANISMS	CK_MECHANISM _TYPE _PTR, pointer to a CK_MECHANISM _TYPE array	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the *ulValueLen* component of the attribute divided by the size of CK_MECHANISM_TYPE.

- Refer to table Table 15 for footnotes

The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys, this field assists in handling 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, if one exists. Cryptoki does not enforce these associations, however. (See Section 10.6 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 CK_TRUE if and only if it is possible to derive other keys from the key.

The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the key was originally generated on the token by a **C_GenerateKey** or **C_GenerateKeyPair** call.

The **CKA_KEY_GEN_MECHANISM** attribute identifies the key generation mechanism used to generate the key material. It contains a valid value only if the **CKA_LOCAL** attribute has the value CK_TRUE. If **CKA_LOCAL** has the value CK_FALSE, the value of the attribute is CK_UNAVAILABLE_INFORMATION.

10.8 Public key objects

Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. The following table defines the attributes common to all public keys, in addition to the common attributes defined for this object class:

Table 28, Common Public Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[8]	Byte array	DER-encoding of the key subject name (default empty)

Attribute	Data type	Meaning
CKA_ENCRYPT[8]	CK_BBOOL	CK_TRUE if key supports encryption[9]
CKA_VERIFY[8]	CK_BBOOL	CK_TRUE if key supports verification where the signature is an appendix to the data[9]
CKA_VERIFY_RECOVER[8]	CK_BBOOL	CK_TRUE if key supports verification where the data is recovered from the signature[9]
CKA_WRAP[8]	CK_BBOOL	CK_TRUE if key supports wrapping (*i.e.*, can be used to wrap other keys)[9]
CKA_TRUSTED[10]	CK_BBOOL	The key can be trusted for the application that it was created. The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of CK_ATTRIBUTE.

- Refer to table Table 15 for footnotes

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.

To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for public keys, use the following table.

Table 29, Mapping of X.509 key usage flags to cryptoki attributes for public keys

Key usage flags for public keys in X.509 public key certificates	Corresponding cryptoki attributes for public keys.
dataEncipherment	CKA_ENCRYPT
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY_RECOVER
keyAgreement	CKA_DERIVE
keyEncipherment	CKA_WRAP
nonRepudiation	CKA_VERIFY
nonRepudiation	CKA_VERIFY_RECOVER

10.9 Private key objects

Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. The following table defines the attributes common to all private keys, in addition to the common attributes defined for this object class:

Table 30, Common Private Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT[8]	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SENSITIVE[8,11]	CK_BBOOL	CK_TRUE if key is sensitive[9]
CKA_DECRYPT[8]	CK_BBOOL	CK_TRUE if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	CK_TRUE if key supports signatures where the signature is an appendix to the data[9]
CKA_SIGN_RECOVER[8]	CK_BBOOL	CK_TRUE if key supports signatures where the data can be recovered from the signature[9]
CKA_UNWRAP[8]	CK_BBOOL	CK_TRUE if key supports unwrapping (*i.e.*, can be used to unwrap other keys)[9]
CKA_EXTRACTABLE[8,12]	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped [9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	CK_TRUE if key has *always* had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	CK_TRUE if key has *never* had the

Attribute	Data type	Meaning
		CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_WRAP_WITH_TRUSTED[11]	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_ALWAYS_AUTHENTICATE	CK_BBOOL	If CK_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK_FALSE.

- Refer to table Table 15 for footnotes

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.

If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE, then certain attributes of the private key cannot be revealed in plaintext outside 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.

The **CKA_ALWAYS_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user to provide a PIN) for each use of a private key. “Use” in this case means a cryptographic operation such as sign or decrypt. This attribute may only be set to CK_TRUE when **CKA_PRIVATE** is also CK_TRUE.

Re-authentication occurs by calling **C_Login** with *userType* set to **CKU_CONTEXT_SPECIFIC** immediately after a cryptographic operation using the key has been initiated (e.g. after **C_SignInit**). In this call, the actual user type is implicitly given by the usage requirements of the active key. If **C_Login** returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g. by **C_Sign**, **C_SignFinal**,..). A return value CKR_PIN_INCORRECT from **C_Login** means that the user was denied permission to use the key and continuing the cryptographic operation will result in a behavior as if **C_Login** had not been called. In both of these cases the session state will remain the same, however repeated failed re-authentication attempts may cause the PIN to be locked. **C_Login** returns in this case CKR_PIN_LOCKED and this also logs the user out from the token. Failing or omitting to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_TRUE will result in CKR_USER_NOT_LOGGED_IN to be returned from calls using the key. **C_Login** will return CKR_OPERATION_NOT_INITIALIZED, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_FALSE.

10.10 Secret key objects

Secret key objects (object class **CKO_SECRET_KEY**) hold secret keys. The following table defines the attributes common to all secret keys, in addition to the common attributes defined for this object class:

Table 31, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE[8,11]	CK_BBOOL	CK_TRUE if object is sensitive (default CK_FALSE)
CKA_ENCRYPT[8]	CK_BBOOL	CK_TRUE if key supports encryption[9]
CKA_DECRYPT[8]	CK_BBOOL	CK_TRUE if key supports decryption[9]
CKA_SIGN[8]	CK_BBOOL	CK_TRUE if key supports signatures (*i.e.*, authentication codes) where the signature is an appendix to the data[9]
CKA_VERIFY[8]	CK_BBOOL	CK_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	CK_TRUE if key supports wrapping (*i.e.*, can be used to wrap other keys)[9]
CKA_UNWRAP[8]	CK_BBOOL	CK_TRUE if key supports unwrapping (*i.e.*, can be used to unwrap other keys)[9]
CKA_EXTRACTABLE[8,12]	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped [9]
CKA_ALWAYS_SENSITIVE[2,4,6]	CK_BBOOL	CK_TRUE if key has *always* had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE[2,4,6]	CK_BBOOL	CK_TRUE if key has *never* had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_CHECK_VALUE	Byte array	Key checksum
CKA_WRAP_WITH_TRUSTED[11]	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_TRUSTED[10]	CK_BBOOL	The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any

Attribute	Data type	Meaning
		keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of CK_ATTRIBUTE
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of CK_ATTRIBUTE.

- Refer to table Table 15 for footnotes

If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE, then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of secret key in the attribute table in the section describing that type of key.

The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE,** of type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to be used to cross-check symmetric keys against other systems where the same key is shared, and as a validity check after manual key entry or restore from backup. Refer to object definitions of specific key types for KCV algorithms.

Properties:

1. For two keys that are cryptographically identical the value of this attribute should be identical.
2. CKA_CHECK_VALUE should not be usable to obtain any part of the key value.
3. Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely (the probability can easily be calculated) but possible.

The attribute is optional but if supported the value of the attribute is always supplied by the library regardless of how the key object is created or derived. It shall be supplied even if the encryption operation for the key is forbidden (i.e. when CKA_ENCRYPT is set to CK_FALSE).

If a value is supplied in the application template (allowed but never necessary) then, if supported, it must match what the library calculates it to be or the library returns a CKR_ATTRIBUTE_VALUE_INVALID. If the library does not support the attribute then it should ignore it. Allowing the attribute in the template this way does no harm and allows the attribute to be treated like any other attribute for the purposes of key wrap and unwrap where the attributes are preserved also.

The generation of the KCV may be prevented by the application supplying the attribute in the template as a no-value (0 length) entry. The application can query the value at any time like any other attribute using C_GetAttributeValue. C_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

Unless otherwise specified for the object definition, the value of this attribute is derived from the key object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

10.11 Domain parameter objects

10.11.1 Definitions

This section defines the object class CKO_DOMAIN_PARAMETERS for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.11.2 Overview

This object class was created to support the storage of certain algorithm's extended parameters. DSA and DH both use domain parameters in the key-pair generation step. In particular, some libraries support the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

To use a domain parameter object you must extract the attributes into a template and supply them (still in the template) to the corresponding key-pair generation function.

Domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**) hold public domain parameters.

The following table defines the attributes common to domain parameter objects in addition to the common attributes defined for this object class:

Table 32, Common Domain Parameter Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE[1]	CK_KEY_TYPE	Type of key the domain parameters can be used to generate.
CKA_LOCAL[2,4]	CK_BBOOL	CK_TRUE only if domain parameters were either • generated locally (*i.e.*, on the token) with a **C_GenerateKey** • created with a **C_CopyObject** call as a copy of domain parameters which had its **CKA_LOCAL** attribute set to CK_TRUE

- Refer to table Table 15 for footnotes

The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the domain parameters were originally generated on the token by a **C_GenerateKey** call.

10.12 Mechanism objects

10.12.1 Definitions

This section defines the object class CKO_MECHANISM for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

10.12.2 Overview

Mechanism objects provide information about mechanisms supported by a device beyond that given by the **CK_MECHANISM_INFO** structure.

When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, mechanism objects are not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_MECHANISM**. This protects applications written to previous versions of cryptoki from finding objects that they do not understand.

Table 33, Common Mechanism Attributes

Attribute	Data Type	Meaning
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object

The **CKA_MECHANISM_TYPE** attribute may not be set.

11 Functions

Cryptoki's functions are organized into the following categories:

- general-purpose functions (4 functions)
- slot and token management functions (9 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 functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes either its entire goal, or nothing at all.

- If a Cryptoki function executes successfully, it returns the value CKR_OK.
- If a Cryptoki function does not execute successfully, it returns some value other than CKR_OK, and the token is in the same state as it was in prior to the function call. If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.

- In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR_GENERAL_ERROR. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function must have a "stub" in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED. The function's entry in the library's **CK_FUNCTION_LIST** structure (as obtained by **C_GetFunctionList**) should point to this stub function (see Section 9.6).

11.1 Function return values

The Cryptoki interface possesses a large number of functions and return values. In Section 11.1, we enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 10.12 details the behavior of Cryptoki functions, including what values each of them may return.

Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions' return values. We have attempted to specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps. For example, it is possible that a particular error code which might apply to a particular Cryptoki function is unfortunately not actually listed in the description of that function as a possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that error code for that function. It would be far preferable for the application to examine the function's return value, see that it indicates some sort of error (even if the application doesn't know precisely *what* kind of error), and behave accordingly.

See Section 11.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.

11.1.1 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.

- 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** structure that can be obtained by calling **C_GetSessionInfo** will hold 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. Depending on what the root cause of the error actually was, it is possible that an attempt to make the exact same function call again would succeed.

- CKR_OK: The function executed successfully. Technically, CKR_OK is not *quite* a "universal" return value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section 11.16) cannot return CKR_OK.

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.

11.1.2 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_WaitForSlotEvent**, **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 invocation, 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*. Note that, as stated in Section 6.7.6, the behavior of Cryptoki is *undefined* if multiple threads of an application attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no guarantee that a function invocation could ever return the value CKR_SESSION_CLOSED—if one thread is using a session when another thread closes that session, that is an instance of multiple threads accessing a common session simultaneously.

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 invocation and a token being removed *during* a function execution.

11.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, or **C_WaitForSlotEvent**) 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. This error code can be returned by more than just the functions mentioned above; in particular, it is possible for **C_GetSlotInfo** to return CKR_DEVICE_ERROR.
- 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 critical (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

11.1.4 Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be returned by an application-supplied callback function. It is:

- CKR_CANCEL: 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 (see Section 11.17). If the callback returns the value CKR_CANCEL, then the function aborts and returns CKR_FUNCTION_CANCELED.

11.1.5 Special return values for mutex-handling functions

There are two other special-purpose return values which are not returned by any actual Cryptoki functions. These values may be returned by application-supplied mutex-handling functions, and they may safely be ignored by application developers who are not using their own threading model. They are:

- CKR_MUTEX_BAD: This error code can be returned by mutex-handling functions who are passed a bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a bad mutex object. There is therefore no guarantee that such a function will successfully detect bad mutex objects and return this value.

- CKR_MUTEX_NOT_LOCKED: This error code can be returned by mutex-unlocking functions. It indicates that the mutex supplied to the mutex-unlocking function was not locked.

11.1.6 All other Cryptoki function return values

Descriptions of the other Cryptoki function return values 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 an execution of a function, then the function may return any applicable error code.

- CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate.

- CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not be set by the application, or which may not be modified by the application. See Section 10.1 for more information.

- 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. See Section 10.1 for more information.

- CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for a particular attribute in a template. See Section 10.1 for more information.

- CKR_BUFFER_TOO_SMALL: The output of the function is too large to fit in the supplied buffer.

- CKR_CANT_LOCK: This value can only be returned by **C_Initialize**. It means that the type of locking requested by the application for thread-safety is not available in this library, and so the application cannot make use of this library in the specified fashion.

- CKR_CRYPTOKI_ALREADY_INITIALIZED: This value can only be returned by **C_Initialize**. It means that the Cryptoki library has already been initialized (by a previous call to **C_Initialize** which did not have a matching **C_Finalize** call).

- CKR_CRYPTOKI_NOT_INITIALIZED: This value can be returned by any function other than **C_Initialize** and **C_GetFunctionList**. It indicates that the function cannot be executed because the Cryptoki library has not yet been initialized by a call to **C_Initialize**.

- CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. 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_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.

- 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_EXCEEDED_MAX_ITERATIONS: An iterative algorithm (for key pair generation, domain parameter generation etc.) failed because we have exceeded the maximum number of iterations. This error code has precedence over CKR_FUNCTION_FAILED. Examples of iterative algorithms include DSA signature generation (retry if either r = 0 or s = 0) and generation of DSA primes p and q specified in FIPS 186-2.

- CKR_FIPS_SELF_TEST_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed. The token entered an error state.


```
Future calls to cryptographic functions on the token will return
CKR_GENERAL_ERROR.  CKR_FIPS_SELF_TEST_FAILED has a higher precedence
over CKR_GENERAL_ERROR. This error may be returned by C_Initialize,
if a power-up self-test  failed, by C_GenerateRandom or C_SeedRandom,
if the continuous random number generator test failed, or by
C_GenerateKeyPair, if the pair-wise consistency test failed.
```

- CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This happens to a cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns CKR_CANCEL (see CKR_CANCEL). It also happens to a function that performs PIN entry through a protected path. The method used to cancel a protected path PIN entry operation is device dependent.

- CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the specified session. This is a legacy error code which is only returned by the legacy functions **C_GetFunctionStatus** and **C_CancelFunction**.

- 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; this stub should simply return the value CKR_FUNCTION_NOT_SUPPORTED.

- CKR_FUNCTION_REJECTED: The signature request is rejected by the user.

- 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_KEY_CHANGED: This value is only returned by **C_SetOperationState**. It indicates that one of the keys specified 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 CK_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: This error code can only be returned by **C_DigestKey**. It indicates that 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: This value is only returned by **C_SetOperationState**. It indicates that the session state cannot be restored because **C_SetOperationState** needs to be supplied with one or more keys that were 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_EXTRACTABLE attribute set to CK_FALSE, Cryptoki (or the token) is unable to wrap the key as requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping key specified is not one of these types). Compare with CKR_KEY_UNEXTRACTABLE.

- CKR_KEY_SIZE_RANGE: Although the requested keyed cryptographic operation could in principle 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_EXTRACTABLE attribute is set to CK_FALSE. Compare with CKR_KEY_NOT_WRAPPABLE.

- CKR_LIBRARY_LOAD_FAILED: The Cryptoki library could not load a dependent shared library.

- CKR_MECHANISM_INVALID: An invalid mechanism was specified to the cryptographic operation. This error code is an appropriate return value if an unknown mechanism was specified or if the mechanism specified cannot be used in the selected token with the selected function.

- CKR_MECHANISM_PARAM_INVALID: Invalid parameters were supplied to the mechanism specified to the cryptographic operation. Which parameter values are supported by a given mechanism can vary from token to token.

- CKR_NEED_TO_CREATE_THREADS: This value can only be returned by **C_Initialize**. It is returned when two conditions hold:

    1. The application called **C_Initialize** in a way which tells the Cryptoki library that application threads executing calls to the library cannot use native operating system methods to spawn new threads.

2. The library cannot function properly without being able to spawn new threads in the above fashion.

- CKR_NO_EVENT: This value can only be returned by **C_GetSlotEvent**. It is returned when **C_GetSlotEvent** is called in non-blocking mode and there are no new slot events to return.

- 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 simultaneous dual cryptographic operations in a session (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_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried out unless C_SetPIN is called to change the PIN value. Whether or not the normal user's PIN on a token ever expires varies from token to token.

- CKR_PIN_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the token. More generally-- when authentication to the token involves something other than a PIN-- 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_PIN_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some particular number of failed authentication attempts has been reached, the token is unwilling to permit further attempts at authentication. Depending on the token, the specified PIN may or may not remain locked indefinitely.

- CKR_PIN_TOO_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is too short, CKR_PIN_LEN_RANGE should be returned instead. This return code only applies to functions which attempt to set a PIN.

- CKR_PUBLIC_KEY_INVALID: The public key fails a public key validation. For example, an EC public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may be returned by C_CreateObject, when the public key is created, or by C_VerifyInit or C_VerifyRecoverInit, when the public key is used. It may also be returned by C_DeriveKey, in preference to CKR_MECHANISM_PARAM_INVALID, if the other party's public key specified in the mechanism's parameters is invalid.

- CKR_RANDOM_NO_RNG: This value can be returned by **C_SeedRandom** and **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator. This return value has higher priority than CKR_RANDOM_SEED_NOT_SUPPORTED.

- CKR_RANDOM_SEED_NOT_SUPPORTED: This value can only be returned by **C_SeedRandom**. It indicates that the token's random number generator does not accept seeding from an application. This return value has lower priority than CKR_RANDOM_NO_RNG.

- CKR_SAVED_STATE_INVALID: This value can only be returned by **C_SetOperationState**. It indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be restored to the specified session.

- CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that 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_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a session with the token is already open, and so the token cannot be initialized.

- CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated] way.

- 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 lower 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_OPERATION_NOT_INITIALIZED.
- CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes. See Section 10.1 for more information.
- CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting attributes. See Section 10.1 for more information.
- 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: This value can only be returned by **C_UnwrapKey**. It indicates that the key handle specified to be used to unwrap another key is not valid.
- CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It indicates that although the requested unwrapping operation could in principle 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: This value can only be returned by **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.
- CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that the specified user cannot be logged into the session,

because it is already logged into the session. For example, if an application has an open SO session, and it attempts to log the SO into it, it will receive this error code.

- CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that the specified user cannot be logged into the session, because another user is already logged into the session. For example, if an application has an open SO session, and it attempts to log the normal user into it, it will receive this error code.

- 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: This value can only be returned by **C_Login**. It indicates that the normal user's PIN has not yet been initialized with **C_InitPIN**.

- CKR_USER_TOO_MANY_TYPES: An attempt was made to have more distinct users simultaneously logged into the token than the token and/or library permits. For example, if some application has an open SO session, and another application attempts to log the normal user into a session, the attempt may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-user tokens.

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

- CKR_WRAPPED_KEY_INVALID: This value can only be returned by **C_UnwrapKey**. It indicates that the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular type of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**), and the wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper type, then **C_UnwrapKey** should return CKR_WRAPPED_KEY_INVALID. This return value has lower priority than CKR_WRAPPED_KEY_LEN_RANGE.

- CKR_WRAPPED_KEY_LEN_RANGE: This value can only be returned by **C_UnwrapKey**. It indicates that 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: This value can only be returned by **C_WrapKey**. It indicates that the key handle specified to be used to wrap another key is not valid.

- CKR_WRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_WrapKey**. It indicates that although the requested wrapping operation could in principle 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: This value can only be returned by **C_WrapKey**. It indicates that the type of the key specified to wrap another key is not consistent with the mechanism specified for wrapping.

11.1.7 More on relative priorities of Cryptoki errors

In general, when a Cryptoki call is made, error codes from Section 11.1.1 (other than CKR_OK) take precedence over error codes from Section 11.1.2, which take precedence over error codes from Section 11.1.3, which take precedence over error codes from Section 11.1.6. One minor implication of this is that functions that use a session handle (*i.e.*, *most* functions!) 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 applies to the result of a Cryptoki call, any of the applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the descriptions of functions.

11.1.8 Error code "gotchas"

Here is a short list of a few particular things about return values that Cryptoki developers might want to be aware of:

1. As mentioned in Sections 11.1.2 and 11.1.3, a Cryptoki library may not be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function invocation.

2. As mentioned in Section 11.1.2, an application should never count on getting a CKR_SESSION_CLOSED error.

3. The difference between CKR_DATA_INVALID and CKR_DATA_LEN_RANGE can be somewhat subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to always treat them equivalently.

4. Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and CKR_ENCRYPTED_DATA_LEN_RANGE, and between

CKR_WRAPPED_KEY_INVALID and CKR_WRAPPED_KEY_LEN_RANGE, can be subtle, and it may be best to treat these return values equivalently.

5. Even with the guidance of Section 10.1, it can be difficult for a Cryptoki library developer to know which of CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, or CKR_TEMPLATE_INCONSISTENT to return. When possible, it is recommended that application developers be generous in their interpretations of these error codes.

11.2 Conventions for functions returning 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 supplied by a subsequent call to **C_DecryptUpdate**, then that call should return 8 bytes of plaintext (one full DES block).

11.3 Disclaimer concerning sample code

For the remainder of this section, 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.

11.4 General-purpose functions

Cryptoki provides the following general-purpose functions:

♦ C_Initialize

```
CK_DEFINE_FUNCTION(CK_RV, C_Initialize)(
  CK_VOID_PTR pInitArgs
);
```

C_Initialize initializes the Cryptoki library. *pInitArgs* either has the value NULL_PTR or points to a **CK_C_INITIALIZE_ARGS** structure containing information on how the library should deal with multi-threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously, it can generally supply the value NULL_PTR to **C_Initialize** (the consequences of supplying this value will be explained below).

If *pInitArgs* is non-NULL_PTR, **C_Initialize** should cast it to a **CK_C_INITIALIZE_ARGS_PTR** and then dereference the resulting pointer to obtain the **CK_C_INITIALIZE_ARGS** fields *CreateMutex*, *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of *pReserved* thereby obtained must be NULL_PTR; if it's not, then **C_Initialize** should return with the value CKR_ARGUMENTS_BAD.

If the **CKF_LIBRARY_CANT_CREATE_OS_THREADS** flag in the *flags* field is set, that indicates that application threads which are executing calls to the Cryptoki library are not permitted to use the native operation system calls to spawn off new threads. In other words, the library's code may not create its own threads. If the library is unable to function properly under this restriction, **C_Initialize** should return with the value CKR_NEED_TO_CREATE_THREADS.

A call to **C_Initialize** specifies one of four different ways to support multi-threaded access via the value of the **CKF_OS_LOCKING_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*, *LockMutex*, and *UnlockMutex* function pointer fields:

1. If the flag *isn't* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value NULL_PTR), that means that the application *won't* be accessing the Cryptoki library from multiple threads simultaneously.

2. If the flag *is* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value NULL_PTR), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use the native operating system primitives to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize** should return with the value CKR_CANT_LOCK.

3. If the flag *isn't* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-NULL_PTR values), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize** should return with the value CKR_CANT_LOCK.

4. If the flag *is* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-NULL_PTR values), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use either the native operating system primitives or the supplied function pointers for mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize** should return with the value CKR_CANT_LOCK.

If some, but not all, of the supplied function pointers to **C_Initialize** are non-NULL_PTR, then **C_Initialize** should return with the value CKR_ARGUMENTS_BAD.

A call to **C_Initialize** with *pInitArgs* set to NULL_PTR is treated like a call to **C_Initialize** with *pInitArgs* pointing to a **CK_C_INITIALIZE_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*, *UnlockMutex*, and *pReserved* fields set to NULL_PTR, and has the *flags* field set to 0.

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; typically, it might cause Cryptoki to initialize its internal memory buffers, or any other resources it requires.

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**. See Section 6.6 for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_CANT_LOCK,
CKR_CRYPTOKI_ALREADY_INITIALIZED, CKR_FUNCTION_FAILED,

CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_NEED_TO_CREATE_THREADS, CKR_OK.

Example: see **C_GetInfo**.

♦ C_Finalize

```
CK_DEFINE_FUNCTION(CK_RV, 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 **C_Finalize** is called with a non-NULL_PTR value for *pReserved*, it should return the value CKR_ARGUMENTS_BAD.

If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application can make calls to other Cryptoki functions. See Section 6.6 for more details.

*Despite the fact that the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded access to a Cryptoki library, the behavior of **C_Finalize** is nevertheless undefined if it is called by an application while other threads of the application are making Cryptoki calls. The exception to this exceptional behavior of **C_Finalize** occurs when a thread calls **C_Finalize** while another of the application's threads is blocking on Cryptoki's **C_WaitForSlotEvent** function. When this happens, the blocked thread becomes unblocked and returns the value CKR_CRYPTOKI_NOT_INITIALIZED. See **C_WaitForSlotEvent** for more information.*

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example: see **C_GetInfo**.

♦ C_GetInfo

```
CK_DEFINE_FUNCTION(CK_RV, C_GetInfo)(
  CK_INFO_PTR pInfo
);
```

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example:

```
CK_INFO info;
CK_RV rv;
CK_C_INITIALIZE_ARGS InitArgs;

InitArgs.CreateMutex = &MyCreateMutex;
InitArgs.DestroyMutex = &MyDestroyMutex;
InitArgs.LockMutex = &MyLockMutex;
InitArgs.UnlockMutex = &MyUnlockMutex;
InitArgs.flags = CKF_OS_LOCKING_OK;
InitArgs.pReserved = NULL_PTR;

rv = C_Initialize((CK_VOID_PTR)&InitArgs);
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_DEFINE_FUNCTION(CK_RV, 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 in turn contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable*. Whether or not this is the 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: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

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);
```

11.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management:

- ♦ **C_GetSlotList**

```
CK_DEFINE_FUNCTION(CK_RV, 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 (CK_TRUE), or all slots (CK_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 (unfortunately) change between when the application asks for how many such slots there are and when the application asks for the slots themselves). However, multiple calls to **C_GetSlotList** are by no means *required*.

All slots which **C_GetSlotList** reports must be able to be queried as valid slots by **C_GetSlotInfo**. Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that **C_GetSlotList,** for list length prediction (NULL pSlotList argument) is called. If an application calls **C_GetSlotList** with a non-NULL pSlotList, and *then* the user adds or removes a hardware device, the changed slot list will only be visible and effective if **C_GetSlotList** is called again with NULL. Even if **C_ GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11 compliant libraries, it may be necessary to successfully call **C_Initialize** or to restart the entire system.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example:

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

/* Get list of all slots */
rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
if (rv == CKR_OK) {
  pSlotList =
    (CK_SLOT_ID_PTR)
        malloc(ulSlotCount*sizeof(CK_SLOT_ID));
  rv = C_GetSlotList(CK_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(
      CK_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_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID.

Example: see **C_GetTokenInfo.**

♦ **C_GetTokenInfo**

```
CK_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

Example:

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

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

```
CK_DEFINE_FUNCTION(CK_RV, C_WaitForSlotEvent)(
  CK_FLAGS flags,
  CK_SLOT_ID_PTR pSlot,
  CK_VOID_PTR pReserved
);
```

C_WaitForSlotEvent waits for a slot event, such as token insertion or token removal, to occur. *flags* determines whether or not the **C_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot* points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved for future versions; for this version of Cryptoki, it should be NULL_PTR.

At present, the only flag defined for use in the *flags* argument is **CKF_DONT_BLOCK**:

Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any unrecognized events involving that slot have occurred. When an application initially calls **C_Initialize**, every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in which the event occurred is set.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and some slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the location pointed to by *pSlot*. If more than one slot's event flag is set at the time of the call, one such slot is chosen by the library to have its event flag cleared and to have its slot ID returned.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and no slot's event flag is set, then the call returns with the value CKR_NO_EVENT. In this case, the contents of the location pointed to by *pSlot* when **C_WaitForSlotEvent** are undefined.

If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag clear in the *flags* argument, then the call behaves as above, except that it will block. That is, if no slot's event flag is set at the time of the call, **C_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has a **C_WaitForSlotEvent** call blocking when another thread of that application calls **C_Finalize**, the **C_WaitForSlotEvent** call returns with the value CKR_CRYPTOKI_NOT_INITIALIZED.

*Although the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded access to a Cryptoki library, **C_WaitForSlotEvent** is exceptional in that the behavior of Cryptoki is undefined if multiple threads of a single application make simultaneous calls to **C_WaitForSlotEvent**.*

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_NO_EVENT, CKR_OK.

Example:

```
CK_FLAGS flags = 0;
CK_SLOT_ID slotID;
CK_SLOT_INFO slotInfo;

.
.
/* Block and wait for a slot event */
rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);
assert(rv == CKR_OK);

/* See what's up with that slot */
rv = C_GetSlotInfo(slotID, &slotInfo);
assert(rv == CKR_OK);
.
.
```

♦ **C_GetMechanismList**

```
CK_DEFINE_FUNCTION(CK_RV, C_GetMechanismList)(
  CK_SLOT_ID slotID,
  CK_MECHANISM_TYPE_PTR pMechanismList,
  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 behavior is by no means required.

Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

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_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

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_DEFINE_FUNCTION(CK_RV, C_InitToken)(
  CK_SLOT_ID slotID,
  CK_UTF8CHAR_PTR pPin,
  CK_ULONG ulPinLen,
  CK_UTF8CHAR_PTR pLabel
);
```

C_InitToken initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (which must be padded with blank characters, and which must *not* be null-terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after the function completes successfully. If the SO PIN is lost, then the card must be reinitialized using a mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token will be reinitialized, and the client must supply the existing SO password in *pPin*.

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 **CKF_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 *any* 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. Unfortunately, it may happen when **C_InitToken** is called 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 the consequences of the **C_InitToken** call are undefined.

The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations, an initialization mechanism outside the scope of Cryptoki must be employed. The definition of "complex token" is product specific.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION_EXISTS, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

Example:

```
CK_SLOT_ID slotID;
CK_UTF8CHAR_PTR pin = “MyPIN”;
CK_UTF8CHAR label[32];
CK_RV rv;

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

♦ **C_InitPIN**

```
CK_DEFINE_FUNCTION(CK_RV, C_InitPIN)(
  CK_SESSION_HANDLE hSession,
  CK_UTF8CHAR_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. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

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 CKF_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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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, CKR_ARGUMENTS_BAD.

Example:

```
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR newPin[]= {"NewPIN"};
CK_RV rv;

rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);
if (rv == CKR_OK) {
  .
  .
}
```

♦ C_SetPIN

```
CK_DEFINE_FUNCTION(CK_RV, C_SetPIN)(
  CK_SESSION_HANDLE hSession,
  CK_UTF8CHAR_PTR pOldPin,
  CK_ULONG ulOldLen,
  CK_UTF8CHAR_PTR pNewPin,
  CK_ULONG ulNewLen
);
```

C_SetPIN modifies the PIN of the user that is currently logged in, or the CKU_USER PIN if the session is not 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. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

C_SetPIN can only be called in the "R/W Public Session" state, "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 CKF_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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

Example:

```
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR oldPin[] = {“OldPIN”};
CK_UTF8CHAR newPin[] = {“NewPIN”};
CK_RV rv;

rv = C_SetPIN(
  hSession, oldPin, sizeof(oldPin)-1, newPin,
        sizeof(newPin)-1);
if (rv == CKR_OK) {
  .
  .
}
```

11.6 Session management functions

A typical application might perform the following series of steps to make use of a token (note that there are other reasonable sequences of events that an application might perform):

1. Select a token.

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

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, or call **C_CloseAllSessions** to close all the application's sessions simultaneously.

As has been observed, 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.

Cryptoki provides the following functions for session management:

♦ **C_OpenSession**

```
CK_DEFINE_FUNCTION(CK_RV, C_OpenSession)(
  CK_SLOT_ID slotID,
  CK_FLAGS flags,
  CK_VOID_PTR pApplication,
  CK_NOTIFY Notify,
  CK_SESSION_HANDLE_PTR phSession
);
```

C_OpenSession opens 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 11.17); *phSession* points to the location that receives the handle for the new session.

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 legacy reasons, the **CKF_SERIAL_SESSION** bit must always be set; if a call to **C_OpenSession** does not have this bit set, the call should return unsuccessfully with the error code CKR_SESSION_PARALLEL_NOT_SUPPORTED.

There may be a limit on the number of concurrent sessions an application may have 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 6.7.7).

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 11.17 for more information about application callbacks.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_COUNT, 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, CKR_ARGUMENTS_BAD.

Example: see **C_CloseSession**.

- **C_CloseSession**

```
CK_DEFINE_FUNCTION(CK_RV, 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 6.7.5-6.7.7 for more details).

If this function is successful and it closes the last session between the application and the token, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

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 actually 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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_SERIAL_SESSION | CKF_RW_SESSION,
        (CK_VOID_PTR) &application, MyNotify,
  &hSession);
if (rv == CKR_OK) {
  .
  .
  C_CloseSession(hSession);
}
```

- **C_CloseAllSessions**

```
CK_DEFINE_FUNCTION(CK_RV, C_CloseAllSessions)(
  CK_SLOT_ID slotID
);
```

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

When a session is closed, all session objects created by the session are destroyed automatically.

After successful execution of this function, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

Example:

```
CK_SLOT_ID slotID;
CK_RV rv;

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

- **C_GetSessionInfo**

```
CK_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

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

C_GetOperationState obtains a copy of 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 11.2 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_DES_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 11.13), then the cryptographic operations state of the session will contain all the necessary information to restore both operations.

An attempt to save the cryptographic operations state of a session which does not currently have some active savable 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_STATE_UNSAVEABLE, CKR_ARGUMENTS_BAD.

Example: see **C_SetOperationState**.

- **C_SetOperationState**

```
CK_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED, CKR_OK, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

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_DEFINE_FUNCTION(CK_RV, C_Login)(
  CK_SESSION_HANDLE hSession,
  CK_USER_TYPE userType,
  CK_UTF8CHAR_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. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

When the user type is either CKU_SO or CKU_USER, 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 user type is CKU_CONTEXT_SPECIFIC , the behavior of C_Login depends on the context in which it is called. Improper use of this user type will result in a return value CKR_OPERATION_NOT_INITIALIZED..

If the token has a "protected authentication path", as indicated by the **CKF_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 an application's session, and then **C_Login** is successfully executed by that application, 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 6.7.7). An attempt to do this will result in the error code CKR_SESSION_READ_ONLY_EXISTS.

C_Login may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do cryptographic operation on this key. See further Section 10.9.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

Example: see **C_Logout**.

- **C_Logout**

```
CK_DEFINE_FUNCTION(CK_RV, 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 from sessions belonging to the application are destroyed.

If there are any active cryptographic or object-finding operations in an application's session, and then **C_Logout** is successfully executed by that application, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example:

```
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR userPIN[] = {“MyPIN”};
CK_RV rv;

rv = C_Login(hSession, CKU_USER, userPIN,
        sizeof(userPIN)-1);
if (rv == CKR_OK) {
  .
  .
  rv == C_Logout(hSession);
  if (rv == CKR_OK) {
    .
    .
  }
}
```

11.7 Object management functions

Cryptoki provides the following functions for managing objects. Additional functions provided specifically for managing key objects are described in Section 11.14.

♦ **C_CreateObject**

```
CK_DEFINE_FUNCTION(CK_RV, 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 a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without creating any object.

If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to CK_FALSE. If that key object is a secret or private key then the new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE.

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_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_EXPIRED, 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_UTF8CHAR application[] = {“My Application”};
CK_BYTE dataValue[] = {...};
CK_BYTE subject[] = {...};
CK_BYTE id[] = {...};
CK_BYTE certificateValue[] = {...};
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE dataTemplate[] = {
  {CKA_CLASS, &dataClass, sizeof(dataClass)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_APPLICATION, application, sizeof(application)-1},
  {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 public key object */
  rv = C_CreateObject(hSession, &keyTemplate, 5, &hKey);
  if (rv == CKR_OK) {
    .
    .
  }
```

♦ **C_CopyObject**

```
CK_DEFINE_FUNCTION(CK_RV, 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 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may specify that the new key's **CKA_SENSITIVE** attribute be CK_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.

If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without creating any object. If the object indicated by hObject has its CKA_COPYABLE attribute set to CK_FALSE, C_CopyObject will return CKR_COPY_PROHIBITED.

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_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN, CKR_COPY_PROHIBITED.

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_BBOOL false = CK_FALSE;
CK_BBOOL true = CK_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_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

Example: see **C_GetObjectSize**.

♦ **C_GetObjectSize**

```
CK_DEFINE_FUNCTION(CK_RV, 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 precise 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_OBJECT_CLASS dataClass = CKO_DATA;
CK_UTF8CHAR application[] = {“My Application”};
CK_BYTE dataValue[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
  {CKA_CLASS, &dataClass, sizeof(dataClass)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_APPLICATION, application, sizeof(application)-1},
  {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_DEFINE_FUNCTION(CK_RV, 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 unextractable, 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.

In the special case of an attribute whose value is an array of attributes, for example **CKA_WRAP_TEMPLATE**, where it is passed in with *pValue* not NULL, then if the *pValue* of elements within the array is NULL_PTR then the *ulValueLen* of elements within the array will be set to the required length. If the *pValue* of elements within the array is not NULL_PTR, then the *ulValueLen* element of attributes within the array must reflect the space that the corresponding *pValue* points to, and *pValue* is filled in if there is sufficient room. Therefore it is important to initialize the contents of a buffer before calling **C_GetAttributeValue** to get such an array value. If any *ulValueLen* within the array isn't large enough, it will be set to –1 and the function will return CKR_BUFFER_TOO_SMALL, as it does if an attribute in the *pTemplate* argument has *ulValueLen* too small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute type having the CKF_ARRAY_ATTRIBUTE bit set.

Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and CKR_BUFFER_TOO_SMALL do not denote true errors for **C_GetAttributeValue**. If a call to **C_GetAttributeValue** returns any of these three values, then the call must nonetheless have processed *every* attribute in the template supplied to **C_GetAttributeValue**. Each attribute in the template whose value *can be* returned by the call to **C_GetAttributeValue** *will be* returned by the call to **C_GetAttributeValue**.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, 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_DEFINE_FUNCTION(CK_RV, 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.

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

The template may specify 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 10.1.2 for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,

CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, 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 hObject;
CK_UTF8CHAR label[] = {“New label”};
CK_ATTRIBUTE template[] = {
  CKA_LABEL, label, sizeof(label)-1
};
CK_RV rv;

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

♦ **C_FindObjectsInit**

```
CK_DEFINE_FUNCTION(CK_RV, 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 initialize a search operation which will match no objects and return CKR_OK.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_FindObjectsFinal**.

♦ C_FindObjects

```
CK_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_FindObjectsFinal**.

♦ C_FindObjectsFinal

```
CK_DEFINE_FUNCTION(CK_RV, 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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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);
```

11.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

♦ **C_EncryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, 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 CK_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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

♦ **C_Encrypt**

```
CK_DEFINE_FUNCTION(CK_RV, C_Encrypt)(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pData,
  CK_ULONG ulDataLen,
  CK_BYTE_PTR pEncryptedData,
  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 11.2 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.

C_Encrypt can not be used to terminate a multi-part operation, and must be called after **C_EncryptInit** without intervening **C_EncryptUpdate** calls.

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.

For most mechanisms, **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** operations followed by **C_EncryptFinal**.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

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

♦ C_EncryptUpdate

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_EncryptFinal.**

♦ C_EncryptFinal

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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+ulEncryptedDat
          a2Len],
      &ulEncryptedData3Len);
    if (rv != CKR_OK) {
       .
       .
    }
  }
```


11.9 Decryption functions

Cryptoki provides the following functions for decrypting data:

♦ **C_DecryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, 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 CK_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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_DecryptFinal**.

♦ **C_Decrypt**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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.

C_Decrypt can not be used to terminate a multi-part operation, and must be called after **C_DecryptInit** without intervening **C_DecryptUpdate** calls.

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.

For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed by **C_DecryptFinal**.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

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

♦ **C_DecryptUpdate**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: See **C_DecryptFinal**.

♦ **C_DecryptFinal**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

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) {
       .
       .
    }
  }
```

11.10 Message digesting functions

Cryptoki provides the following functions for digesting data:

♦ C_DigestInit

```
CK_DEFINE_FUNCTION(CK_RV, 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 message digest. To process additional data (in single or multiple parts), the application must call **C_DigestInit** again.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_DigestFinal**.

- **C_Digest**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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.

C_Digest can not be used to terminate a multi-part operation, and must be called after **C_DigestInit** without intervening **C_DigestUpdate** calls.

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** operations followed by **C_DigestFinal**.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

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

♦ C_DigestUpdate

```
CK_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DigestFinal**.

♦ C_DigestKey

```
CK_DEFINE_FUNCTION(CK_RV, C_DigestKey)(
  CK_SESSION_HANDLE hSession,
  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_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_DigestFinal**.

- **C_DigestFinal**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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);
.
.
```

11.11 Signing and MACing functions

Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations also encompass message authentication codes):

♦ C_SignInit

```
CK_DEFINE_FUNCTION(CK_RV, 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 CK_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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_SignFinal**.

- ♦ **C_Sign**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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 can not be used to terminate a multi-part operation, and must be called after **C_SignInit** without intervening **C_SignUpdate** calls.

For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by **C_SignFinal**.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED.

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

- **C_SignUpdate**

```
CK_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_SignFinal**.

- **C_SignFinal**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED.

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_DEFINE_FUNCTION(CK_RV, 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 CK_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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_SignRecover**.

- ♦ **C_SignRecover**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

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) {
  ulSignatureLen = sizeof(signature);
  rv = C_SignRecover(
    hSession, data, sizeof(data), signature,
        &ulSignatureLen);
  if (rv == CKR_OK) {
     .
     .
  }
}
```


11.12 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):

♦ **C_VerifyInit**

```
CK_DEFINE_FUNCTION(CK_RV, 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 CK_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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_VerifyFinal**.

♦ C_Verify

```
CK_DEFINE_FUNCTION(CK_RV, 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 can not be used to terminate a multi-part operation, and must be called after **C_VerifyInit** without intervening **C_VerifyUpdate** calls.

For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by **C_VerifyFinal**.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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_DEFINE_FUNCTION(CK_RV, C_VerifyUpdate)(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pPart,
  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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see **C_VerifyFinal**.

♦ **C_VerifyFinal**

```
CK_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,

CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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_DEFINE_FUNCTION(CK_RV, 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 CK_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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_VerifyRecover**.

- **C_VerifyRecover**

```
CK_DEFINE_FUNCTION(CK_RV, 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 11.2 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,

CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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);
  .
  .
}
```

11.13 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.

♦ **C_DigestEncryptUpdate**

```
CK_DEFINE_FUNCTION(CK_RV, 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; *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

C_DigestEncryptUpdate uses the convention described in Section 11.2 on producing output. If a **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because *pEncryptedPart* has the value NULL_PTR, or because *pulEncryptedPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active digest operation.

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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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_DEFINE_FUNCTION(CK_RV, 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; *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_DecryptDigestUpdate uses the convention described in Section 11.2 on producing output. If a **C_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because *pPart* has the value NULL_PTR, or because *pulPartLen* is too small to hold the entire decrypted part output), then no plaintext is passed to the active digest operation.

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 via a **C_DigestUpdate** call.

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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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_DEFINE_FUNCTION(CK_RV, 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; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

C_SignEncryptUpdate uses the convention described in Section 11.2 on producing output. If a **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because *pEncryptedPart* has the value NULL_PTR, or because *pulEncryptedPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active signing operation.

Signature and encryption operations must both be active (they must have been initialized with **C_SignInit** 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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

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_SignFinal(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_DEFINE_FUNCTION(CK_RV, 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; *pEncryptedPart* points to the encrypted data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

C_DecryptVerifyUpdate uses the convention described in Section 11.2 on producing output. If a **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because *pPart* has the value NULL_PTR, or because *pulPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active verification operation.

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 via a **C_VerifyUpdate** call.

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_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, 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_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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) {
  .
  .
}
```

11.14 Key management functions

Cryptoki provides the following functions for key management:

♦ **C_GenerateKey**

```
CK_DEFINE_FUNCTION(CK_RV, 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 or set of domain parameters, creating a new object. *hSession* is the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey* points to the location that receives the handle of the new key or set of domain parameters.

If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the value CKO_DOMAIN_PARAMETERS; otherwise, it will have the value CKO_SECRET_KEY.

Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the template does not need to supply a key type. If it does supply a key type which is inconsistent with the generation mechanism, **C_GenerateKey** fails and returns the error code CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without creating an object.

The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to CK_TRUE.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, 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_DEFINE_FUNCTION(CK_RV, 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 CKA_CLASS attribute is treated similarly.

If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return without creating any key objects.

A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no keys; or it can succeed, and create a matching public/private key pair.

The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL** attributes set to CK_TRUE.

*Note carefully the order of the arguments to **C_GenerateKeyPair**. The last two arguments do not have the same order as they did in the original Cryptoki Version 1.0 document. The order of these two arguments has caused some unfortunate confusion.*

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, 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 = CK_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_DEFINE_FUNCTION(CK_RV, 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 11.2 on producing output.

The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping, must be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped must also be CK_TRUE.

If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its **CKA_EXTRACTABLE** attribute set to CK_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 be used in the following situations:

- To wrap any secret key with a public key that supports encryption and decryption.
- To wrap any secret key with any other secret key. Consideration must be given to key size and mechanism strength or the token may not allow the operation.
- To wrap a private key with any secret key.

Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute CKA_WRAP_TEMPLATE can be used on the wrapping key to specify an attribute set that will be compared against the attributes of the key to be wrapped. If all attributes match according to the C_FindObject rules of attribute matching then the wrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this attribute is not supplied then any template is acceptable. Attributes not present are not checked. If any attribute mismatch occurs on an attempt to wrap a key then the function shall return CKR_KEY_HANDLE_INVALID.

Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, 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_DEFINE_FUNCTION(CK_RV, 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 CK_TRUE.

The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by default set to CK_TRUE.

Some mechanisms may modify, or attempt to modify. the contents of the pMechanism structure at the same time that the key is unwrapped.

If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without creating any key object.

The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to CK_FALSE.

To partition the unwrapping keys so they can only unwrap a subset of keys the attribute CKA_UNWRAP_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute conflict occurs on an attempt to unwrap a key then the function shall return CKR_TEMPLATE_INCONSISTENT.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, 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 = CK_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_DEFINE_FUNCTION(CK_RV, 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 11.17.2 for any constraints of this type.

If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without creating any key object.

The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to CK_FALSE.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,

CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, 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 = CK_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) {
      .
      .
    }
  }
}
```

11.15 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

♦ **C_SeedRandom**

```
CK_DEFINE_FUNCTION(CK_RV, 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_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_GenerateRandom**.

♦ **C_GenerateRandom**

```
CK_DEFINE_FUNCTION(CK_RV, C_GenerateRandom)(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pRandomData,
  CK_ULONG ulRandomLen
);
```

C_GenerateRandom generates random or pseudo-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 or pseudo-random data to be generated.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, 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) {
  .
  .
}
```

11.16 Parallel function management functions

Cryptoki provides the following functions for managing parallel execution of cryptographic functions. These functions exist only for backwards compatibility.

♦ **C_GetFunctionStatus**

```
CK_DEFINE_FUNCTION(CK_RV, C_GetFunctionStatus)(
  CK_SESSION_HANDLE hSession
);
```

In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should simply return the value CKR_FUNCTION_NOT_PARALLEL.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

♦ **C_CancelFunction**

```
CK_DEFINE_FUNCTION(CK_RV, C_CancelFunction)(
  CK_SESSION_HANDLE hSession
);
```

In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value CKR_FUNCTION_NOT_PARALLEL.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

11.17 Callback functions

Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain events.

11.17.1 Surrender callbacks

Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session's callback with the arguments `(hSession, CKN_SURRENDER,` `pApplication)`, where `hSession` is the session's handle and `pApplication` was supplied to **C_OpenSession** when the session was opened. 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, if desired.

A typical use of a surrender callback might be to give an application user feedback during a lengthy key pair generation operation. Each time the application receives a callback, it could display an additional "." to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the key pair generation operation (probably by returning the value CKR_CANCEL) if the user hit <ESCAPE>.

A Cryptoki library is not *required* to make *any* surrender callbacks.

11.17.2 Vendor-defined callbacks

Library vendors can also define additional types of callbacks. Because of this extension capability, application-supplied notification callback routines should examine each callback they receive, and if they are unfamiliar with the type of that callback, they should immediately give control back to the library by returning with the value CKR_OK.

12 Cryptoki tips and reminders

In this section, we clarify, review, and/or emphasize a few odds and ends about how Cryptoki works.

12.1 Operations, sessions, and threads

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) in a session, 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 a session has an active operation and the application attempts to use that session 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.

To abandon an active operation an application may have to complete the operation and discard the result. Closing the session will also have this effect. Alternatively. the library may allow active operations to be abandoned by the application, simply by allowing initialization for some other operation. In this case CKR_OPERATION_ACTIVE will not be returned but the previous active operation will be unusable.

Different threads of an application should never share sessions, unless they are extremely careful not to make function calls at the same time. This is true even if the Cryptoki library was initialized with locking enabled for thread-safety.

12.2 Multiple Application Access Behavior

When multiple applications, or multiple threads within an application, are accessing a set of common objects the issue of object protection becomes important. This is especially the case when application A activates an operation using object O, and application B attempts to delete O before application A has finished the operation. Unfortunately, variation in device capabilities makes an absolute behavior specification impractical. General guidelines are presented here for object protection behavior.

Whenever possible, deleting an object in one application should not cause that object to become unavailable to another application or thread that is using the object in an active operation until that operation is complete. For instance, application A has begun a signature operation with private key P and application B attempts to delete P while the signature is in progress. In this case, one of two things should happen. The object is deleted from the device but the operation is allow to complete because the operation uses a temporary copy of the object, or the delete operation blocks until the signature operation has completed. If neither of these actions can be supported by an implementation, then the error code CKR_OBJECT_HANDLE_INVALID may be returned to application A to indicate that the key being used to perform its active operation has been deleted.

Whenever possible, changing the value of an object attribute should impact the behavior of active operations in other applications or threads. If this can not be supported by an implementation, then the appropriate error code indicating the reason for the failure should be returned to the application with the active operation.

12.3 Objects, attributes, and templates

In general, a Cryptoki function which requires a template for an object needs the template to specify—either explicitly or implicitly—any attributes that are not specified elsewhere. If a template specifies a particular attribute 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 always single-valued.

12.4 Signing with recovery

Signing with recovery is a general alternative to ordinary digital signatures ("signing with appendix") 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 input to the verification process, and the verification process outputs either "signature invalid" or—if the signature is valid—the original message.

Consider a simple example with the **CKM_RSA_X_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 valid if and only if the result matches the original message.

If **CKM_RSA_X_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*.

A Manifest constants

The following definitions can be found in the appropriate header file.

Also, refer [PKCS #11 M1] and [PKCS #11 M2] for additional definitions.

```
#define CK_INVALID_HANDLE      0

#define CKN_SURRENDER 0

#define CK_UNAVAILABLE_INFORMATION     (~0UL)
#define CK_EFFECTIVELY_INFINITE        0

#define CKF_DONT_BLOCK 1

#define CKF_ARRAY_ATTRIBUTE    0x40000000

#define CKU_SO                 0
#define CKU_USER               1
#define CKU_CONTEXT_SPECIFIC   2

#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

#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_HW_FEATURE         0x00000005
#define CKO_DOMAIN_PARAMETERS  0x00000006
#define CKO_MECHANISM          0x00000007
#define CKO_VENDOR_DEFINED     0x80000000

#define CKH_MONOTONIC_COUNTER  0x00000001
#define CKH_CLOCK              0x00000002
#define CKH_USER_INTERFACE     0x00000003
#define CKH_VENDOR_DEFINED     0x80000000

#define CKK_VENDOR_DEFINED  0x80000000

#define CKC_VENDOR_DEFINED  0x80000000

#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_OBJECT_ID                       0x00000012
#define CKA_CERTIFICATE_TYPE                0x00000080
#define CKA_ISSUER                          0x00000081
#define CKA_SERIAL_NUMBER                   0x00000082
```

```
#define CKA_AC_ISSUER                   0x00000083
#define CKA_OWNER                       0x00000084
#define CKA_ATTR_TYPES                  0x00000085
#define CKA_TRUSTED                     0x00000086
#define CKA_CERTIFICATE_CATEGORY        0x00000087
#define CKA_JAVA_MIDP_SECURITY_DOMAIN   0x00000088
#define CKA_URL                         0x00000089
#define CKA_HASH_OF_SUBJECT_PUBLIC_KEY  0x0000008A
#define CKA_HASH_OF_ISSUER_PUBLIC_KEY   0x0000008B
#define CKA_NAME_HASH_ALGORITHM         0x0000008C
#define CKA_CHECK_VALUE                 0x00000090
#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_PRIME_BITS                  0x00000133
#define CKA_SUBPRIME_BITS               0x00000134
#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_KEY_GEN_MECHANISM           0x00000166
#define CKA_MODIFIABLE                  0x00000170
#define CKA_COPYABLE                    0x00000171
#define CKA_ECDSA_PARAMS                0x00000180
#define CKA_EC_PARAMS                   0x00000180
#define CKA_EC_POINT                    0x00000181
#define CKA_SECONDARY_AUTH              0x00000200 /* Deprecated */
#define CKA_AUTH_PIN_FLAGS              0x00000201 /* Deprecated */
#define CKA_ALWAYS_AUTHENTICATE         0x00000202

#define CKA_WRAP_WITH_TRUSTED           0x00000210
#define CKA_WRAP_TEMPLATE      (CKF_ARRAY_ATTRIBUTE|0x00000211)
#define CKA_UNWRAP_TEMPLATE    (CKF_ARRAY_ATTRIBUTE|0x00000212)
#define CKA_HW_FEATURE_TYPE             0x00000300
#define CKA_RESET_ON_INIT               0x00000301
```

```
#define CKA_HAS_RESET                   0x00000302
#define CKA_PIXEL_X                     0x00000400
#define CKA_PIXEL_Y                     0x00000401
#define CKA_RESOLUTION                  0x00000402
#define CKA_CHAR_ROWS                   0x00000403
#define CKA_CHAR_COLUMNS                0x00000404
#define CKA_COLOR                       0x00000405
#define CKA_BITS_PER_PIXEL              0x00000406
#define CKA_CHAR_SETS                   0x00000480
#define CKA_ENCODING_METHODS            0x00000481
#define CKA_MIME_TYPES                  0x00000482
#define CKA_MECHANISM_TYPE              0x00000500
#define CKA_REQUIRED_CMS_ATTRIBUTES     0x00000501
#define CKA_DEFAULT_CMS_ATTRIBUTES      0x00000502
#define CKA_SUPPORTED_CMS_ATTRIBUTES    0x00000503
#define CKA_ALLOWED_MECHANISMS (CKF_ARRAY_ATTRIBUTE|0x00000600)
#define CKA_VENDOR_DEFINED              0x80000000

#define CKM_VENDOR_DEFINED                  0x80000000

#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_ARGUMENTS_BAD                     0x00000007
#define CKR_NO_EVENT                          0x00000008
#define CKR_NEED_TO_CREATE_THREADS            0x00000009
#define CKR_CANT_LOCK                         0x0000000A
#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_COPY_PROHIBITED                   0x0000001A
#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_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_PIN_EXPIRED                         0x000000A3
#define CKR_PIN_LOCKED                          0x000000A4
#define CKR_SESSION_CLOSED                      0x000000B0
#define CKR_SESSION_COUNT                       0x000000B1
#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_USER_ANOTHER_ALREADY_LOGGED_IN      0x00000104
#define CKR_USER_TOO_MANY_TYPES                 0x00000105
#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_DOMAIN_PARAMS_INVALID               0x00000130
#define CKR_BUFFER_TOO_SMALL                    0x00000150
#define CKR_SAVED_STATE_INVALID                 0x00000160
#define CKR_INFORMATION_SENSITIVE               0x00000170
#define CKR_STATE_UNSAVEABLE                    0x00000180
#define CKR_CRYPTOKI_NOT_INITIALIZED            0x00000190
#define CKR_CRYPTOKI_ALREADY_INITIALIZED        0x00000191
#define CKR_MUTEX_BAD                           0x000001A0
#define CKR_MUTEX_NOT_LOCKED                    0x000001A1
#define CKR_FUNCTION_REJECTED                   0x00000200
#define CKR_VENDOR_DEFINED                      0x80000000
```

B 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 two common types of applications:

Table B-1, Mechanisms and profiles

Mechanism	Application	
	Government Authentication-only	Cellular Digital Packet Data
CKM_DSA_KEY_PAIR_GEN	✓	
CKM_DSA	✓	
CKM_DH_PKCS_KEY_PAIR_GEN		✓
CKM_DH_PKCS_DERIVE		✓
CKM_RC4_KEY_GEN		✓
CKM_RC4		✓
CKM_SHA_1	✓	

B.1 Government authentication-only

The U.S. government has standardized on the Digital Signature Algorithm as defined in FIPS PUB 186-2 for signatures and the Secure Hash Algorithm as defined in FIPS PUB 180-2 for message digesting. The relevant mechanisms include the following:

DSA key generation (512-1024 bits)
DSA (512-1024 bits)
SHA-1

B.2 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.)

B.3 Other profiles

The reader is also informed of the presence of other profiles of PKCS #11 v2. – See [PKCS #11-C] and [PKCS #11-P]

C 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 Programming Interface, April 29, 1994
- X/Open GCS-API - Generic Cryptographic Service API, Draft 2, February 14, 1995

C.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 C-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

FORTEZZA CIPG	**Equivalent Cryptoki**
CI_GetStatus	C_GetTokenInfo
CI_GetTime	C_GetTokenInfo or C_GetAttributeValue(clock object) ***[preferred]***
CI_Hash	C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal
CI_Initialize	C_Initialize
CI_InitializeHash	C_DigestInit
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

C.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 C-2, GCS-API vs. Cryptoki

GCS-API	Cryptoki implementation
retrieve_CC	
release_CC	
generate_hash	C_DigestInit, C_Digest
generate_random_number	C_GenerateRandom
generate_checkvalue	C_SignInit, C_Sign, C_SignUpdate, C_SignFinal
verify_checkvalue	C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal
data_encipher	C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal
data_decipher	C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal
create_CC	
derive_key	C_DeriveKey
generate_key	C_GenerateKey
store_CC	
delete_CC	
replicate_CC	
export_key	C_WrapKey
import_key	C_UnwrapKey
archive_CC	C_WrapKey
restore_CC	C_UnwrapKey
set_key_state	
generate_key_pattern	
verify_key_pattern	
derive_clear_key	C_DeriveKey
generate_clear_key	C_GenerateKey
load_key_parts	
clear_key_encipher	C_WrapKey

GCS-API	Cryptoki implementation
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	

D Intellectual property considerations

The RSA public-key cryptosystem is described in U.S. Patent 4,405,829, which expired on September 20, 2000. The RC5 block cipher is protected by U.S. Patents 5,724,428 and 5,835,600. RSA Security Inc. makes no other patent claims on the constructions described in this document, although specific underlying techniques may be covered.

RSA, RC2 and RC4 are registered trademarks of RSA Security Inc. RC5 is a trademark of RSA Security Inc.

CAST, CAST3, CAST5, and CAST128 are registered trademarks of Entrust Technologies. OS/2 and CDMF (Commercial Data Masking Facility) are registered trademarks of International Business Machines Corporation. LYNKS is a registered trademark of SPYRUS Corporation. IDEA is a registered trademark of Ascom Systec. Windows, Windows 3.1, Windows 95, Windows NT, and Developer Studio are registered trademarks of Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories. FORTEZZA is a registered trademark of the National Security Agency.

License to copy this document is granted provided that it is identified as “RSA Security Inc. Public-Key Cryptography Standards (PKCS)” in all material mentioning or referencing this document.

RSA Security Inc. makes no other representations regarding intellectual property claims by other parties. Such determination is the responsibility of the user.

E Method for Exposing Multiple-PINs on a Token Through Cryptoki (deprecated)

Note: This support may be present for backwards compatibility. Refer to PKCS11 V 2.11 for details.

F Revision History

This is the initial version of PKCS #11 Base Functionality v2.30.

