



PKCS #11 Cryptographic Token Interface Base Specification Version 3.0

Committee Specification 01

19 December 2019

This stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/cs01/pkcs11-base-v3.0-cs01.docx> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/cs01/pkcs11-base-v3.0-cs01.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/cs01/pkcs11-base-v3.0-cs01.pdf>

Previous stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.docx>
(Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.pdf>

Latest stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.docx> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.pdf>

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Tony Cox (tony.cox@cryptsoft.com), Cryptsoft Pty Ltd
Robert Relyea (rrelyea@redhat.com), Red Hat

Editors:

Chris Zimman (chris@wmpp.com), Individual
Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- PKCS #11 header files:
<https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/cs01/include/pkcs11-v3.0/>

Related work:

This specification replaces or supersedes:

- *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. Edited by Robert Griffin and Tim Hudson. Latest stage. <http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html>.

This specification is related to:

- *PKCS #11 Cryptographic Token Interface Profiles Version 3.0*. Edited by Tim Hudson. Latest stage. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.
- *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. Latest stage. <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>.

- PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0. Edited by Chris Zimman and Dieter Bong. Latest stage. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.

Abstract:

This document defines data types, functions and other basic components of the PKCS #11 Cryptoki interface.

Status:

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11#technical.

TC members should send comments on this document to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "[Send A Comment](#)" button on the TC's web page at <https://www.oasis-open.org/committees/pkcs11/>.

This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

Note that any machine-readable content ([Computer Language Definitions](#)) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[PKCS11-Base-v3.0]

PKCS #11 Cryptographic Token Interface Base Specification Version 3.0. Edited by Chris Zimman and Dieter Bong. 19 December 2019. OASIS Committee Specification 01. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/cs01/pkcs11-base-v3.0-cs01.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.

Notices

Copyright © OASIS Open 2019. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](#) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](#), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark> for above guidance.

Table of Contents

1	Introduction	9
1.1	IPR Policy	9
1.2	Terminology	9
1.3	Definitions	9
1.4	Symbols and abbreviations.....	10
1.5	Normative References	13
1.6	Non-Normative References	14
2	Platform- and compiler-dependent directives for C or C++	16
2.1	Structure packing	16
2.2	Pointer-related macros	16
3	General data types	18
3.1	General information	18
3.2	Slot and token types	19
3.3	Session types	24
3.4	Object types	26
3.5	Data types for mechanisms	30
3.6	Function types	32
3.7	Locking-related types.....	37
4	Objects	41
4.1	Creating, modifying, and copying objects	42
4.1.1	Creating objects	42
4.1.2	Modifying objects.....	43
4.1.3	Copying objects	43
4.2	Common attributes	44
4.3	Hardware Feature Objects.....	44
4.3.1	Definitions.....	44
4.3.2	Overview.....	44
4.3.3	Clock.....	45
4.3.3.1	Definition	45
4.3.3.2	Description.....	45
4.3.4	Monotonic Counter Objects.....	45
4.3.4.1	Definition	45
4.3.4.2	Description.....	45
4.3.5	User Interface Objects.....	45
4.3.5.1	Definition	45
4.3.5.2	Description.....	46
4.4	Storage Objects	46
4.4.1	The CKA_UNIQUE_ID attribute	47
4.5	Data objects	48
4.5.1	Definitions.....	48
4.5.2	Overview.....	48
4.6	Certificate objects	48
4.6.1	Definitions.....	48
4.6.2	Overview.....	48

4.6.3 X.509 public key certificate objects	50
4.6.4 WTLS public key certificate objects.....	51
4.6.5 X.509 attribute certificate objects	52
4.7 Key objects	53
4.7.1 Definitions	53
4.7.2 Overview.....	53
4.8 Public key objects	54
4.9 Private key objects.....	56
4.9.1 RSA private key objects	58
4.10 Secret key objects	59
4.11 Domain parameter objects.....	61
4.11.1 Definitions	61
4.11.2 Overview.....	61
4.12 Mechanism objects	61
4.12.1 Definitions	61
4.12.2 Overview.....	61
4.13 Profile objects	62
4.13.1 Definitions.....	62
4.13.2 Overview.....	62
5 Functions	63
5.1 Function return values	66
5.1.1 Universal Cryptoki function return values.....	67
5.1.2 Cryptoki function return values for functions that use a session handle	67
5.1.3 Cryptoki function return values for functions that use a token	67
5.1.4 Special return value for application-supplied callbacks	68
5.1.5 Special return values for mutex-handling functions	68
5.1.6 All other Cryptoki function return values	68
5.1.7 More on relative priorities of Cryptoki errors	73
5.1.8 Error code “gotchas”	74
5.2 Conventions for functions returning output in a variable-length buffer	74
5.3 Disclaimer concerning sample code	75
5.4 General-purpose functions	75
5.4.1 C_Initialize	75
5.4.2 C_Finalize.....	76
5.4.3 C_GetInfo	76
5.4.4 C_GetFunctionList.....	77
5.4.5 C_GetInterfaceList	78
5.4.6 C_GetInterface	79
5.5 Slot and token management functions	81
5.5.1 C_GetSlotList	81
5.5.2 C_GetSlotInfo	82
5.5.3 C_GetTokenInfo	83
5.5.4 C_WaitForSlotEvent.....	83
5.5.5 C_GetMechanismList	84
5.5.6 C_GetMechanismInfo.....	85
5.5.7 C_InitToken	86

5.5.8 C_InitPIN	87
5.5.9 C_SetPIN.....	88
5.6 Session management functions.....	89
5.6.1 C_OpenSession	89
5.6.2 C_CloseSession	90
5.6.3 C_CloseAllSessions	91
5.6.4 C_GetSessionInfo	92
5.6.5 C_SessionCancel.....	92
5.6.6 C_GetOperationState	93
5.6.7 C_SetOperationState	94
5.6.8 C_Login	97
5.6.9 C_LoginUser.....	98
5.6.10 C_Logout.....	99
5.7 Object management functions	100
5.7.1 C_CreateObject.....	100
5.7.2 C_CopyObject	102
5.7.3 C_DestroyObject	103
5.7.4 C_GetObjectSize.....	104
5.7.5 C_GetAttributeValue	105
5.7.6 C_SetAttributeValue	107
5.7.7 C_FindObjectsInit.....	107
5.7.8 C_FindObjects.....	108
5.7.9 C_FindObjectsFinal	109
5.8 Encryption functions	109
5.8.1 C_EncryptInit.....	109
5.8.2 C_Encrypt.....	110
5.8.3 C_EncryptUpdate	111
5.8.4 C_EncryptFinal.....	111
5.9 Message-based encryption functions	113
5.9.1 C_MessageEncryptInit	113
5.9.2 C_EncryptMessage	114
5.9.3 C_EncryptMessageBegin.....	114
5.9.4 C_EncryptMessageNext.....	115
5.9.5 C_MessageEncryptFinal	116
5.10 Decryption functions	118
5.10.1 C_DecryptInit.....	118
5.10.2 C_Decrypt.....	118
5.10.3 C_DecryptUpdate	119
5.10.4 C_DecryptFinal.....	119
5.11 Message-based decryption functions	121
5.11.1 C_MessageDecryptInit	121
5.11.2 C_DecryptMessage	122
5.11.3 C_DecryptMessageBegin.....	123
5.11.4 C_DecryptMessageNext	123
5.11.5 C_MessageDecryptFinal	124
5.12 Message digesting functions	124

5.12.1 C_DigestInit	124
5.12.2 C_Digest	125
5.12.3 C_DigestUpdate	125
5.12.4 C_DigestKey	126
5.12.5 C_DigestFinal	126
5.13 Signing and MACing functions	127
5.13.1 C_SignInit	127
5.13.2 C_Sign	128
5.13.3 C_SignUpdate	129
5.13.4 C_SignFinal	129
5.13.5 C_SignRecoverInit	130
5.13.6 C_SignRecover	130
5.14 Message-based signing and MACing functions	131
5.14.1 C_MessageSignInit	132
5.14.2 C_SignMessage	132
5.14.3 C_SignMessageBegin	133
5.14.4 C_SignMessageNext	133
5.14.5 C_MessageSignFinal	134
5.15 Functions for verifying signatures and MACs	134
5.15.1 C_VerifyInit	134
5.15.2 C_Verify	135
5.15.3 C_VerifyUpdate	136
5.15.4 C_VerifyFinal	136
5.15.5 C_VerifyRecoverInit	137
5.15.6 C_VerifyRecover	137
5.16 Message-based functions for verifying signatures and MACs	139
5.16.1 C_MessageVerifyInit	139
5.16.2 C_VerifyMessage	139
5.16.3 C_VerifyMessageBegin	140
5.16.4 C_VerifyMessageNext	140
5.16.5 C_MessageVerifyFinal	141
5.17 Dual-function cryptographic functions	141
5.17.1 C_DigestEncryptUpdate	141
5.17.2 C_DecryptDigestUpdate	144
5.17.3 C_SignEncryptUpdate	147
5.17.4 C_DecryptVerifyUpdate	149
5.18 Key management functions	152
5.18.1 C_GenerateKey	152
5.18.2 C_GenerateKeyPair	153
5.18.3 C_WrapKey	155
5.18.4 C_UnwrapKey	156
5.18.5 C_DeriveKey	158
5.19 Random number generation functions	160
5.19.1 C_SeedRandom	160
5.19.2 C_GenerateRandom	160
5.20 Parallel function management functions	161

5.20.1 C_GetFunctionStatus	161
5.20.2 C_CancelFunction	161
5.21 Callback functions.....	162
5.21.1 Surrender callbacks.....	162
5.21.2 Vendor-defined callbacks	162
6 PKCS #11 Implementation Conformance	163
Appendix A. Acknowledgments	164
Appendix B. Manifest constants	167
Appendix C. Revision History	168

1 Introduction

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

The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that 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. The supplier of a Cryptoki library implementation typically provides these data types and functions via ANSI C header files. Generic ANSI C header files for Cryptoki are available from the PKCS#11 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 PKCS#11 Mechanisms documents.

1.1 IPR Policy

This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

1.2 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [\[RFC2119\]](#).

1.3 Definitions

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

API	Application programming interface.
Application	Any computer program that calls the Cryptoki interface.
ASN.1	Abstract Syntax Notation One, as defined in X.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 5652)

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 186-4.
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.

1.4 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

82 The following prefixes are used in this standard:

83 *Table 2, Prefixes*

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

84

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

86

```

87  /* an unsigned 8-bit value */
88  typedef unsigned char CK_BYTE;
89
90  /* an unsigned 8-bit character */
91  typedef CK_BYTE CK_CHAR;
92
93  /* an 8-bit UTF-8 character */
94  typedef CK_BYTE CK_UTF8CHAR;
95
96  /* a BYTE-sized Boolean flag */
97  typedef CK_BYTE CK_BBOOL;
98
99  /* 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 define TRUE and FALSE as (CK_DISABLE_TRUE_FALSE may be set by the application vendor):

```

#ifndef CK_DISABLE_TRUE_FALSE

```

```

143 #ifndef FALSE
144 #define FALSE CK_FALSE
145 #endif
146
147 #ifndef TRUE
148 #define TRUE CK_TRUE
149 #endif
150 #endif
151

```

1.5 Normative References

- [FIPS PUB 46-3]** NIST. *FIPS 46-3: Data Encryption Standard*. October 1999.
URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- [FIPS PUB 81]** NIST. *FIPS 81: DES Modes of Operation*. December 1980.
URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>
- [FIPS PUB 186-4]** NIST. *FIPS 186-4: Digital Signature Standard*. July, 2013.
URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>
- [PKCS11-Curr]** *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/os/pkcs11-curr-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html>.
- [PKCS11-Hist]** *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 2.40*. Edited by Susan Gleeson and Chris Zimman. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/os/pkcs11-hist-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-hist/v2.40/pkcs11-hist-v2.40.html>.
- [PKCS11-Prof]** *PKCS #11 Cryptographic Token Interface Profiles Version 2.40*. Edited by Tim Hudson. 14 April 2015. OASIS Standard. <http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/os/pkcs11-profiles-v2.40-os.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-profiles/v2.40/pkcs11-profiles-v2.40.html>.
- [PKCS #1]** RSA Laboratories. *RSA Cryptography Standard*. v2.1, June 14, 2002.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf>
- [PKCS #3]** RSA Laboratories. *Diffie-Hellman Key-Agreement Standard*. v1.4, November 1993.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-3.doc>
- [PKCS #5]** RSA Laboratories. *Password-Based Encryption Standard*. v2.0, March 25, 1999
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>
- [PKCS #7]** RSA Laboratories. *Cryptographic Message Syntax Standard*. v1.5, November 1993
URL : <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-7.doc>
- [PKCS #8]** RSA Laboratories. *Private-Key Information Syntax Standard*. v1.2, November 1993.
URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-8.doc>
- [PKCS11-UG]** *PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40*. Edited by John Leiseboer and Robert Griffin. 16 November 2014. OASIS Committee Note 02. <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cn02/pkcs11-ug-v2.40-cn02.html>. Latest version: <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html>.
- [PKCS #12]** RSA Laboratories. *Personal Information Exchange Syntax Standard*. v1.0, June 1999.

194	[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP
195		14, RFC 2119, March 1997.
196		URL: http://www.ietf.org/rfc/rfc2119.txt .
197	[RFC 2279]	F. Yergeau. <i>RFC 2279: UTF-8, a transformation format of ISO 10646</i> Alis
198		Technologies, January 1998.
199		URL: http://www.ietf.org/rfc/rfc2279.txt
200	[RFC 2534]	Masinter, L., Wing, D., Mutz, A., and K. Holtman. <i>RFC 2534: Media Features for</i>
201		<i>Display, Print, and Fax</i> . March 1999.
202		URL: http://www.ietf.org/rfc/rfc2534.txt
203	[RFC 5652]	R. Housley. <i>RFC 5652: Cryptographic Message Syntax</i> . Septmber 2009. URL:
204		http://www.ietf.org/rfc/rfc5652.txt
205	[RFC 5707]	Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)",
206		RFC 5705, March 2010.
207		URL: http://www.ietf.org/rfc/rfc5705.txt
208	[TLS]	[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246,
209		January 1999. URL: http://www.ietf.org/rfc/rfc2246.txt , superseded by [RFC4346]
210		Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version
211		1.1", RFC 4346, April 2006. URL: http://www.ietf.org/rfc/rfc4346.txt , which was
212		superseded by [TLS12].
213	[TLS12]	[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS)
214		Protocol Version 1.2", RFC 5246, August 2008.
215		URL: http://www.ietf.org/rfc/rfc5246.txt
216	[X.500]	ITU-T. Information Technology — Open Systems Interconnection — The
217		Directory: Overview of Concepts, Models and Services. February 2001. Identical
218		to ISO/IEC 9594-1
219	[X.509]	ITU-T. Information Technology — Open Systems Interconnection — The
220		Directory: Public-key and Attribute Certificate Frameworks. March 2000.
221		Identical to ISO/IEC 9594-8
222	[X.680]	ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1):
223		Specification of Basic Notation. July 2002. Identical to ISO/IEC 8824-1
224	[X.690]	ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic
225		Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished
226		Encoding Rules (DER). July 2002. Identical to ISO/IEC 8825-1
227		

228 1.6 Non-Normative References

229	[ANSI C]	ANSI/ISO. American National Standard for Programming Languages – C. 1990.
230	[CC/PP]	W3C. Composite Capability/Preference Profiles (CC/PP): Structure and
231		Vocabularies. World Wide Web Consortium, January 2004.
232		URL: http://www.w3.org/TR/CCPP-struct-vocab/
233	[CDPD]	Ameritech Mobile Communications et al. Cellular Digital Packet Data System
234		Specifications: Part 406: Airlink Security. 1993.
235	[GCS-API]	X/Open Company Ltd. Generic Cryptographic Service API (GCS-API), Base -
236		Draft 2. February 14, 1995.
237	[ISO/IEC 7816-1]	ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
238		Contacts — Part 1: Physical Characteristics. 1998.
239	[ISO/IEC 7816-4]	ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
240		Contacts — Part 4: Interindustry Commands for Interchange. 1995.
241	[ISO/IEC 8824-1]	ISO. Information Technology-- Abstract Syntax Notation One (ASN.1):
242		Specification of Basic Notation. 2002.
243	[ISO/IEC 8825-1]	ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic
244		Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished
245		Encoding Rules (DER). 2002.

246	[ISO/IEC 9594-1]	ISO. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. 2001.
247		
248	[ISO/IEC 9594-8]	ISO. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. 2001
249		
250	[ISO/IEC 9796-2]	ISO. Information Technology — Security Techniques — Digital Signature Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.
251		
252		
253	[Java MIDP]	Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition. November 2002.
254		URL: http://jcp.org/jsr/detail/118.jsp
255		
256	[MeT-PTD]	MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February 2003.
257		URL: http://www.mobiletransaction.org
258		
259	[PCMCIA]	Personal Computer Memory Card International Association. <i>PC Card Standard</i> , Release 2.1., July 1993.
260		
261	[SEC 1]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography</i> . Version 1.0, September 20, 2000.
262		
263		
264	[SEC 2]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters</i> . Version 1.0, September 20, 2000.
265		
266		
267	[WIM]	WAP. Wireless Identity Module. — WAP-260-WIM-20010712-a. July 2001.
268		URL:
269		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-260-wim-20010712-a.pdf
270		
271	[WPKI]	Wireless Application Protocol: Public Key Infrastructure Definition. — WAP-217-WPKI-20010424-a. April 2001.
272		URL:
273		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-217-wpki-20010424-a.pdf
274		
275		
276	[WTLS]	WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.
277		URL:
278		http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-261-wtls-20010406-a.pdf
279		
280		
281		

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

There is a large array of Cryptoki-related data types that 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 this section.

Platform specific implementation hints can be found in the pkcs11.h header file.

2.1 Structure packing

Cryptoki structures are packed to occupy as little space as is possible. Cryptoki structures **SHALL** be packed with 1-byte alignment.

2.2 Pointer-related macros

Because different platforms and compilers have different ways of dealing with different types of pointers, the following 6 macros **SHALL** 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_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 **SHALL** be 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 **SHALL** 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)(
```



```
321     CK_VOID_PTR pReserved
322 );
323 myC_InitializeType myC_Initialize;
```

324 ♦ CK_CALLBACK_FUNCTION

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

```
332     CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);
333
```

334 or:

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

337 ♦ NULL_PTR

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

3 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 2.

3.1 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 or TLS 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 4 below lists the major and minor version values for the officially published Cryptoki specifications.

Table 4, 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
2.40	0x02	0x28
3.0	0x03	0x00

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:

```

373 typedef struct CK_INFO {
374     CK_VERSION cryptokiVersion;
375     CK_UTF8CHAR manufacturerID[32];
376     CK_FLAGS flags;
377     CK_UTF8CHAR libraryDescription[32];
378     CK_VERSION libraryVersion;
379 } CK_INFO;
380

```

381 The fields of the structure have the following meanings:

382	<i>cryptokiVersion</i>	Cryptoki interface version number, for compatibility with future
383		revisions of this interface
384	<i>manufacturerID</i>	ID of the Cryptoki library manufacturer. MUST be padded with the
385		blank character (' '). Should <i>not</i> be null-terminated.
386	<i>flags</i>	bit flags reserved for future versions. MUST be zero for this version
387	<i>libraryDescription</i>	character-string description of the library. MUST be padded with the
388		blank character (' '). Should <i>not</i> be null-terminated.
389	<i>libraryVersion</i>	Cryptoki library version number

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

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

393 ♦ **CK_NOTIFICATION**

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

```

396 typedef CK_ULONG CK_NOTIFICATION;
397

```

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

```

399 CKN_SURRENDER
400

```

401 The notifications have the following meanings:

402	<i>CKN_SURRENDER</i>	Cryptoki is surrendering the execution of a function executing in a
403		session so that the application may perform other operations. After
404		performing any desired operations, the application should indicate
405		to Cryptoki whether to continue or cancel the function (see Section
406		5.21.1).

407 **3.2 Slot and token types**

408 Cryptoki represents slot and token information with the following types:

409 ♦ **CK_SLOT_ID; CK_SLOT_ID_PTR**

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

```

411 typedef CK_ULONG CK_SLOT_ID;
412

```

A list of **CK_SLOT_IDs** 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 (' '). MUST NOT be null-terminated.

manufacturerID ID of the slot manufacturer. MUST be padded with the blank character (' '). MUST 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 5, 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];
```

```

446 CK_UTF8CHAR manufacturerID[32];
447 CK_UTF8CHAR model[16];
448 CK_CHAR serialNumber[16];
449 CK_FLAGS flags;
450 CK_ULONG ulMaxSessionCount;
451 CK_ULONG ulSessionCount;
452 CK_ULONG ulMaxRwSessionCount;
453 CK_ULONG ulRwSessionCount;
454 CK_ULONG ulMaxPinLen;
455 CK_ULONG ulMinPinLen;
456 CK_ULONG ulTotalPublicMemory;
457 CK_ULONG ulFreePublicMemory;
458 CK_ULONG ulTotalPrivateMemory;
459 CK_ULONG ulFreePrivateMemory;
460 CK_VERSION hardwareVersion;
461 CK_VERSION firmwareVersion;
462 CK_CHAR utcTime[16];
463 } CK_TOKEN_INFO;
464

```

465 The fields of the structure have the following meanings:

466	<i>label</i>	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
467		
468		
469	<i>manufacturerID</i>	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
470		
471	<i>model</i>	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
472		
473	<i>serialNumber</i>	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
474		
475	<i>flags</i>	bit flags indicating capabilities and status of the device as defined below
476		
477	<i>ulMaxSessionCount</i>	maximum number of sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
478		
479		
480	<i>ulSessionCount</i>	number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
481		
482	<i>ulMaxRwSessionCount</i>	maximum number of read/write sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
483		
484		
485	<i>ulRwSessionCount</i>	number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
486		
487	<i>ulMaxPinLen</i>	maximum length in bytes of the PIN
488	<i>ulMinPinLen</i>	minimum length in bytes of the PIN
489	<i>ulTotalPublicMemory</i>	the total amount of memory on the token in bytes in which public objects may be stored (see CK_TOKEN_INFO Note below)
490		

Bit Flag	Mask	Meaning
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.
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.

CK_TOKEN_INFO 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.

3.3 Session types

Cryptoki represents session information with the following types:

562 ♦ CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR

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

```
564 typedef CK_ULONG CK_SESSION_HANDLE;  
565
```

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

```
568 CK_INVALID_HANDLE  
569
```

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

571 ♦ CK_USER_TYPE

572 **CK_USER_TYPE** holds the types of Cryptoki users described in [\[PKCS11-UG\]](#) and, in addition, a
573 context-specific type described in Section 4.9. It is defined as follows:

```
574 typedef CK_ULONG CK_USER_TYPE;  
575
```

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

```
577 CKU_SO  
578 CKU_USER  
579 CKU_CONTEXT_SPECIFIC
```

580 ♦ CK_STATE

581 **CK_STATE** holds the session state, as described in [\[PKCS11-UG\]](#). It is defined as follows:

```
582 typedef CK_ULONG CK_STATE;  
583
```

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

```
585 CKS_RO_PUBLIC_SESSION  
586 CKS_RO_USER_FUNCTIONS  
587 CKS_RW_PUBLIC_SESSION  
588 CKS_RW_USER_FUNCTIONS  
589 CKS_RW_SO_FUNCTIONS
```

590 ♦ CK_SESSION_INFO; CK_SESSION_INFO_PTR

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

```
592 typedef struct CK_SESSION_INFO {  
593     CK_SLOT_ID slotID;  
594     CK_STATE state;  
595     CK_FLAGS flags;  
596     CK_ULONG ulDeviceError;  
597 } CK_SESSION_INFO;  
598
```

599

600 The fields of the structure have the following meanings:

601 *slotID* ID of the slot that interfaces with the token

602 *state* the state of the session

603 *flags* bit flags that define the type of session; the flags are defined below

604 *ulDeviceError* an error code defined by the cryptographic device. Used for errors

605 not covered by Cryptoki.

606 The following table defines the *flags* field:

607 *Table 7, 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

608 CK_SESSION_INFO_PTR is a pointer to a CK_SESSION_INFO.

609 3.4 Object types

610 Cryptoki represents object information with the following types:

611 ♦ CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR

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

```
613 typedef CK_ULONG CK_OBJECT_HANDLE;
614
```

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

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

```
623 CK_INVALID_HANDLE
624
```

625 CK_OBJECT_HANDLE_PTR is a pointer to a CK_OBJECT_HANDLE.

626 ♦ CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR

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

```
629 typedef CK_ULONG CK_OBJECT_CLASS;
630
```

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

633 Vendor defined values for this type may also be specified.

```
634 CKO_VENDOR_DEFINED
635
```

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

638 **CK_OBJECT_CLASS_PTR** is a pointer to a **CK_OBJECT_CLASS**.

639 ♦ **CK_HW_FEATURE_TYPE**

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

```
642 typedef CK_ULONG CK_HW_FEATURE_TYPE;  
643
```

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

646 Vendor defined values for this type may also be specified.

```
647 CKH_VENDOR_DEFINED  
648
```

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

651 ♦ **CK_KEY_TYPE**

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

```
653 typedef CK_ULONG CK_KEY_TYPE;  
654
```

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

657 Vendor defined values for this type may also be specified.

```
658 CKK_VENDOR_DEFINED  
659
```

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

662 ♦ **CK_CERTIFICATE_TYPE**

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

```
664 typedef CK_ULONG CK_CERTIFICATE_TYPE;  
665
```

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

668 Vendor defined values for this type may also be specified.

```
669 CKC_VENDOR_DEFINED  
670
```

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

673 ♦ **CK_CERTIFICATE_CATEGORY**

674 **CK_CERTIFICATE_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

```
675 typedef CK_ULONG CK_CERTIFICATE_CATEGORY;  
676
```

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

Constant	Value	Meaning
CK_CERTIFICATE_CATEGORY_UNSPECIFIED	0x00000000UL	No category specified
CK_CERTIFICATE_CATEGORY_TOKEN_USER	0x00000001UL	Certificate belongs to owner of the token
CK_CERTIFICATE_CATEGORY_AUTHORITY	0x00000002UL	Certificate belongs to a certificate authority
CK_CERTIFICATE_CATEGORY_OTHER_ENTITY	0x00000003UL	Certificate belongs to an end entity (i.e.: not a CA)

678 ♦ CK_ATTRIBUTE_TYPE

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

```
680 typedef CK_ULONG CK_ATTRIBUTE_TYPE;  
681
```

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

684 Vendor defined values for this type may also be specified.

```
685 CKA_VENDOR_DEFINED  
686
```

687 Attribute types **CKA_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
688 interoperability, vendors should register their attribute types through the PKCS process.

689 ♦ CK_ATTRIBUTE; CK_ATTRIBUTE_PTR

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

```
692 typedef struct CK_ATTRIBUTE {  
693     CK_ATTRIBUTE_TYPE type;  
694     CK_VOID_PTR pValue;  
695     CK_ULONG ulValueLen;  
696 } CK_ATTRIBUTE;  
697
```

698 The fields of the structure have the following meanings:

699 *type* the attribute type

700 *pValue* pointer to the value of the attribute

701 *ulValueLen* length in bytes of the value

702 If an attribute has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. An array of
703 **CK_ATTRIBUTES** is called a “template” and is used for creating, manipulating and searching for objects.
704 The order of the attributes in a template *never* matters, even if the template contains vendor-specific
705 attributes. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the
706 application and Cryptoki library **MUST** ensure that the pointer can be safely cast to the expected type
707 (i.e., without word-alignment errors).

708
709 The constant **CK_UNAVAILABLE_INFORMATION** is used in the *ulValueLen* field to denote an invalid or
710 unavailable value. See **C_GetAttributeValue** for further details.

711
712 **CK_ATTRIBUTE_PTR** is a pointer to a **CK_ATTRIBUTE**.

713 ♦ **CK_DATE**

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

```
715     typedef struct CK_DATE {  
716         CK_CHAR year[4];  
717         CK_CHAR month[2];  
718         CK_CHAR day[2];  
719     } CK_DATE;  
720
```

721 The fields of the structure have the following meanings:

722	<i>year</i>	the year ("1900" - "9999")
723	<i>month</i>	the month ("01" - "12")
724	<i>day</i>	the day ("01" - "31")

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

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

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

731 ♦ **CK_PROFILE_ID; CK_PROFILE_ID_PTR**

732 **CK_PROFILE_ID** is an unsigned long value representing a specific token profile. It is defined as follows:

```
733     typedef CK_ULONG CK_PROFILE_ID;  
734
```

735 Profiles are defined in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater
736 than 0xffffffff may cause compatibility issues on platforms that have **CK_ULONG** values of 32 bits, and
737 should be avoided.

738 Vendor defined values for this type may also be specified.

```
739     CKP_VENDOR_DEFINED  
740
```

741 Profile IDs **CKP_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
742 interoperability, vendors should register their object classes through the PKCS process.

743
744 *Valid Profile IDs in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki defines
745 the following symbolic value:

```
746     CKP_INVALID_ID
```

747 **CK_PROFILE_ID_PTR** is a pointer to a **CK_PROFILE_ID**.

◆ CK_JAVA_MIDP_SECURITY_DOMAIN

CK_JAVA_MIDP_SECURITY_DOMAIN is a value that identifies the Java MIDP security domain of a certificate. It is defined as follows:

```
typedef CK_ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

For this version of Cryptoki, the following security domains are defined. See the Java MIDP specification for further information:

Constant	Value	Meaning
CK_SECURITY_DOMAIN_UNSPECIFIED	0x00000000UL	No domain specified
CK_SECURITY_DOMAIN_MANUFACTURER	0x00000001UL	Manufacturer protection domain
CK_SECURITY_DOMAIN_OPERATOR	0x00000002UL	Operator protection domain
CK_SECURITY_DOMAIN_THIRD_PARTY	0x00000003UL	Third party protection domain

3.5 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

779 *pParameter* pointer to the parameter if required by the mechanism

780	<i>ulParameterLen</i>	length in bytes of the parameter
-----	-----------------------	----------------------------------

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

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

```
785  ♦ CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR
```

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

```
788     typedef struct CK_MECHANISM_INFO {
789         CK_ULONG ulMinKeySize;
790         CK_ULONG ulMaxKeySize;
791         CK_FLAGS flags;
792     } CK_MECHANISM_INFO;
793
```

794 The fields of the structure have the following meanings:

795	<i>ulMinKeySize</i>	the minimum size of the key for the mechanism (whether this is
796		measured in bits or in bytes is mechanism-dependent)

797	<i>ulMaxKeySize</i>	the maximum size of the key for the mechanism (whether this is
798		measured in bits or in bytes is mechanism-dependent)

```
799      flags      bit flags specifying mechanism capabilities
```

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

801 The following table defines the *flags* field:

802 Table 8, 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_MESSAGE_ENCRYPT	0x00000002	True if the mechanism can be used with C_MessageEncryptInit
CKF_MESSAGE_DECRYPT	0x00000004	True if the mechanism can be used with C_MessageDecryptInit
CKF_MESSAGE_SIGN	0x00000008	True if the mechanism can be used with C_MessageSignInit
CKF_MESSAGE_VERIFY	0x00000010	True if the mechanism can be used with C_MessageVerifyInit
CKF_MULTI_MESSAGE	0x00000020	True if the mechanism can be used with C_*MessageBegin . One of CKF_MESSAGE_* flag must also be set.
CKF_FIND_OBJECTS	0x00000040	This flag can be passed in as a parameter to C_SessionCancel to cancel an active object search operation. Any other use of this flag is outside the scope of this standard.

Bit Flag	Mask	Meaning
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.

803 CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

804 3.6 Function types

805 Cryptoki represents information about functions with the following data types:

806 ♦ CK_RV

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

```
808 typedef CK_ULONG CK_RV;
809
```

810 Vendor defined values for this type may also be specified.

```
811 CKR_VENDOR_DEFINED
812
```

813 Section 5.1 defines the meaning of each **CK_RV** value. Return values **CKR_VENDOR_DEFINED** and
814 above are permanently reserved for token vendors. For interoperability, vendors should register their
815 return values through the PKCS process.

816 ♦ **CK_NOTIFY**

817 **CK_NOTIFY** is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is
818 defined as follows:

```
819 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (  
820     CK_SESSION_HANDLE hSession,  
821     CK_NOTIFICATION event,  
822     CK_VOID_PTR pApplication  
823 );  
824
```

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

826	<i>hSession</i>	The handle of the session performing the callback
827	<i>event</i>	The type of notification callback
828	<i>pApplication</i>	An application-defined value. This is the same value as was passed 829 to C_OpenSession to open the session performing the callback

830 ♦ **CK_C_XXX**

831 Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the
832 Cryptoki API (see Section 4.12 for detailed information about each of them), Cryptoki defines a type
833 **CK_C_XXX**, which is a pointer to a function with the same arguments and return value as **C_XXX** has.
834 An appropriately-set variable of type **CK_C_XXX** may be used by an application to call the Cryptoki
835 function **C_XXX**.

836 ♦ **CK_FUNCTION_LIST;** **CK_FUNCTION_LIST_PTR;** 837 **CK_FUNCTION_LIST_PTR_PTR**

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

```
840 typedef struct CK_FUNCTION_LIST {  
841     CK_VERSION version;  
842     CK_C_Initialize C_Initialize;  
843     CK_C_Finalize C_Finalize;  
844     CK_C_GetInfo C_GetInfo;  
845     CK_C_GetFunctionList C_GetFunctionList;  
846     CK_C_GetSlotList C_GetSlotList;  
847     CK_C_GetSlotInfo C_GetSlotInfo;  
848     CK_C_GetTokenInfo C_GetTokenInfo;  
849     CK_C_GetMechanismList C_GetMechanismList;  
850     CK_C_GetMechanismInfo C_GetMechanismInfo;  
851     CK_C_InitToken C_InitToken;  
852     CK_C_InitPIN C_InitPIN;  
853     CK_C_SetPIN C_SetPIN;  
854     CK_C_OpenSession C_OpenSession;  
855     CK_C_CloseSession C_CloseSession;  
856     CK_C_CloseAllSessions C_CloseAllSessions;  
857     CK_C_GetSessionInfo C_GetSessionInfo;  
858  
859     CK_C_GetOperationState C_GetOperationState;  
860     CK_C_SetOperationState C_SetOperationState;  
861     CK_C_Login C_Login;  
862     CK_C_Logout C_Logout;  
863     CK_C_CreateObject C_CreateObject;  
864     CK_C_CopyObject C_CopyObject;  
865     CK_C_DestroyObject C_DestroyObject;
```

```

866 CK_C_GetObjectSize C_GetObjectSize;
867 CK_C_GetAttributeValue C_GetAttributeValue;
868 CK_C_SetAttributeValue C_SetAttributeValue;
869 CK_C_FindObjectsInit C_FindObjectsInit;
870 CK_C_FindObjects C_FindObjects;
871 CK_C_FindObjectsFinal C_FindObjectsFinal;
872 CK_C_EncryptInit C_EncryptInit;
873 CK_C_Encrypt C_Encrypt;
874 CK_C_EncryptUpdate C_EncryptUpdate;
875 CK_C_EncryptFinal C_EncryptFinal;
876 CK_C_DecryptInit C_DecryptInit;
877 CK_C_Decrypt C_Decrypt;
878 CK_C_DecryptUpdate C_DecryptUpdate;
879 CK_C_DecryptFinal C_DecryptFinal;
880 CK_C_DigestInit C_DigestInit;
881 CK_C_Digest C_Digest;
882 CK_C_DigestUpdate C_DigestUpdate;
883 CK_C_DigestKey C_DigestKey;
884 CK_C_DigestFinal C_DigestFinal;
885 CK_C_SignInit C_SignInit;
886 CK_C_Sign C_Sign;
887 CK_C_SignUpdate C_SignUpdate;
888 CK_C_SignFinal C_SignFinal;
889 CK_C_SignRecoverInit C_SignRecoverInit;
890 CK_C_SignRecover C_SignRecover;
891 CK_C_VerifyInit C_VerifyInit;
892 CK_C_Verify C_Verify;
893 CK_C_VerifyUpdate C_VerifyUpdate;
894 CK_C_VerifyFinal C_VerifyFinal;
895 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
896 CK_C_VerifyRecover C_VerifyRecover;
897 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
898 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
899 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
900 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
901 CK_C_GenerateKey C_GenerateKey;
902 CK_C_GenerateKeyPair C_GenerateKeyPair;
903 CK_C_WrapKey C_WrapKey;
904 CK_C_UnwrapKey C_UnwrapKey;
905 CK_C_DeriveKey C_DeriveKey;
906 CK_C_SeedRandom C_SeedRandom;
907 CK_C_GenerateRandom C_GenerateRandom;
908 CK_C_GetFunctionStatus C_GetFunctionStatus;
909 CK_C_CancelFunction C_CancelFunction;
910 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
911 } CK_FUNCTION_LIST;
912

```

913 Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it
914 which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section
915 5.2). The value that this pointer points to can be used by an application to quickly find out where the
916 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API
917 MUST have an entry point defined in the Cryptoki library's **CK_FUNCTION_LIST** structure. If a particular
918 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the
919 library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns
920 CKR_FUNCTION_NOT_SUPPORTED.

921 In this structure 'version' is the cryptoki specification version number. The major and minor versions must
922 be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for
923 this version of the specification may be returned via **C_GetInterfaceList** or **C_GetInterface**.

924

925 An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST**
926 structure. Whether or not it can, it should never attempt to do so.
927 PKCS #11 modules must not add new functions at the end of the **CK_FUNCTION_LIST** that are not
928 contained within the defined structure. If a PKCS#11 module needs to define additional functions, they
929 should be placed within a vendor defined interface returned via **C_GetInterfaceList** or **C_GetInterface**.
930 **CK_FUNCTION_LIST_PTR** is a pointer to a **CK_FUNCTION_LIST**.
931 **CK_FUNCTION_LIST_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_PTR**.
932

933 ♦ **CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR;**
934 **CK_FUNCTION_LIST_3_0_PTR_PTR**

935 **CK_FUNCTION_LIST_3_0** is a structure which contains the same function pointers as in
936 **CK_FUNCTION_LIST** and additional functions added to the end of the structure that were defined in
937 Cryptoki version 3.0. It is defined as follows:

```
938 typedef struct CK_FUNCTION_LIST_3_0 {  
939     CK_VERSION version;  
940     CK_C_Initialize C_Initialize;  
941     CK_C_Finalize C_Finalize;  
942     CK_C_GetInfo C_GetInfo;  
943     CK_C_GetFunctionList C_GetFunctionList;  
944     CK_C_GetSlotList C_GetSlotList;  
945     CK_C_GetSlotInfo C_GetSlotInfo;  
946     CK_C_GetTokenInfo C_GetTokenInfo;  
947     CK_C_GetMechanismList C_GetMechanismList;  
948     CK_C_GetMechanismInfo C_GetMechanismInfo;  
949     CK_C_InitToken C_InitToken;  
950     CK_C_InitPIN C_InitPIN;  
951     CK_C_SetPIN C_SetPIN;  
952     CK_C_OpenSession C_OpenSession;  
953     CK_C_CloseSession C_CloseSession;  
954     CK_C_CloseAllSessions C_CloseAllSessions;  
955     CK_C_GetSessionInfo C_GetSessionInfo;  
956     CK_C_GetOperationState C_GetOperationState;  
957     CK_C_SetOperationState C_SetOperationState;  
958     CK_C_Login C_Login;  
959     CK_C_Logout C_Logout;  
960     CK_C_CreateObject C_CreateObject;  
961     CK_C_CopyObject C_CopyObject;  
962     CK_C_DestroyObject C_DestroyObject;  
963     CK_C_GetObjectSize C_GetObjectSize;  
964     CK_C_GetAttributeValue C_GetAttributeValue;  
965     CK_C_SetAttributeValue C_SetAttributeValue;  
966     CK_C_FindObjectsInit C_FindObjectsInit;  
967     CK_C_FindObjects C_FindObjects;  
968     CK_C_FindObjectsFinal C_FindObjectsFinal;  
969     CK_C_EncryptInit C_EncryptInit;  
970     CK_C_Encrypt C_Encrypt;  
971     CK_C_EncryptUpdate C_EncryptUpdate;  
972     CK_C_EncryptFinal C_EncryptFinal;  
973     CK_C_DecryptInit C_DecryptInit;  
974     CK_C_Decrypt C_Decrypt;  
975     CK_C_DecryptUpdate C_DecryptUpdate;  
976     CK_C_DecryptFinal C_DecryptFinal;  
977     CK_C_DigestInit C_DigestInit;  
978     CK_C_Digest C_Digest;  
979     CK_C_DigestUpdate C_DigestUpdate;  
980     CK_C_DigestKey C_DigestKey;  
981     CK_C_DigestFinal C_DigestFinal;
```

```

982 CK_C_SignInit C_SignInit;
983 CK_C_Sign C_Sign;
984 CK_C_SignUpdate C_SignUpdate;
985 CK_C_SignFinal C_SignFinal;
986 CK_C_SignRecoverInit C_SignRecoverInit;
987 CK_C_SignRecover C_SignRecover;
988 CK_C_VerifyInit C_VerifyInit;
989 CK_C_Verify C_Verify;
990 CK_C_VerifyUpdate C_VerifyUpdate;
991 CK_C_VerifyFinal C_VerifyFinal;
992 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
993 CK_C_VerifyRecover C_VerifyRecover;
994 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
995 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
996 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
997 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
998 CK_C_GenerateKey C_GenerateKey;
999 CK_C_GenerateKeyPair C_GenerateKeyPair;
1000 CK_C_WrapKey C_WrapKey;
1001 CK_C_UnwrapKey C_UnwrapKey;
1002 CK_C_DeriveKey C_DeriveKey;
1003 CK_C_SeedRandom C_SeedRandom;
1004 CK_C_GenerateRandom C_GenerateRandom;
1005 CK_C_GetFunctionStatus C_GetFunctionStatus;
1006 CK_C_CancelFunction C_CancelFunction;
1007 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1008 CK_C_GetInterfaceList C_GetInterfaceList;
1009 CK_C_GetInterface C_GetInterface;
1010 CK_C_LoginUser C_LoginUser;
1011 CK_C_SessionCancel C_SessionCancel;
1012 CK_C_MessageEncryptInit C_MessageEncryptInit;
1013 CK_C_EncryptMessage C_EncryptMessage;
1014 CK_C_EncryptMessageBegin C_EncryptMessageBegin;
1015 CK_C_EncryptMessageNext C_EncryptMessageNext;
1016 CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1017 CK_C_MessageDecryptInit C_MessageDecryptInit;
1018 CK_C_DecryptMessage C_DecryptMessage;
1019 CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1020 CK_C_DecryptMessageNext C_DecryptMessageNext;
1021 CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1022 CK_C_MessageSignInit C_MessageSignInit;
1023 CK_C_SignMessage C_SignMessage;
1024 CK_C_SignMessageBegin C_SignMessageBegin;
1025 CK_C_SignMessageNext C_SignMessageNext;
1026 CK_C_MessageSignFinal C_MessageSignFinal;
1027 CK_C_MessageVerifyInit C_MessageVerifyInit;
1028 CK_C_VerifyMessage C_VerifyMessage;
1029 CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1030 CK_C_VerifyMessageNext C_VerifyMessageNext;
1031 CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1032 } CK_FUNCTION_LIST_3_0;
1033

```

1034 For a general description of **CK_FUNCTION_LIST_3_0** see **CK_FUNCTION_LIST**.

1035 In this structure, *version* is the cryptoki specification version number. It should match the value of
1036 *cryptokiVersion* returned in the **CK_INFO** structure, but must be 3.0 at minimum.

1037 This function list may be returned via **C_GetInterfaceList** or **C_GetInterface**

1038 **CK_FUNCTION_LIST_3_0_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0**.

1039 **CK_FUNCTION_LIST_3_0_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0_PTR**.

◆ **CK_INTERFACE; CK_INTERFACE_PTR; CK_INTERFACE_PTR_PTR**

CK_INTERFACE is a structure which contains an interface name with a function list and flag.
It is defined as follows:

```
typedef struct CK_INTERFACE {  
    CK_UTF8CHAR_PTR pInterfaceName;  
    CK_VOID_PTR      pFunctionList;  
    CK_FLAGS         flags;  
} CK_INTERFACE;
```

The fields of the structure have the following meanings:

<i>pInterfaceName</i>	the name of the interface
<i>pFunctionList</i>	the interface function list which must always begin with a CK_VERSION structure as the first field
<i>flags</i>	bit flags specifying interface capabilities

The interface name “PKCS 11” is reserved for use by interfaces defined within the cryptoki specification. Interfaces starting with the string: “Vendor ” are reserved for vendor use and will not otherwise be defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface names of “Vendor {vendor name}”. For example “Vendor ACME Inc”.

The following table defines the flags field:

Table 9, CK_INTERFACE Flags

Bit Flag	Mask	Meaning
CKF_INTERFACE_FORK_SAFE	0x00000001	The returned interface will have fork tolerant semantics. When the application forks, each process will get its own copy of all session objects, session states, login states, and encryption states. Each process will also maintain access to token objects with their previously supplied handles.

CK_INTERFACE_PTR is a pointer to a **CK_INTERFACE**.

CK_INTERFACE_PTR_PTR is a pointer to a **CK_INTERFACE_PTR**.

3.7 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:

```

1070 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX) (
1071     CK_VOID_PTR_PTR ppMutex
1072 );
1073

```

1074 Calling a CK_CREATEMUTEX function returns the pointer to the new mutex object in the location pointed
1075 to by ppMutex. Such a function should return one of the following values:

```

1076 CKR_OK, CKR_GENERAL_ERROR
1077 CKR_HOST_MEMORY

```

1078 ♦ CK_DESTROYMUTEX

1079 **CK_DESTROYMUTEX** is the type of a pointer to an application-supplied function which destroys an
1080 existing mutex object. It is defined as follows:

```

1081 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX) (
1082     CK_VOID_PTR pMutex
1083 );
1084

```

1085 The argument to a CK_DESTROYMUTEX function is a pointer to the mutex object to be destroyed. Such
1086 a function should return one of the following values:

```

1087 CKR_OK, CKR_GENERAL_ERROR
1088 CKR_HOST_MEMORY
1089 CKR_MUTEX_BAD

```

1090 ♦ CK_LOCKMUTEX and CK_UNLOCKMUTEX

1091 **CK_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing
1092 mutex object. **CK_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which
1093 unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- 1094 • If a CK_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a
1095 lock on that mutex and returns.
- 1096 • If a CK_LOCKMUTEX function is called on a mutex which is locked by some thread other than the
1097 calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- 1098 • If a CK_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the
1099 behavior of the function call is undefined.
- 1100 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that
1101 mutex is unlocked and the function call returns. Furthermore:
 - 1102 ○ If exactly one thread was blocking on that particular mutex, then that thread stops blocking,
1103 obtains a lock on that mutex, and its CK_LOCKMUTEX call returns.
 - 1104 ○ If more than one thread was blocking on that particular mutex, then exactly one of the
1105 blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on
1106 the mutex, and its CK_LOCKMUTEX call returns. All other threads blocking on that particular
1107 mutex continue to block.
- 1108 • If a CK_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call
1109 returns the error code CKR_MUTEX_NOT_LOCKED.
- 1110 • If a CK_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the
1111 calling thread, the behavior of the function call is undefined.

1112 **CK_LOCKMUTEX** is defined as follows:

```

1113 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX) (
1114     CK_VOID_PTR pMutex
1115 );

```

1116

1117 The argument to a CK_LOCKMUTEX function is a pointer to the mutex object to be locked. Such a
1118 function should return one of the following values:

1119 CKR_OK, CKR_GENERAL_ERROR
1120 CKR_HOST_MEMORY,
1121 CKR_MUTEX_BAD
1122

1123 **CK_UNLOCKMUTEX** is defined as follows:

1124 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX) (
1125 CK_VOID_PTR pMutex
1126);
1127

1128 The argument to a CK_UNLOCKMUTEX function is a pointer to the mutex object to be unlocked. Such a
1129 function should return one of the following values:

1130 CKR_OK, CKR_GENERAL_ERROR
1131 CKR_HOST_MEMORY
1132 CKR_MUTEX_BAD
1133 CKR_MUTEX_NOT_LOCKED

1134 ♦ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

1135 **CK_C_INITIALIZE_ARGS** is a structure containing the optional arguments for the **C_Initialize** function.
1136 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals
1137 with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

1138 typedef struct CK_C_INITIALIZE_ARGS {
1139 CK_CREATEMUTEX CreateMutex;
1140 CK_DESTROYMUTEX DestroyMutex;
1141 CK_LOCKMUTEX LockMutex;
1142 CK_UNLOCKMUTEX UnlockMutex;
1143 CK_FLAGS flags;
1144 CK_VOID_PTR pReserved;
1145 } CK_C_INITIALIZE_ARGS;
1146

1147 The fields of the structure have the following meanings:

1148	<i>CreateMutex</i>	pointer to a function to use for creating mutex objects
1149	<i>DestroyMutex</i>	pointer to a function to use for destroying mutex objects
1150	<i>LockMutex</i>	pointer to a function to use for locking mutex objects
1151	<i>UnlockMutex</i>	pointer to a function to use for unlocking mutex objects
1152	<i>flags</i>	bit flags specifying options for C_Initialize ; the flags are defined below
1153		
1154	<i>pReserved</i>	reserved for future use. Should be NULL_PTR for this version of
1155		Cryptoki

1156 The following table defines the flags field:

1157 *Table 10, 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 <i>not</i> 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

1158 CK_C_INITIALIZE_ARGS_PTR is a pointer to a CK_C_INITIALIZE_ARGS.

4 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:

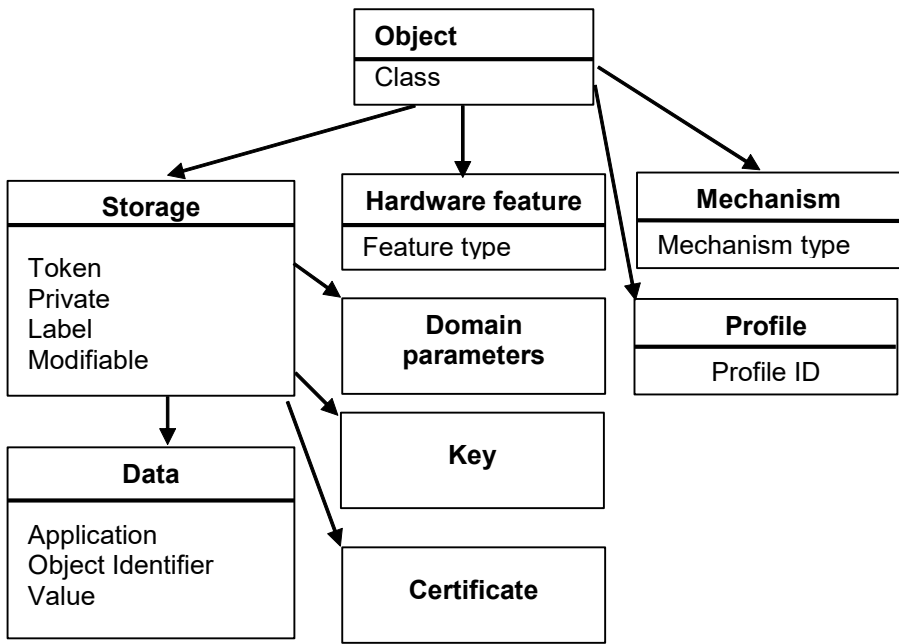


Figure 1, Object Attribute Hierarchy

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 4 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_BYTES
Big integer	a string of CK_BYTES 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_CHARS (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.

4.1 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 5.18) 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 [PKCS11-Curr] and [PKCS11-Hist] for specification of mechanisms for PKCS #11). 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.

4.1.1 Creating objects

Objects may be created with the Cryptoki functions **C_CreateObject** (see Section 5.7), **C_GenerateKey**, **C_GenerateKeyPair**, **C_UnwrapKey**, and **C_DeriveKey** (see Section 5.18). 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 4.1.3.

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.

4.1.2 Modifying objects

Objects may be modified with the Cryptoki function **C_SetAttributeValue** (see Section 5.7). 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 4.1.1—and the error codes they return—apply to modifying objects with **C_SetAttributeValue**, except for the possibility of a template being incomplete.

4.1.3 Copying objects

Unless an object's CKA_COPYABLE (see Table 17) attribute is set to CK_FALSE, it may be copied with the Cryptoki function **C_CopyObject** (see Section 5.7). 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 four special attributes **CKA_TOKEN**, **CKA_PRIVATE**, **CKA_MODIFIABLE** and **CKA_DESTROYABLE**. 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_ACTION_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.

4.2 Common attributes

Table 11, 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 12, Common Object Attributes

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

Refer to Table 11 for footnotes

The above table defines the attributes common to all objects.

4.3 Hardware Feature Objects

4.3.1 Definitions

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

4.3.2 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 13, Hardware Feature Common Attributes

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE ¹	CK_HW_FEATURE_TYPE	Hardware feature (type)

Refer to Table 11 for footnotes

4.3.3 Clock

4.3.3.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type CK_HW_FEATURE_TYPE.

4.3.3.2 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 14, 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.

4.3.4 Monotonic Counter Objects

4.3.4.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type CK_HW_FEATURE_TYPE.

4.3.4.2 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 15, 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.

¹Read Only

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

4.3.5 User Interface Objects

4.3.5.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type CK_HW_FEATURE_TYPE.

4.3.5.2 Description

User interface objects represent the presentation capabilities of the device.

Table 16, 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.

4.4 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.

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.
CKA_DESTROYABLE	CK_BBOOL	CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.
CKA_UNIQUE_ID ²⁴⁶	RFC2279 string	The unique identifier assigned to the object.

1338 Only the **CKA_LABEL** attribute can be modified after the object is created. (The **CKA_TOKEN**,
1339 **CKA_PRIVATE**, and **CKA_MODIFIABLE** attributes can be changed in the process of copying an object,
1340 however.)

1341 The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

1342 When the **CKA_PRIVATE** attribute is CK_TRUE, a user may not access the object until the user has
1343 been authenticated to the token.

1344 The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only.

1345 The **CKA_LABEL** attribute is intended to assist users in browsing.

1346 The value of the CKA_COPYABLE attribute determines whether or not an object can be copied. This
1347 attribute can be used in conjunction with CKA_MODIFIABLE to prevent changes to the permitted usages
1348 of keys and other objects.

1349 The value of the CKA_DESTROYABLE attribute determines whether the object can be destroyed using
1350 C_DestroyObject.

1351 4.4.1 The CKA_UNIQUE_ID attribute

1352 Any time a new object is created, a value for CKA_UNIQUE_ID MUST be generated by the token and
1353 stored with the object. The specific algorithm used to generate unique ID values for objects is token-
1354 specific, but values generated MUST be unique across all objects visible to any particular session, and
1355 SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling
1356 C_InitToken, MAY cause reuse of CKA_UNIQUE_ID values.

1357 Any attempt to modify the CKA_UNIQUE_ID attribute of an existing object or to specify the value of the
1358 CKA_UNIQUE_ID attribute in the template for an operation that creates one or more objects MUST fail.
1359 Operations failing for this reason return the error code CKR_ATTRIBUTE_READ_ONLY.

1360

4.5 Data objects

4.5.1 Definitions

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

4.5.2 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 18, 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)}
};
```

4.6 Certificate objects

4.6.1 Definitions

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

4.6.2 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:

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	CKA_CERTIFICATE_CATEGORY	(default CK_CERTIFICATE_CATEGORY_UNSPECIFIED)
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)
CKA_PUBLIC_KEY_INFO	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)

1400 Refer to Table 11 for footnotes

1401 Cryptoki does not enforce the relationship of the CKA_PUBLIC_KEY_INFO to the public key in the
 1402 certificate, but does recommend that the key be extracted from the certificate to create this value.

1403 The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of
 1404 Cryptoki supports the following certificate types:

- 1405 • X.509 public key certificate
- 1406 • WTLS public key certificate
- 1407 • X.509 attribute certificate

1408 The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It MUST be set by a token
 1409 initialization application or by the token's SO. Trusted certificates cannot be modified.

1410 The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user
 1411 certificate for which the corresponding private key is available on the token ("token user"), a CA certificate
 1412 ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an
 1413 object is created.

1414 The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to
 1415 the categorization of the certificates.

1416 **CKA_CHECK_VALUE**: The value of this attribute is derived from the certificate by taking the first three
 1417 bytes of the SHA-1 hash of the certificate object's CKA_VALUE attribute.

1418 The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not
 1419 attach any special meaning to them. When present, the application is responsible to set them to values
 1420 that match the certificate's encoded "not before" and "not after" fields (if any).

4.6.3 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 20, 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_JAVA_MIDP_SECURITY_DOMAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)
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.

¹MUST be specified when the object is created.

²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

³MUST be non-empty if CKA_VALUE is empty.

⁴Can 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)}
};
```

4.6.4 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 21: 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

Attribute	Data type	Meaning
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.

¹MUST be specified when the object is created. Can only be empty if CKA_VALUE is empty.

²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

³MUST be non-empty if CKA_VALUE is empty.

⁴Can 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].

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)}
};

```

4.6.5 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:

1512 Table 22, 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.

1513 ¹MUST be specified when the object is created

1514 Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be

1515 modified after the object is created.

1516 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)}
};
```

1531 **4.7 Key objects**

1532 **4.7.1 Definitions**

1533 There is no CKO_ definition for the base key object class, only for the key types derived from it.

1534 This section defines the object class CKO_PUBLIC_KEY, CKO_PRIVATE_KEY and

1535 CKO_SECRET_KEY for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

1536 **4.7.2 Overview**

1537 Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret

1538 keys. The following common footnotes apply to all the tables describing attributes of keys:

1539 The following table defines the attributes common to public key, private key and secret key classes, in

1540 addition to the common attributes defined for this object class:

1541 Table 23, 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>i.e.</i> , 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 <ul style="list-style-type: none"> generated locally (<i>i.e.</i>, 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.
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 <i>ulValueLen</i> component of the attribute divided by the size of CK_MECHANISM_TYPE.

1542 Refer to Table 11 for footnotes

1543 The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys,
1544 this field assists in handling multiple keys held by the same subject; the key identifier for a public key and
1545 its corresponding private key should be the same. The key identifier should also be the same as for the
1546 corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See
1547 Section 0 for further commentary.)

1548 In the case of secret keys, the meaning of the **CKA_ID** attribute is up to the application.

1549 Note that the **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does
1550 not attach any special meaning to them. In particular, it does not restrict usage of a key according to the
1551 dates; doing this is up to the application.

1552 The **CKA_DERIVE** attribute has the value CK_TRUE if and only if it is possible to derive other keys from
1553 the key.

1554 The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the key was originally
1555 generated on the token by a **C_GenerateKey** or **C_GenerateKeyPair** call.

1556 The **CKA_KEY_GEN_MECHANISM** attribute identifies the key generation mechanism used to generate
1557 the key material. It contains a valid value only if the **CKA_LOCAL** attribute has the value CK_TRUE. If
1558 **CKA_LOCAL** has the value CK_FALSE, the value of the attribute is
1559 CK_UNAVAILABLE_INFORMATION.

1560 4.8 Public key objects

1561 Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. The following table defines the
1562 attributes common to all public keys, in addition to the common attributes defined for this object class:

Attribute	Data type	Meaning
CKA_SUBJECT ⁸	Byte array	DER-encoding of the key subject name (default empty)
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>i.e.</i> , 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 <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_PUBLIC_KEY_INFO	Byte array	DER-encoding of the SubjectPublicKeyInfo for this public key. (MAY be empty, DEFAULT derived from the underlying public key data)

1564 Refer to Table 11 for footnotes

1565 It is intended in the interests of interoperability that the subject name and key identifier for a public key will
 1566 be the same as those for the corresponding certificate and private key. However, Cryptoki does not
 1567 enforce this, and it is not required that the certificate and private key also be stored on the token.

1568 To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for
 1569 public keys, use the following table.

1570 Table 25, 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

1571 The value of the CKA_PUBLIC_KEY_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:

```
1572     SubjectPublicKeyInfo ::= SEQUENCE {
1573         algorithm          AlgorithmIdentifier,
1574         subjectPublicKey     BIT_STRING }
```

1575 The encodings for the subjectPublicKey field are specified in the description of the public key types in the
1576 appropriate [PKCS11-Curr] document for the key types defined within this specification.

1577 4.9 Private key objects

1578 Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. The following table defines the
1579 attributes common to all private keys, in addition to the common attributes defined for this object class:

1580 Table 26, 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>i.e.</i> , 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 <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> had the 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.

Attribute	Data type	Meaning
		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 <i>ulValueLen</i> 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.
CKA_PUBLIC_KEY_INFO ⁸	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the associated public key (MAY be empty; DEFAULT derived from the underlying private key data; MAY be manually set for specific key types; if set; MUST be consistent with the underlying private key data)

⁸ Refer to Table 11 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.

The **CKA_PUBLIC_KEY_INFO** attribute represents the public key associated with this private key. The data it represents may either be stored as part of the private key data, or regenerated as needed from the private key.

If this attribute is supplied as part of a template for **C_CreateObject**, **C_CopyObject** or **C_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key data and the public key data as supplied in **CKA_PUBLIC_KEY_INFO**. This can be done either by deriving a public key from the private key and comparing the values, or by doing a sign and verify operation. If there is a mismatch, the command SHALL return **CKR_ATTRIBUTE_VALUE_INVALID**. A token MAY choose not to support the **CKA_PUBLIC_KEY_INFO** attribute for commands which create new private keys. If it does not support the attribute, the command SHALL return **CKR_ATTRIBUTE_TYPE_INVALID**.

As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public key information. In particular, the RSA private key description has been modified in <this version> to add the **CKA_PUBLIC_EXPONENT** to the list of attributes required for an RSA private key. All other private key types described in this specification contain sufficient information to recover the associated public key.

4.9.1 RSA private key objects

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

Table 27, RSA Private Key Object Attributes

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

Refer to Table 11 for footnotes

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

Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above attributes, which can assist in performing rapid RSA computations. Other tokens might store only the **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values. Effective with version 2.40, tokens MUST also store **CKA_PUBLIC_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the associated public key.

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 27 it keeps track of. Later, if an application asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed efficiently from these four values). However, a Cryptoki implementation may or may not actually do this extra computation. The only attributes from Table 27 for which a Cryptoki

implementation is *required* to be able to return values are **CKA_MODULUS**,
CKA_PRIVATE_EXPONENT, and **CKA_PUBLIC_EXPONENT**. A token SHOULD also be able to return
CKA_PUBLIC_KEY_INFO for an RSA private key. See the general guidance for Private Keys above.

4.10 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 28, 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>i.e.</i> , authentication codes) where the signature is an appendix to the data ⁹
CKA_VERIFY ⁸	CK_BBOOL	CK_TRUE if key supports verification (<i>i.e.</i> , 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>i.e.</i> , can be used to wrap other keys) ⁹
CKA_UNWRAP ⁸	CK_BBOOL	CK_TRUE if key supports unwrapping (<i>i.e.</i> , 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 <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> 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 keys wrapped using this wrapping key. Keys that do not

Attribute	Data type	Meaning
		match cannot be wrapped. The number of attributes in the array is the <i>ulValueLen</i> 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 <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.

1652 Refer to Table 11 for footnotes

1653 If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE,
1654 then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which
1655 attributes these are is specified for each type of secret key in the attribute table in the section describing
1656 that type of key.

1657 The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE**, of
1658 type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to
1659 be used to cross-check symmetric keys against other systems where the same key is shared, and as a
1660 validity check after manual key entry or restore from backup. Refer to object definitions of specific key
1661 types for KCV algorithms.

1662 Properties:

- 1663 1. For two keys that are cryptographically identical the value of this attribute should be identical.
- 1664 2. CKA_CHECK_VALUE should not be usable to obtain any part of the key value.
- 1665 3. Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely
1666 (the probability can easily be calculated) but possible.

1667 The attribute is optional, but if supported, regardless of how the key object is created or derived, the value
1668 of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is
1669 forbidden (i.e. when CKA_ENCRYPT is set to CK_FALSE).

1670 If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST
1671 match what the library calculates it to be or the library returns a CKR_ATTRIBUTE_VALUE_INVALID. If
1672 the library does not support the attribute then it should ignore it. Allowing the attribute in the template this
1673 way does no harm and allows the attribute to be treated like any other attribute for the purposes of key
1674 wrap and unwrap where the attributes are preserved also.

1675 The generation of the KCV may be prevented by the application supplying the attribute in the template as
1676 a no-value (0 length) entry. The application can query the value at any time like any other attribute using
1677 C_GetAttributeValue. C_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

1678 Unless otherwise specified for the object definition, the value of this attribute is derived from the key
1679 object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the
1680 default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

4.11 Domain parameter objects

4.11.1 Definitions

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

4.11.2 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 29, 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 <ul style="list-style-type: none">generated locally (<i>i.e.</i>, on the token) with a C_GenerateKeycreated with a C_CopyObject call as a copy of domain parameters which had its CKA_LOCAL attribute set to CK_TRUE

¹ Refer to Table 11 for footnotes

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

4.12 Mechanism objects

4.12.1 Definitions

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

4.12.2 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.

1709 Table 30, Common Mechanism Attributes

Attribute	Data Type	Meaning
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object

1710 The **CKA_MECHANISM_TYPE** attribute may not be set.

1711

1712 4.13 Profile objects

1713 4.13.1 Definitions

1714 This section defines the object class CKO_PROFILE for type CK_OBJECT_CLASS as used in the
1715 CKA_CLASS attribute of objects.

1716 4.13.2 Overview

1717 Profile objects (object class CKO_PROFILE) describe which PKCS #11 profiles the token implements.
1718 Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given
1719 token can contain more than one profile ID. The following table lists the attributes supported by profile
1720 objects, in addition to the common attributes defined for this object class:

1721 Table 31, Profile Object Attributes

Attribute	Data type	Meaning
CKA_PROFILE_ID	CK_PROFILE_ID	ID of the supported profile.

1722 The **CKA_PROFILE_ID** attribute identifies a profile that the token supports.

5 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)
- message-based encryption functions (5 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.

The Cryptoki API functions are presented in the following table:

Table 32, 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
	C_GetInterfaceList	obtains list of interfaces supported by Cryptoki library
	C_GetInterface	obtains interface specific entry points to 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

Category	Function	Description
	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_SessionCancel	terminates active session based operations
	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_LoginUser	logs into a token with explicit user name
	C_Logout	logs out from a token
Object management functions	C_CreateObject	creates an object
	C_CopyObject	creates a copy of an object
	C_DestroyObject	destroys an object
	C_GetObjectSize	obtains the size of an object in bytes
	C_GetAttributeValue	obtains an attribute value of an object
	C_SetAttributeValue	modifies an attribute value of an object
	C_FindObjectsInit	initializes an object search operation
	C_FindObjects	continues an object search operation
	C_FindObjectsFinal	finishes an object search operation
Encryption functions	C_EncryptInit	initializes an encryption operation
	C_Encrypt	encrypts single-part data
	C_EncryptUpdate	continues a multiple-part encryption operation
	C_EncryptFinal	finishes a multiple-part encryption operation
Message-based Encryption Functions	C_MessageEncryptInit	initializes a message-based encryption process
	C_EncryptMessage	encrypts a single-part message
	C_EncryptMessageBegin	begins a multiple-part message encryption operation
	C_EncryptMessageNext	continues or finishes a multiple-part message encryption operation
	C_MessageEncryptFinal	finishes a message-based encryption process
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-based Decryption Functions	C_MessageDecryptInit	initializes a message decryption operation
	C_DecryptMessage	decrypts single-part data
	C_DecryptMessageBegin	starts a multiple-part message decryption operation
	C_DecryptMessageNext	Continues and finishes a multiple-part message decryption operation

Category	Function	Description
	C_MessageDecryptFinal	finishes a message 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
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
Message-based Signature functions	C_MessageSignInit	initializes a message signature operation
	C_SignMessage	signs single-part data
	C_SignMessageBegin	starts a multiple-part message signature operation
	C_SignMessageNext	continues and finishes a multiple-part message signature operation
	C_MessageSignFinal	finishes a message signature operation
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
Message-based Functions for verifying signatures and MACs	C_MessageVerifyInit	initializes a message verification operation
	C_VerifyMessage	verifies single-part data
	C_VerifyMessageBegin	starts a multiple-part message verification operation
	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation
	C_MessageVerifyFinal	finishes a message verification operation
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	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair

Category	Function	Description
functions	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
Random number generation functions	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Parallel function management functions	C_GetFunctionStatus	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

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 3.6).

5.1 Function return values

The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1 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 5.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.

5.1.1 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 5.20) 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.

5.1.2 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 **[PKCS11-UG]**, 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**. An example of multiple threads accessing a common session simultaneously is where one thread is using a session when another thread closes that same session.

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.

5.1.3 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.

1828 • CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can
 1829 be returned by more than just the functions mentioned above; in particular, it is possible for
 1830 **C_GetSlotInfo** to return CKR_DEVICE_ERROR.

1831 • CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was*
 1832 *invoked*.

1833 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.

1834 The relative priorities of these errors are in the order listed above, e.g., if either of
 1835 CKR_DEVICE_MEMORY or CKR_DEVICE_ERROR would be an appropriate error return, then
 1836 CKR_DEVICE_MEMORY should be returned.

1837 In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction
 1838 between a token being removed *before* a function invocation and a token being removed *during* a
 1839 function execution.

1840 5.1.4 Special return value for application-supplied callbacks

1841 There is a special-purpose return value which is not returned by any function in the actual Cryptoki API,
 1842 but which may be returned by an application-supplied callback function. It is:

1843 • CKR_CANCEL: When a function executing in serial with an application decides to give the application
 1844 a chance to do some work, it calls an application-supplied function with a CKN_SURRENDER
 1845 callback (see Section 5.21). If the callback returns the value CKR_CANCEL, then the function aborts
 1846 and returns CKR_FUNCTION_CANCELED.

1847 5.1.5 Special return values for mutex-handling functions

1848 There are two other special-purpose return values which are not returned by any actual Cryptoki
 1849 functions. These values may be returned by application-supplied mutex-handling functions, and they may
 1850 safely be ignored by application developers who are not using their own threading model. They are:

1851 • CKR_MUTEX_BAD: This error code can be returned by mutex-handling functions that are passed a
 1852 bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a
 1853 bad mutex object. There is therefore no guarantee that such a function will successfully detect bad
 1854 mutex objects and return this value.

1855 • CKR_MUTEX_NOT_LOCKED: This error code can be returned by mutex-unlocking functions. It
 1856 indicates that the mutex supplied to the mutex-unlocking function was not locked.

1857 5.1.6 All other Cryptoki function return values

1858 Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions
 1859 of particular error codes, there are in general no particular priorities among the errors listed below, i.e., if
 1860 more than one error code might apply to an execution of a function, then the function may return any
 1861 applicable error code.

1862 • CKR_ACTION_PROHIBITED: This value can only be returned by C_CopyObject,
 1863 C_SetAttributeValue and C_DestroyObject. It denotes that the action may not be taken, either
 1864 because of underlying policy restrictions on the token, or because the object has the relevant
 1865 CKA_COPYABLE, CKA_MODIFIABLE or CKA_DESTROYABLE policy attribute set to CK_FALSE.

1866 • CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments
 1867 supplied to the Cryptoki function were in some way not appropriate.

1868 • CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not
 1869 be set by the application, or which may not be modified by the application. See Section 4.1 for more
 1870 information.

1871 • CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object
 1872 which cannot be satisfied because the object is either sensitive or un-extractable.

- 1873 • CKR_ATTRIBUTE_TYPE_INVALID: An invalid attribute type was specified in a template. See
1874 Section 4.1 for more information.
- 1875 • CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for a particular attribute in a
1876 template. See Section 4.1 for more information.
- 1877 • CKR_BUFFER_TOO_SMALL: The output of the function is too large to fit in the supplied buffer.
- 1878 • CKR_CANT_LOCK: This value can only be returned by **C_Initialize**. It means that the type of locking
1879 requested by the application for thread-safety is not available in this library, and so the application
1880 cannot make use of this library in the specified fashion.
- 1881 • CKR_CRYPTOKI_ALREADY_INITIALIZED: This value can only be returned by **C_Initialize**. It
1882 means that the Cryptoki library has already been initialized (by a previous call to **C_Initialize** which
1883 did not have a matching **C_Finalize** call).
- 1884 • CKR_CRYPTOKI_NOT_INITIALIZED: This value can be returned by any function other than
1885 **C_Initialize**, **C_GetFunctionList**, **C_GetInterfaceList** and **C_GetInterface**. It indicates that the
1886 function cannot be executed because the Cryptoki library has not yet been initialized by a call to
1887 **C_Initialize**.
- 1888 • CKR_CURVE_NOT_SUPPORTED: This curve is not supported by this token. Used with Elliptic
1889 Curve mechanisms.
- 1890 • CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This return
1891 value has lower priority than CKR_DATA_LEN_RANGE.
- 1892 • CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length.
1893 Depending on the operation's mechanism, this could mean that the plaintext data is too short, too
1894 long, or is not a multiple of some particular block size. This return value has higher priority than
1895 CKR_DATA_INVALID.
- 1896 • CKR_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the
1897 function. Which representation methods of domain parameters are supported by a given mechanism
1898 can vary from token to token.
- 1899 • CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been
1900 determined to be invalid ciphertext. This return value has lower priority than
1901 CKR_ENCRYPTED_DATA_LEN_RANGE.
- 1902 • CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been
1903 determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's
1904 mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some
1905 particular block size. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.
- 1906 • CKR_EXCEEDED_MAX_ITERATIONS: An iterative algorithm (for key pair generation, domain
1907 parameter generation etc.) failed because we have exceeded the maximum number of iterations.
1908 This error code has precedence over CKR_FUNCTION_FAILED. Examples of iterative algorithms
1909 include DSA signature generation (retry if either $r = 0$ or $s = 0$) and generation of DSA primes p and q
1910 specified in FIPS 186-4.
- 1911 • CKR_FIPS_SELF_TEST_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed.
1912 The token entered an error state. Future calls to cryptographic functions on the token will return
1913 CKR_GENERAL_ERROR. CKR_FIPS_SELF_TEST_FAILED has a higher precedence over
1914 CKR_GENERAL_ERROR. This error may be returned by **C_Initialize**, if a power-up self-test failed,
1915 by **C_GenerateRandom** or **C_SeedRandom**, if the continuous random number generator test failed,
1916 or by **C_GenerateKeyPair**, if the pair-wise consistency test failed.
- 1917 • CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This happens to a
1918 cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns
1919 CKR_CANCEL (see CKR_CANCEL). It also happens to a function that performs PIN entry through a
1920 protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- 1921 • CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the
1922 specified session. This is a legacy error code which is only returned by the legacy functions
1923 **C_GetFunctionStatus** and **C_CancelFunction**.

- 1924 • **CKR_FUNCTION_NOT_SUPPORTED**: The requested function is not supported by this Cryptoki
1925 library. Even unsupported functions in the Cryptoki API should have a “stub” in the library; this stub
1926 should simply return the value **CKR_FUNCTION_NOT_SUPPORTED**.
- 1927 • **CKR_FUNCTION_REJECTED**: The signature request is rejected by the user.
- 1928 • **CKR_INFORMATION_SENSITIVE**: The information requested could not be obtained because the
1929 token considers it sensitive, and is not able or willing to reveal it.
- 1930 • **CKR_KEY_CHANGED**: This value is only returned by **C_SetOperationState**. It indicates that one of
1931 the keys specified is not the same key that was being used in the original saved session.
- 1932 • **CKR_KEY_FUNCTION_NOT_PERMITTED**: An attempt has been made to use a key for a
1933 cryptographic purpose that the key’s attributes are not set to allow it to do. For example, to use a key
1934 for performing encryption, that key **MUST** have its **CKA_ENCRYPT** attribute set to **CK_TRUE** (the
1935 fact that the key **MUST** have a **CKA_ENCRYPT** attribute implies that the key cannot be a private
1936 key). This return value has lower priority than **CKR_KEY_TYPE_INCONSISTENT**.
- 1937 • **CKR_KEY_HANDLE_INVALID**: The specified key handle is not valid. It may be the case that the
1938 specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a
1939 valid key handle.
- 1940 • **CKR_KEY_INDIGESTIBLE**: This error code can only be returned by **C_DigestKey**. It indicates that
1941 the value of the specified key cannot be digested for some reason (perhaps the key isn’t a secret key,
1942 or perhaps the token simply can’t digest this kind of key).
- 1943 • **CKR_KEY_NEEDED**: This value is only returned by **C_SetOperationState**. It indicates that the
1944 session state cannot be restored because **C_SetOperationState** needs to be supplied with one or
1945 more keys that were being used in the original saved session.
- 1946 • **CKR_KEY_NOT_NEEDED**: An extraneous key was supplied to **C_SetOperationState**. For
1947 example, an attempt was made to restore a session that had been performing a message digesting
1948 operation, and an encryption key was supplied.
- 1949 • **CKR_KEY_NOT_WRAPPABLE**: Although the specified private or secret key does not have its
1950 **CKA_EXTRACTABLE** attribute set to **CK_FALSE**, Cryptoki (or the token) is unable to wrap the key as
1951 requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping
1952 key specified is not one of these types). Compare with **CKR_KEY_UNEXTRACTABLE**.
- 1953 • **CKR_KEY_SIZE_RANGE**: Although the requested keyed cryptographic operation could in principle
1954 be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key’s
1955 size is outside the range of key sizes that it can handle.
- 1956 • **CKR_KEY_TYPE_INCONSISTENT**: The specified key is not the correct type of key to use with the
1957 specified mechanism. This return value has a higher priority than
1958 **CKR_KEY_FUNCTION_NOT_PERMITTED**.
- 1959 • **CKR_KEY_UNEXTRACTABLE**: The specified private or secret key can’t be wrapped because its
1960 **CKA_EXTRACTABLE** attribute is set to **CK_FALSE**. Compare with **CKR_KEY_NOT_WRAPPABLE**.
- 1961 • **CKR_LIBRARY_LOAD_FAILED**: The Cryptoki library could not load a dependent shared library.
- 1962 • **CKR_MECHANISM_INVALID**: An invalid mechanism was specified to the cryptographic operation.
1963 This error code is an appropriate return value if an unknown mechanism was specified or if the
1964 mechanism specified cannot be used in the selected token with the selected function.
- 1965 • **CKR_MECHANISM_PARAM_INVALID**: Invalid parameters were supplied to the mechanism specified
1966 to the cryptographic operation. Which parameter values are supported by a given mechanism can
1967 vary from token to token.
- 1968 • **CKR_NEED_TO_CREATE_THREADS**: This value can only be returned by **C_Initialize**. It is
1969 returned when two conditions hold:
 - 1970 1. The application called **C_Initialize** in a way which tells the Cryptoki library that application
1971 threads executing calls to the library cannot use native operating system methods to spawn new
1972 threads.

- 1973 2. The library cannot function properly without being able to spawn new threads in the above
1974 fashion.
- 1975 • **CKR_NO_EVENT**: This value can only be returned by **C_WaitForSlotEvent**. It is returned when
1976 **C_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
- 1977 • **CKR_OBJECT_HANDLE_INVALID**: The specified object handle is not valid. We reiterate here that 0
1978 is never a valid object handle.
- 1979 • **CKR_OPERATION_ACTIVE**: There is already an active operation (or combination of active
1980 operations) which prevents Cryptoki from activating the specified operation. For example, an active
1981 object-searching operation would prevent Cryptoki from activating an encryption operation with
1982 **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent
1983 Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous
1984 dual cryptographic operations in a session (see the description of the
1985 **CKF_DUAL_CRYPTO_OPERATIONS** flag in the **CK_TOKEN_INFO** structure), an active signature
1986 operation would prevent Cryptoki from activating an encryption operation.
- 1987 • **CKR_OPERATION_NOT_INITIALIZED**: There is no active operation of an appropriate type in the
1988 specified session. For example, an application cannot call **C_Encrypt** in a session without having
1989 called **C_EncryptInit** first to activate an encryption operation.
- 1990 • **CKR_PIN_EXPIRED**: The specified PIN has expired, and the requested operation cannot be carried
1991 out unless **C_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a
1992 token ever expires varies from token to token.
- 1993 • **CKR_PIN_INCORRECT**: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the
1994 token. More generally-- when authentication to the token involves something other than a PIN-- the
1995 attempt to authenticate the user has failed.
- 1996 • **CKR_PIN_INVALID**: The specified PIN has invalid characters in it. This return code only applies to
1997 functions which attempt to set a PIN.
- 1998 • **CKR_PIN_LEN_RANGE**: The specified PIN is too long or too short. This return code only applies to
1999 functions which attempt to set a PIN.
- 2000 • **CKR_PIN_LOCKED**: The specified PIN is "locked", and cannot be used. That is, because some
2001 particular number of failed authentication attempts has been reached, the token is unwilling to permit
2002 further attempts at authentication. Depending on the token, the specified PIN may or may not remain
2003 locked indefinitely.
- 2004 • **CKR_PIN_TOO_WEAK**: The specified PIN is too weak so that it could be easy to guess. If the PIN is
2005 too short, **CKR_PIN_LEN_RANGE** should be returned instead. This return code only applies to
2006 functions which attempt to set a PIN.
- 2007 • **CKR_PUBLIC_KEY_INVALID**: The public key fails a public key validation. For example, an EC
2008 public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may
2009 be returned by **C_CreateObject**, when the public key is created, or by **C_VerifyInit** or
2010 **C_VerifyRecoverInit**, when the public key is used. It may also be returned by **C_DeriveKey**, in
2011 preference to **CKR_MECHANISM_PARAM_INVALID**, if the other party's public key specified in the
2012 mechanism's parameters is invalid.
- 2013 • **CKR_RANDOM_NO_RNG**: This value can be returned by **C_SeedRandom** and
2014 **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.
2015 This return value has higher priority than **CKR_RANDOM_SEED_NOT_SUPPORTED**.
- 2016 • **CKR_RANDOM_SEED_NOT_SUPPORTED**: This value can only be returned by **C_SeedRandom**.
2017 It indicates that the token's random number generator does not accept seeding from an application.
2018 This return value has lower priority than **CKR_RANDOM_NO_RNG**.
- 2019 • **CKR_SAVED_STATE_INVALID**: This value can only be returned by **C_SetOperationState**. It
2020 indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be
2021 restored to the specified session.

- 2022 • CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that the
2023 attempt to open a session failed, either because the token has too many sessions already open, or
2024 because the token has too many read/write sessions already open.
- 2025 • CKR_SESSION_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a
2026 session with the token is already open, and so the token cannot be initialized.
- 2027 • CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel
2028 sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel
2029 sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by
2030 **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated]
2031 way.
- 2032 • CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action
2033 because it is a read-only session. This return value has lower priority than
2034 CKR_TOKEN_WRITE_PROTECTED.
- 2035 • CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot
2036 be logged in.
- 2037 • CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a
2038 read-only session cannot be opened.
- 2039 • CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on
2040 the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- 2041 • CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower
2042 priority than CKR_SIGNATURE_LEN_RANGE.
- 2043 • CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- 2044 • CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be
2045 saved for some reason (possibly the token is simply unable to save the current state). This return
2046 value has lower priority than CKR_OPERATION_NOT_INITIALIZED.
- 2047 • CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and
2048 lacks some necessary attributes. See Section 4.1 for more information.
- 2049 • CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting
2050 attributes. See Section 4.1 for more information.
- 2051 • CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in
2052 the slot.
- 2053 • CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the
2054 token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.
- 2055 • CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_UnwrapKey**.
2056 It indicates that the key handle specified to be used to unwrap another key is not valid.
- 2057 • CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It
2058 indicates that although the requested unwrapping operation could in principle be carried out, this
2059 Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the
2060 range of key sizes that it can handle.
- 2061 • CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by
2062 **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent
2063 with the mechanism specified for unwrapping.
- 2064 • CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that
2065 the specified user cannot be logged into the session, because it is already logged into the session.
2066 For example, if an application has an open SO session, and it attempts to log the SO into it, it will
2067 receive this error code.
- 2068 • CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It
2069 indicates that the specified user cannot be logged into the session, because another user is already

2070 logged into the session. For example, if an application has an open SO session, and it attempts to
 2071 log the normal user into it, it will receive this error code.

- 2072 • **CKR_USER_NOT_LOGGED_IN**: The desired action cannot be performed because the appropriate
 2073 user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out
 2074 unless it is logged in. Another example is that a private object cannot be created on a token unless
 2075 the session attempting to create it is logged in as the normal user. A final example is that
 2076 cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 2077 • **CKR_USER_PIN_NOT_INITIALIZED**: This value can only be returned by **C_Login**. It indicates that
 2078 the normal user's PIN has not yet been initialized with **C_InitPIN**.
- 2079 • **CKR_USER_TOO_MANY_TYPES**: An attempt was made to have more distinct users simultaneously
 2080 logged into the token than the token and/or library permits. For example, if some application has an
 2081 open SO session, and another application attempts to log the normal user into a session, the attempt
 2082 may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be
 2083 supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-
 2084 user tokens.
- 2085 • **CKR_USER_TYPE_INVALID**: An invalid value was specified as a **CK_USER_TYPE**. Valid types are
 2086 **CKU_SO**, **CKU_USER**, and **CKU_CONTEXT_SPECIFIC**.
- 2087 • **CKR_WRAPPED_KEY_INVALID**: This value can only be returned by **C_UnwrapKey**. It indicates
 2088 that the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular
 2089 type of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**),
 2090 and the wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper
 2091 type, then **C_UnwrapKey** should return **CKR_WRAPPED_KEY_INVALID**. This return value has
 2092 lower priority than **CKR_WRAPPED_KEY_LEN_RANGE**.
- 2093 • **CKR_WRAPPED_KEY_LEN_RANGE**: This value can only be returned by **C_UnwrapKey**. It
 2094 indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
 2095 This return value has higher priority than **CKR_WRAPPED_KEY_INVALID**.
- 2096 • **CKR_WRAPPING_KEY_HANDLE_INVALID**: This value can only be returned by **C_WrapKey**. It
 2097 indicates that the key handle specified to be used to wrap another key is not valid.
- 2098 • **CKR_WRAPPING_KEY_SIZE_RANGE**: This value can only be returned by **C_WrapKey**. It indicates
 2099 that although the requested wrapping operation could in principle be carried out, this Cryptoki library
 2100 (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range
 2101 of key sizes that it can handle.
- 2102 • **CKR_WRAPPING_KEY_TYPE_INCONSISTENT**: This value can only be returned by **C_WrapKey**. It
 2103 indicates that the type of the key specified to wrap another key is not consistent with the mechanism
 2104 specified for wrapping.
- 2105 • **CKR_OPERATION_CANCEL_FAILED**: This value can only be returned by **C_SessionCancel**. It
 2106 means that one or more of the requested operations could not be cancelled for implementation or
 2107 vendor-specific reasons.

2108 5.1.7 More on relative priorities of Cryptoki errors

2109 In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than **CKR_OK**) take
 2110 precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
 2111 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that
 2112 functions that use a session handle (*i.e.*, *most* functions!) never return the error code
 2113 **CKR_TOKEN_NOT_PRESENT** (they return **CKR_SESSION_HANDLE_INVALID** instead). Other than
 2114 these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
 2115 applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
 2116 descriptions of functions.

5.1.8 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 5.1.2 and 5.1.3, 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 5.1.2, 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 4.1, 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.

5.2 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 block size 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).

5.3 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.

5.4 General-purpose functions

Cryptoki provides the following general-purpose functions:

5.4.1 C_Initialize

```
CK_DECLARE_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`.

2214 A call to **C_Initialize** with *pInitArgs* set to `NULL_PTR` is treated like a call to **C_Initialize** with *pInitArgs*
2215 pointing to a **CK_C_INITIALIZE_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*,
2216 *UnlockMutex*, and *pReserved* fields set to `NULL_PTR`, and has the *flags* field set to 0.

2217 **C_Initialize** should be the first Cryptoki call made by an application, except for calls to
2218 **C_GetFunctionList**, **C_GetInterfaceList**, or **C_GetInterface**. What this function actually does is
2219 implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or
2220 any other resources it requires.

2221 If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize**
2222 should (eventually) be succeeded by a single call to **C_Finalize**. See [\[PKCS11-UG\]](#) for further details.

2223 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CANT_LOCK`,
2224 `CKR_CRYPTOKI_ALREADY_INITIALIZED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`,
2225 `CKR_HOST_MEMORY`, `CKR_NEED_TO_CREATE_THREADS`, `CKR_OK`.

2226 Example: see **C_GetInfo**.

2227 5.4.2 C_Finalize

```
2228 CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (  
2229     CK_VOID_PTR pReserved  
2230 );
```

2231 **C_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the
2232 last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for
2233 this version, it should be set to `NULL_PTR` (if **C_Finalize** is called with a non-`NULL_PTR` value for
2234 *pReserved*, it should return the value `CKR_ARGUMENTS_BAD`).

2235 If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to
2236 **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application
2237 can make calls to other Cryptoki functions. See [\[PKCS11-UG\]](#) for further details.

2238 *Despite the fact that the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded*
2239 *access to a Cryptoki library, the behavior of **C_Finalize** is nevertheless undefined if it is called by an*
2240 *application while other threads of the application are making Cryptoki calls. The exception to this*
2241 *exceptional behavior of **C_Finalize** occurs when a thread calls **C_Finalize** while another of the*
2242 *application's threads is blocking on Cryptoki's **C_WaitForSlotEvent** function. When this happens, the*
2243 *blocked thread becomes unblocked and returns the value `CKR_CRYPTOKI_NOT_INITIALIZED`. See*
2244 ***C_WaitForSlotEvent** for more information.*

2245 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
2246 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

2247 Example: see **C_GetInfo**.

2248 5.4.3 C_GetInfo

```
2249 CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (  
2250     CK_INFO_PTR pInfo  
2251 );
```

2252 **C_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the
2253 information.

2254 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
2255 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

2256 Example:

```
2257 CK_INFO info;  
2258 CK_RV rv;  
2259 CK_C_INITIALIZE_ARGS InitArgs;
```

```

2260
2261 InitArgs.CreateMutex = &MyCreateMutex;
2262 InitArgs.DestroyMutex = &MyDestroyMutex;
2263 InitArgs.LockMutex = &MyLockMutex;
2264 InitArgs.UnlockMutex = &MyUnlockMutex;
2265 InitArgs.flags = CKF_OS_LOCKING_OK;
2266 InitArgs.pReserved = NULL_PTR;
2267
2268 rv = C_Initialize((CK_VOID_PTR)&InitArgs);
2269 assert(rv == CKR_OK);
2270
2271 rv = C_GetInfo(&info);
2272 assert(rv == CKR_OK);
2273 if(info.cryptokiVersion.major == 2) {
2274     /* Do lots of interesting cryptographic things with the token */
2275     .
2276     .
2277 }
2278
2279 rv = C_Finalize(NULL_PTR);
2280 assert(rv == CKR_OK);

```

2281 5.4.4 C_GetFunctionList

```

2282 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList)(
2283     CK_FUNCTION_LIST_PTR_PTR ppFunctionList
2284 );

```

2285 **C_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*
2286 points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which in turn
2287 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*
2288 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.*
2289 Whether or not this is the case, no attempt should be made to write to this memory.

2290 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2291 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2292 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2293 Return values: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2294 CKR_HOST_MEMORY, CKR_OK.

2295 Example:

```

2296 CK_FUNCTION_LIST_PTR pFunctionList;
2297 CK_C_Initialize pC_Initialize;
2298 CK_RV rv;
2299
2300 /* It's OK to call C_GetFunctionList before calling C_Initialize */
2301 rv = C_GetFunctionList(&pFunctionList);
2302 assert(rv == CKR_OK);

```



```

2303 pC_Initialize = pFunctionList -> C_Initialize;
2304
2305 /* Call the C_Initialize function in the library */
2306 rv = (*pC_Initialize)(NULL_PTR);

```

2307 5.4.5 C_GetInterfaceList

```

2308 CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList)(
2309     CK_INTERFACE_PTR    pInterfaceList,
2310     CK_ULONG_PTR        pulCount
2311 );

```

2312 **C_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points
2313 to the location that receives the number of interfaces.

2314 There are two ways for an application to call **C_GetInterfaceList**:

- 2315 1. If *pInterfaceList* is `NULL_PTR`, then all that **C_GetInterfaceList** does is return (in **pulCount*) the
2316 number of interfaces, without actually returning a list of interfaces. The contents of **pulCount* on
2317 entry to **C_GetInterfaceList** has no meaning in this case, and the call returns the value `CKR_OK`.
- 2318 2. If *pInterfaceList* is not `NULL_PTR`, then **pulCount* MUST contain the size (in terms of
2319 **CK_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to
2320 hold the list of interfaces, then the list is returned in it, and `CKR_OK` is returned. If not, then the call
2321 to **C_GetInterfaceList** returns the value `CKR_BUFFER_TOO_SMALL`. In either case, the value
2322 **pulCount* is set to hold the number of interfaces.

2323 Because **C_GetInterfaceList** does not allocate any space of its own, an application will often call
2324 **C_GetInterfaceList** twice. However, this behavior is by no means required.

2325 **C_GetInterfaceList** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
2326 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
2327 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
2328 made to write to this memory. The same caveat applies to the interface names returned.

2329
2330 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2331 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2332 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2333 Return values: `CKR_BUFFER_TOO_SMALL`, `CKR_ARGUMENTS_BAD`, `CKR_FUNCTION_FAILED`,
2334 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

2335 Example:

```

2336 CK_ULONG ulCount=0;
2337 CK_INTERFACE_PTR interfaceList=NULL;
2338 CK_RV rv;
2339 int I;
2340
2341 /* get number of interfaces */
2342 rv = C_GetInterfaceList(NULL,&ulCount);
2343 if (rv == CKR_OK) {
2344     /* get copy of interfaces */
2345     interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
2346     rv = C_GetInterfaceList(interfaceList,&ulCount);
2347     for(i=0;i<ulCount;i++) {

```

```

2348     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2349           interfaceList[i].pInterfaceName,
2350           ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
2351           ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
2352           interfaceList[i].pFunctionList,
2353           interfaceList[i].flags);
2354 }
2355 }
2356

```

2357 5.4.6 C_GetInterface

```

2358 CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) (
2359     CK_UTF8CHAR_PTR      pInterfaceName,
2360     CK_VERSION_PTR       pVersion,
2361     CK_INTERFACE_PTR_PTR ppInterface,
2362     CK_FLAGS              flags
2363 );

```

2364 **C_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies
 2365 the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that
 2366 receives the interface, *flags* specifies the required interface flags.

2367 There are multiple ways for an application to specify a particular interface when calling **C_GetInterface**:

- 2368 1. If *pInterfaceName* is not NULL_PTR, the name of the interface returned must match. If
 2369 *pInterfaceName* is NULL_PTR, the cryptoki library can return a default interface of its choice
- 2370 2. If *pVersion* is not NULL_PTR, the version of the interface returned must match. If *pVersion* is
 2371 NULL_PTR, the cryptoki library can return an interface of any version
- 2372 3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include
 2373 additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

2374 **C_GetInterface** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
 2375 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
 2376 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
 2377 made to write to this memory. The same caveat applies to the interface names returned.

2378 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
 2379 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
 2380 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2381 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,
 2382 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2383 Example:

```

2384 CK_INTERFACE_PTR interface;
2385 CK_RV rv;
2386 CK_VERSION version;
2387 CK_FLAGS flags=CKF_ INTERFACE_FORK_SAFE;
2388
2389 /* get default interface */
2390 rv = C_GetInterface(NULL,NULL,&interface,flags);
2391 if (rv == CKR_OK) {

```

```

2392     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2393           interface->pInterfaceName,
2394           ((CK_VERSION *)interface->pFunctionList)->major,
2395           ((CK_VERSION *)interface->pFunctionList)->minor,
2396           interface->pFunctionList,
2397           interface->flags);
2398 }
2399
2400 /* get default standard interface */
2401 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
2402 if (rv == CKR_OK) {
2403     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2404           interface->pInterfaceName,
2405           ((CK_VERSION *)interface->pFunctionList)->major,
2406           ((CK_VERSION *)interface->pFunctionList)->minor,
2407           interface->pFunctionList,
2408           interface->flags);
2409 }
2410
2411 /* get specific standard version interface */
2412 version.major=3;
2413 version.minor=0;
2414 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
2415 if (rv == CKR_OK) {
2416     CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
2417
2418     /* ... use the new functions */
2419     pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
2420                        pUsername,ulUsernameLen);
2421 }
2422
2423 /* get specific vendor version interface */
2424 version.major=1;
2425 version.minor=0;
2426 rv = C_GetInterface((CK_UTF8CHAR_PTR)
2427                     "Vendor VendorName",&version,&interface,flags);
2428 if (rv == CKR_OK) {
2429     CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
2430
2431     /* ... use vendor specific functions */
2432     pkcs11->C_VendorFunction1(param1,param2,param3);
2433 }
2434

```


5.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management:

5.5.1 C_GetSlotList

```
CK_DECLARE_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 =
```

```

2481     (CK_SLOT_ID_PTR) malloc (ulSlotCount*sizeof(CK_SLOT_ID));
2482     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
2483     if (rv == CKR_OK) {
2484         /* Now use that list of all slots */
2485         .
2486         .
2487     }
2488
2489     free(pSlotList);
2490 }
2491
2492 /* Get list of all slots with a token present */
2493 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
2494 ulSlotWithTokenCount = 0;
2495 while (1) {
2496     rv = C_GetSlotList(
2497         CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
2498     if (rv != CKR_BUFFER_TOO_SMALL)
2499         break;
2500     pSlotWithTokenList = realloc(
2501         pSlotWithTokenList,
2502         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
2503 }
2504
2505 if (rv == CKR_OK) {
2506     /* Now use that list of all slots with a token present */
2507     .
2508     .
2509 }
2510
2511 free(pSlotWithTokenList);

```

2512 5.5.2 C_GetSlotInfo

```

2513 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo) (
2514     CK_SLOT_ID slotID,
2515     CK_SLOT_INFO_PTR pInfo
2516 );

```

2517 **C_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo*
2518 points to the location that receives the slot information.

2519 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2520 CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
2521 CKR_OK, CKR_SLOT_ID_INVALID.

2522 Example: see **C_GetTokenInfo**.

5.5.3 C_GetTokenInfo

```
CK_DECLARE_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);
}
```

5.5.4 C_WaitForSlotEvent

```
CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent)(
    CK_FLAGS flags,
    CK_SLOT_ID_PTR pSlot,
```

```

2565         CK_VOID_PTR pReserved
2566     );

```

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:

```

2595     CK_FLAGS flags = 0;
2596     CK_SLOT_ID slotID;
2597     CK_SLOT_INFO slotInfo;
2598     CK_RV rv;
2599     .
2600     .
2601     /* Block and wait for a slot event */
2602     rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);
2603     assert(rv == CKR_OK);
2604
2605     /* See what's up with that slot */
2606     rv = C_GetSlotInfo(slotID, &slotInfo);
2607     assert(rv == CKR_OK);
2608

```

2609 5.5.5 C_GetMechanismList

```

2610 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList) (

```

```

2611     CK_SLOT_ID slotID,
2612     CK_MECHANISM_TYPE_PTR pMechanismList,
2613     CK_ULONG_PTR pulCount
2614 );

```

2615 **C_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID
2616 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

2617 There are two ways for an application to call **C_GetMechanismList**:

- 2618 1. If *pMechanismList* is **NULL_PTR**, then all that **C_GetMechanismList** does is return (in **pulCount*)
2619 the number of mechanisms, without actually returning a list of mechanisms. The contents of
2620 **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the
2621 value **CKR_OK**.
- 2622 2. If *pMechanismList* is not **NULL_PTR**, then **pulCount* MUST contain the size (in terms of
2623 **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large
2624 enough to hold the list of mechanisms, then the list is returned in it, and **CKR_OK** is returned. If not,
2625 then the call to **C_GetMechanismList** returns the value **CKR_BUFFER_TOO_SMALL**. In either
2626 case, the value **pulCount* is set to hold the number of mechanisms.

2627 Because **C_GetMechanismList** does not allocate any space of its own, an application will often call
2628 **C_GetMechanismList** twice. However, this behavior is by no means required.

2629 Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
2630 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
2631 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
2632 **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
2633 **CKR_ARGUMENTS_BAD**.

2634 Example:

```

2635 CK_SLOT_ID slotID;
2636 CK_ULONG ulCount;
2637 CK_MECHANISM_TYPE_PTR pMechanismList;
2638 CK_RV rv;
2639
2640 .
2641 .
2642 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);
2643 if ((rv == CKR_OK) && (ulCount > 0)) {
2644     pMechanismList =
2645         (CK_MECHANISM_TYPE_PTR)
2646         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));
2647     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);
2648     if (rv == CKR_OK) {
2649         .
2650         .
2651     }
2652     free(pMechanismList);
2653 }

```

2654 5.5.6 C_GetMechanismInfo

```

2655 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (

```

```

2656     CK_SLOT_ID slotID,
2657     CK_MECHANISM_TYPE type,
2658     CK_MECHANISM_INFO_PTR pInfo
2659 );

```

2660 **C_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token.
2661 *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives
2662 the mechanism information.

2663 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2664 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2665 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID,
2666 CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2667 **Example:**

```

2668 CK_SLOT_ID slotID;
2669 CK_MECHANISM_INFO info;
2670 CK_RV rv;
2671
2672 .
2673 .
2674 /* Get information about the CKM_MD2 mechanism for this token */
2675 rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
2676 if (rv == CKR_OK) {
2677     if (info.flags & CKF_DIGEST) {
2678         .
2679         .
2680     }
2681 }

```

2682 5.5.7 C_InitToken

```

2683 CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
2684     CK_SLOT_ID slotID,
2685     CK_UTF8CHAR_PTR pPin,
2686     CK_ULONG ulPinLen,
2687     CK_UTF8CHAR_PTR pLabel
2688 );

```

2689 **C_InitToken** initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN
2690 (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-
2691 byte label of the token (which **MUST** be padded with blank characters, and which **MUST not** be null-
2692 terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may
2693 impose subset restrictions.

2694 If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the
2695 initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the
2696 existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after
2697 the function completes successfully. If the SO PIN is lost, then the card **MUST** be reinitialized using a
2698 mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the
2699 **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token
2700 will be reinitialized, and the client **MUST** supply the existing SO password in *pPin*.

2701 When a token is initialized, all objects that can be destroyed are destroyed (*i.e.*, all except for
 2702 “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled
 2703 until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created,
 2704 and attributes of some objects may be set to default values.

2705 If the token has a “protected authentication path”, as indicated by the
 2706 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
 2707 that there is some way for a user to be authenticated to the token without having the application send a
 2708 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
 2709 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*
 2710 parameter to **C_InitToken** should be **NULL_PTR**. During the execution of **C_InitToken**, the SO’s PIN will
 2711 be entered through the protected authentication path.

2712 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether
 2713 or not **C_InitToken** can be used to initialize the token.

2714 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
 2715 call to **C_InitToken** is made under such circumstances, the call fails with error **CKR_SESSION_EXISTS**.
 2716 Unfortunately, it may happen when **C_InitToken** is called that some other application *does* have an open
 2717 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
 2718 applications using the token. If this is the case, then the consequences of the **C_InitToken** call are
 2719 undefined.

2720 The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,
 2721 an initialization mechanism outside the scope of Cryptoki **MUST** be employed. The definition of “complex
 2722 token” is product specific.

2723 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
 2724 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
 2725 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_PIN_INCORRECT**,
 2726 **CKR_PIN_LOCKED**, **CKR_SESSION_EXISTS**, **CKR_SLOT_ID_INVALID**,
 2727 **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
 2728 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

2729 Example:

```
2730 CK_SLOT_ID slotID;
2731 CK_UTF8CHAR pin[] = {"MyPIN"};
2732 CK_UTF8CHAR label[32];
2733 CK_RV rv;
2734
2735 .
2736 .
2737 memset(label, ' ', sizeof(label));
2738 memcpy(label, "My first token", strlen("My first token"));
2739 rv = C_InitToken(slotID, pin, strlen(pin), label);
2740 if (rv == CKR_OK) {
2741     .
2742     .
2743 }
```

2744 5.5.8 C_InitPIN

```
2745 CK_DECLARE_FUNCTION(CK_RV, C_InitPIN)(
2746     CK_SESSION_HANDLE hSession,
2747     CK_UTF8CHAR_PTR pPin,
```



```

2748     CK_ULONG ulPinLen
2749 );

```

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 to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad 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 PIN pad, 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:

```

2771 CK_SESSION_HANDLE hSession;
2772 CK_UTF8CHAR newPin[] = {"NewPIN"};
2773 CK_RV rv;
2774
2775 rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);
2776 if (rv == CKR_OK) {
2777     .
2778     .
2779 }

```

2780 5.5.9 C_SetPIN

```

2781 CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (
2782     CK_SESSION_HANDLE hSession,
2783     CK_UTF8CHAR_PTR pOldPin,
2784     CK_ULONG ulOldLen,
2785     CK_UTF8CHAR_PTR pNewPin,
2786     CK_ULONG ulNewLen
2787 );

```

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.

2796 If the token has a “protected authentication path”, as indicated by the
 2797 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
 2798 that there is some way for a user to be authenticated to the token without having to send a PIN through
 2799 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
 2800 on the slot device. To modify the current user’s PIN on a token with such a protected authentication path,
 2801 the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be NULL_PTR. During the execution of
 2802 **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
 2803 path. It is not specified how the PIN pad should be used to enter *two* PINs; this varies.

2804 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
 2805 or not **C_SetPIN** can be used to modify the current user’s PIN.

2806 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 2807 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 2808 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT,
 2809 CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
 2810 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
 2811 CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

2812 Example:

```
2813 CK_SESSION_HANDLE hSession;
2814 CK_UTF8CHAR oldPin[] = {"OldPIN"};
2815 CK_UTF8CHAR newPin[] = {"NewPIN"};
2816 CK_RV rv;
2817
2818 rv = C_SetPIN(
2819     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);
2820 if (rv == CKR_OK) {
2821     .
2822     .
2823 }
```

2824 5.6 Session management functions

2825 A typical application might perform the following series of steps to make use of a token (note that there
 2826 are other reasonable sequences of events that an application might perform):

- 2827 1. Select a token.
- 2828 2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
- 2829 3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a
 2830 shared login state, **C_Login** only needs to be called for one of the sessions.
- 2831 4. Perform cryptographic operations using the sessions with the token.
- 2832 5. Call **C_CloseSession** once for each session that the application has with the token, or call
 2833 **C_CloseAllSessions** to close all the application’s sessions simultaneously.

2834 As has been observed, an application may have concurrent sessions with more than one token. It is also
 2835 possible for a token to have concurrent sessions with more than one application.

2836 Cryptoki provides the following functions for session management:

2837 5.6.1 C_OpenSession

```
2838 CK_DECLARE_FUNCTION(CK_RV, C_OpenSession)(
2839     CK_SLOT_ID slotID,
2840     CK_FLAGS flags,
2841     CK_VOID_PTR pApplication,
```

```

2842     CK_NOTIFY Notify,
2843     CK_SESSION_HANDLE_PTR phSession
2844 );

```

2845 **C_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the
 2846 slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to
 2847 the notification callback; *Notify* is the address of the notification callback function (see Section 5.21);
 2848 *phSession* points to the location that receives the handle for the new session.

2849 When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or
 2850 more bit flags defined in the **CK_SESSION_INFO** data type. For legacy reasons, the
 2851 **CKF_SERIAL_SESSION** bit MUST always be set; if a call to **C_OpenSession** does not have this bit set,
 2852 the call should return unsuccessfully with the error code
 2853 **CKR_SESSION_PARALLEL_NOT_SUPPORTED**.

2854 There may be a limit on the number of concurrent sessions an application may have with the token, which
 2855 may depend on whether the session is “read-only” or “read/write”. An attempt to open a session which
 2856 does not succeed because there are too many existing sessions of some type should return
 2857 **CKR_SESSION_COUNT**.

2858 If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only
 2859 sessions may be opened with it.

2860 If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any
 2861 attempt to open a R/O session with the token fails with error code
 2862 **CKR_SESSION_READ_WRITE_SO_EXISTS** (see [PKCS11-UG] for further details).

2863 The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the
 2864 application does not wish to support callbacks, it should pass a value of **NULL_PTR** as the *Notify*
 2865 parameter. See Section 5.21 for more information about application callbacks.

2866 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
 2867 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**,
 2868 **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_SESSION_COUNT**,
 2869 **CKR_SESSION_PARALLEL_NOT_SUPPORTED**, **CKR_SESSION_READ_WRITE_SO_EXISTS**,
 2870 **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
 2871 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

2872 Example: see **C_CloseSession**.

2873 5.6.2 C_CloseSession

```

2874 CK_DECLARE_FUNCTION(CK_RV, C_CloseSession)(
2875     CK_SESSION_HANDLE hSession
2876 );

```

2877 **C_CloseSession** closes a session between an application and a token. *hSession* is the session's
 2878 handle.

2879 When a session is closed, all session objects created by the session are destroyed automatically, even if
 2880 the application has other sessions “using” the objects (see [PKCS11-UG] for further details).

2881 If this function is successful and it closes the last session between the application and the token, the login
 2882 state of the token for the application returns to public sessions. Any new sessions to the token opened by
 2883 the application will be either R/O Public or R/W Public sessions.

2884 Depending on the token, when the last open session any application has with the token is closed, the
 2885 token may be “ejected” from its reader (if this capability exists).

2886 Despite the fact this **C_CloseSession** is supposed to close a session, the return value
 2887 **CKR_SESSION_CLOSED** is an *error* return. It actually indicates the (probably somewhat unlikely) event
 2888 that while this function call was executing, another call was made to **C_CloseSession** to close this
 2889 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and
 2890 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of
 2891 behavior.

2892 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2893 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2894 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

2895 Example:

```
2896 CK_SLOT_ID slotID;  
2897 CK_BYTE application;  
2898 CK_NOTIFY MyNotify;  
2899 CK_SESSION_HANDLE hSession;  
2900 CK_RV rv;  
2901  
2902 .  
2903 .  
2904 application = 17;  
2905 MyNotify = &EncryptionSessionCallback;  
2906 rv = C_OpenSession(  
2907     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,  
2908     (CK_VOID_PTR) &application, MyNotify,  
2909     &hSession);  
2910 if (rv == CKR_OK) {  
2911     .  
2912     .  
2913     C_CloseSession(hSession);  
2914 }
```

2915 5.6.3 C_CloseAllSessions

```
2916 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions) (  
2917     CK_SLOT_ID slotID  
2918 );
```

2919 **C_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token's slot.

2920 When a session is closed, all session objects created by the session are destroyed automatically.

2921 After successful execution of this function, the login state of the token for the application returns to public
2922 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W
2923 Public sessions.

2924 Depending on the token, when the last open session any application has with the token is closed, the
2925 token may be “ejected” from its reader (if this capability exists).

2926 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2927 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2928 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

2929 Example:

```
2930 CK_SLOT_ID slotID;  
2931 CK_RV rv;  
2932  
2933 .  
2934 .
```

```
2935 | rv = C_CloseAllSessions(slotID);
```

2936 5.6.4 C_GetSessionInfo

```
2937 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (  
2938     CK_SESSION_HANDLE hSession,  
2939     CK_SESSION_INFO_PTR pInfo  
2940 );
```

2941 **C_GetSessionInfo** obtains information about a session. *hSession* is the session's handle; *pInfo* points to
2942 the location that receives the session information.

2943 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2944 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2945 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
2946 CKR_ARGUMENTS_BAD.

2947 Example:

```
2948 CK_SESSION_HANDLE hSession;  
2949 CK_SESSION_INFO info;  
2950 CK_RV rv;  
2951  
2952 .  
2953 .  
2954 rv = C_GetSessionInfo(hSession, &info);  
2955 if (rv == CKR_OK) {  
2956     if (info.state == CKS_RW_USER_FUNCTIONS) {  
2957         .  
2958         .  
2959     }  
2960     .  
2961     .  
2962 }
```

2963 5.6.5 C_SessionCancel

```
2964 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (  
2965     CK_SESSION_HANDLE hSession  
2966     CK_FLAGS flags  
2967 );
```

2968 **C_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*
2969 indicates the operations to cancel.

2970 To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical
2971 bitwise OR of one or more of the bit flags defined in the **CK_MECHANISM_INFO** structure.

2972 If no flags are set, the session state will not be modified and CKR_OK will be returned.

2973 If a flag is set for an operation that has not been initialized in the session, the operation flag will be
2974 ignored and **C_SessionCancel** will behave as if the operation flag was not set.

2975 If any of the operations indicated by the *flags* parameter cannot be cancelled,
2976 CKR_OPERATION_CANCEL_FAILED must be returned. If multiple operation flags were set and
2977 CKR_OPERATION_CANCEL_FAILED is returned, this function does not provide any information about
2978 which operation(s) could not be cancelled. If an application desires to know if any single operation could
2979 not be cancelled, the application should not call **C_SessionCancel** with multiple flags set.

2980 If **C_SessionCancel** is called from an application callback (see Section 5.16), no action will be taken by
2981 the library and CKR_FUNCTION_FAILED must be returned.

2982 If **C_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation
2983 should still be left in an active state. However, it is expected that some Cryptoki implementations may not
2984 support this and return CKR_OPERATION_CANCEL_FAILED unless flags for both operations are
2985 provided.

2986 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2987 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2988 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_CANCEL_FAILED,
2989 CKR_TOKEN_NOT_PRESENT.

2990 Example:

```
2991 CK_SESSION_HANDLE hSession;  
2992 CK_RV rv;  
2993  
2994 rv = C_EncryptInit(hSession, &mechanism, hKey);  
2995 if (rv != CKR_OK)  
2996 {  
2997     .  
2998     .  
2999 }  
3000  
3001 rv = C_SessionCancel (hSession, CKF_ENCRYPT);  
3002 if (rv != CKR_OK)  
3003 {  
3004     .  
3005     .  
3006 }  
3007  
3008 rv = C_EncryptInit(hSession, &mechanism, hKey);  
3009 if (rv != CKR_OK)  
3010 {  
3011     .  
3012     .  
3013 }  
3014
```

3015
3016
3017 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual
3018 operations. The additional text is highlighted.
3019

3020 5.6.6 C_GetOperationState

```
3021 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (  
3022     CK_SESSION_HANDLE hSession,  
3023     CK_BYTE_PTR pOperationState,
```

```
3024 CK_ULONG_PTR pulOperationStateLen
3025 );
```

3026 **C_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a
3027 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the
3028 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

3029 Although the saved state output by **C_GetOperationState** is not really produced by a "cryptographic
3030 mechanism", **C_GetOperationState** nonetheless uses the convention described in Section 5.2 on
3031 producing output.

3032 Precisely what the "cryptographic operations state" this function saves is varies from token to token;
3033 however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic
3034 activities of a session.

3035 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is
3036 using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized
3037 properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The
3038 application now wants to "save the state" of this digest operation, so that it can continue it later. In this
3039 particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic
3040 operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit
3041 internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data
3042 indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken
3043 together, these three pieces of information suffice to continue the current hashing operation at a later
3044 time.

3045 Consider next a session which is performing an encryption operation with DES (a block cipher with a
3046 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_DES_CBC**
3047 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been
3048 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already
3049 been produced and output. In this case, the cryptographic operations state of the session most likely
3050 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-
3051 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some
3052 administrative data indicating that this saved state comes from a session which was performing DES
3053 encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState**
3054 for more information on whether or not the key is present in the saved state).

3055 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the
3056 cryptographic operations state of the session will contain all the necessary information to restore both
3057 operations.

3058 An attempt to save the cryptographic operations state of a session which does not currently have some
3059 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message
3060 recovery, verification without message recovery, or some legal combination of two of these) should fail
3061 with the error **CKR_OPERATION_NOT_INITIALIZED**.

3062 An attempt to save the cryptographic operations state of a session which is performing an appropriate
3063 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain
3064 necessary state information and/or key information can't leave the token, for example) should fail with the
3065 error **CKR_STATE_UNSAVEABLE**.

3066 Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
3067 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
3068 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
3069 **CKR_OPERATION_NOT_INITIALIZED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**,
3070 **CKR_STATE_UNSAVEABLE**, **CKR_ARGUMENTS_BAD**.

3071 Example: see **C_SetOperationState**.

3072 5.6.7 C_SetOperationState

```
3073 CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState) (
3074     CK_SESSION_HANDLE hSession,
```

```

3075 CK_BYTE_PTR pOperationState,
3076 CK_ULONG ulOperationStateLen,
3077 CK_OBJECT_HANDLE hEncryptionKey,
3078 CK_OBJECT_HANDLE hAuthenticationKey
3079 );

```

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).

3128 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3129 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3130 CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED,
3131 CKR_OK, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED,
3132 CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

3133 Example:

```
3134 CK_SESSION_HANDLE hSession;  
3135 CK_MECHANISM digestMechanism;  
3136 CK_BYTE_PTR pState;  
3137 CK_ULONG ulStateLen;  
3138 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};  
3139 CK_BYTE data2[] = {0x02, 0x04, 0x08};  
3140 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};  
3141 CK_BYTE pDigest[20];  
3142 CK_ULONG ulDigestLen;  
3143 CK_RV rv;  
3144  
3145 .  
3146 .  
3147 /* Initialize hash operation */  
3148 rv = C_DigestInit(hSession, &digestMechanism);  
3149 assert(rv == CKR_OK);  
3150  
3151 /* Start hashing */  
3152 rv = C_DigestUpdate(hSession, data1, sizeof(data1));  
3153 assert(rv == CKR_OK);  
3154  
3155 /* Find out how big the state might be */  
3156 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);  
3157 assert(rv == CKR_OK);  
3158  
3159 /* Allocate some memory and then get the state */  
3160 pState = (CK_BYTE_PTR) malloc(ulStateLen);  
3161 rv = C_GetOperationState(hSession, pState, &ulStateLen);  
3162  
3163 /* Continue hashing */  
3164 rv = C_DigestUpdate(hSession, data2, sizeof(data2));  
3165 assert(rv == CKR_OK);  
3166  
3167 /* Restore state. No key handles needed */  
3168 rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);  
3169 assert(rv == CKR_OK);  
3170  
3171 /* Continue hashing from where we saved state */
```



```

3172 rv = C_DigestUpdate(hSession, data3, sizeof(data3));
3173 assert(rv == CKR_OK);
3174
3175 /* Conclude hashing operation */
3176 ulDigestLen = sizeof(pDigest);
3177 rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
3178 if (rv == CKR_OK) {
3179     /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
3180     .
3181     .
3182 }

```

3183 5.6.8 C_Login

```

3184 CK_DECLARE_FUNCTION(CK_RV, C_Login)(
3185     CK_SESSION_HANDLE hSession,
3186     CK_USER_TYPE userType,
3187     CK_UTF8CHAR_PTR pPin,
3188     CK_ULONG ulPinLen
3189 );

```

3190 **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to
3191 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid
3192 UTF8 character, but the token may impose subset restrictions.

3193 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3194 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3195 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of C_Login depends on the
3196 context in which it is called. Improper use of this user type will result in a return value
3197 CKR_OPERATION_NOT_INITIALIZED..

3198 If the token has a "protected authentication path", as indicated by the
3199 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3200 that there is some way for a user to be authenticated to the token without having to send a PIN through
3201 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3202 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
3203 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3204 parameter to **C_Login** should be NULL_PTR. When **C_Login** returns, whatever authentication method
3205 supported by the token will have been performed; a return value of CKR_OK means that the user was
3206 successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3207 denied access.

3208 If there are any active cryptographic or object finding operations in an application's session, and then
3209 **C_Login** is successfully executed by that application, it may or may not be the case that those operations
3210 are still active. Therefore, before logging in, any active operations should be finished.

3211 If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the
3212 SO into a session (see [\[PKCS11-UG\]](#) for further details). An attempt to do this will result in the error code
3213 CKR_SESSION_READ_ONLY_EXISTS.

3214 C_Login may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3215 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3216 cryptographic operation on this key. See further Section 4.9.

3217 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3218 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3219 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3220 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,

3221 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3222 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
3223 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3224 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3225 Example: see **C_Logout**.

3226 5.6.9 C_LoginUser

```
3227 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (  
3228     CK_SESSION_HANDLE hSession,  
3229     CK_USER_TYPE userType,  
3230     CK_UTF8CHAR_PTR pPin,  
3231     CK_ULONG ulPinLen,  
3232     CK_UTF8CHAR_PTR pUsername,  
3233     CK_ULONG ulUsernameLen  
3234 );
```

3235 **C_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*
3236 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,
3237 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to
3238 contain any valid UTF8 character, but the token may impose subset restrictions.

3239 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3240 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3241 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of **C_LoginUser** depends
3242 on the context in which it is called. Improper use of this user type will result in a return value
3243 CKR_OPERATION_NOT_INITIALIZED.

3244 If the token has a "protected authentication path", as indicated by the
3245 CKF_PROTECTED_AUTHENTICATION_PATH flag in its CK_TOKEN_INFO being set, then that means
3246 that there is some way for a user to be authenticated to the token without having to send a PIN through
3247 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3248 on the slot device. The user might not even use a PIN—authentication could be achieved by some
3249 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3250 parameter to **C_LoginUser** should be NULL_PTR. When **C_LoginUser** returns, whatever authentication
3251 method supported by the token will have been performed; a return value of CKR_OK means that the user
3252 was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3253 denied access.

3254 If there are any active cryptographic or object finding operations in an application's session, and then
3255 **C_LoginUser** is successfully executed by that application, it may or may not be the case that those
3256 operations are still active. Therefore, before logging in, any active operations should be finished.

3257 If the application calling **C_LoginUser** has a R/O session open with the token, then it will be unable to log
3258 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
3259 code CKR_SESSION_READ_ONLY_EXISTS.

3260 **C_LoginUser** may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3261 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3262 cryptographic operation on this key. See further Section 4.9.

3263 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3264 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3265 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3266 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
3267 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3268 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
3269 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3270 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3271 Example:

```

3272 CK_SESSION_HANDLE hSession;
3273 CK_UTF8CHAR userPin[] = {"MyPIN"};
3274 CK_UTF8CHAR userName[] = {"MyUserName"};
3275 CK_RV rv;
3276
3277 rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName,
3278 sizeof(userName)-1);
3279 if (rv == CKR_OK) {
3280     .
3281     .
3282     rv = C_Logout(hSession);
3283     if (rv == CKR_OK) {
3284         .
3285         .
3286     }
3287 }

```

3288 5.6.10 C_Logout

```

3289 CK_DECLARE_FUNCTION(CK_RV, C_Logout) (
3290     CK_SESSION_HANDLE hSession
3291 );

```

3292 **C_Logout** logs a user out from a token. *hSession* is the session's handle.

3293 Depending on the current user type, if the call succeeds, each of the application's sessions will enter
3294 either the "R/W Public Session" state or the "R/O Public Session" state.

3295 When **C_Logout** successfully executes, any of the application's handles to private objects become invalid
3296 (even if a user is later logged back into the token, those handles remain invalid). In addition, all private
3297 session objects from sessions belonging to the application are destroyed.

3298 If there are any active cryptographic or object-finding operations in an application's session, and then
3299 **C_Logout** is successfully executed by that application, it may or may not be the case that those
3300 operations are still active. Therefore, before logging out, any active operations should be finished.

3301 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3302 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3303 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3304 CKR_USER_NOT_LOGGED_IN.

3305 Example:

```

3306 CK_SESSION_HANDLE hSession;
3307 CK_UTF8CHAR userPin[] = {"MyPIN"};
3308 CK_RV rv;
3309
3310 rv = C_Login(hSession, CKU_USER, userPin, sizeof(userPin)-1);
3311 if (rv == CKR_OK) {
3312     .
3313     .
3314     rv = C_Logout(hSession);
3315     if (rv == CKR_OK) {

```

3316
3317
3318
3319

```
.  
.   
}  
}
```

3320 5.7 Object management functions

3321 Cryptoki provides the following functions for managing objects. Additional functions provided specifically
3322 for managing key objects are described in Section 5.18.

3323 5.7.1 C_CreateObject

```
3324 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject) (  
3325     CK_SESSION_HANDLE hSession,  
3326     CK_ATTRIBUTE_PTR pTemplate,  
3327     CK_ULONG ulCount,  
3328     CK_OBJECT_HANDLE_PTR phObject  
3329 );
```

3330 **C_CreateObject** creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's
3331 template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives
3332 the new object's handle.

3333 If a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without
3334 creating any object.

3335 If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to
3336 CK_FALSE. If that key object is a secret or private key then the new key will have the
3337 **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE**
3338 attribute set to CK_FALSE.

3339 Only session objects can be created during a read-only session. Only public objects can be created
3340 unless the normal user is logged in.

3341 Whenever an object is created, a value for CKA_UNIQUE_ID is generated and assigned to the new
3342 object (See Section 4.4.1).

3343 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
3344 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
3345 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
3346 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
3347 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3348 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3349 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
3350 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3351 Example:

```
3352 CK_SESSION_HANDLE hSession;  
3353 CK_OBJECT_HANDLE  
3354     hData,  
3355     hCertificate,  
3356     hKey;  
3357 CK_OBJECT_CLASS  
3358     dataClass = CKO_DATA,  
3359     certificateClass = CKO_CERTIFICATE,  
3360     keyClass = CKO_PUBLIC_KEY;  
3361 CK_KEY_TYPE keyType = CKK_RSA;
```

```

3362 CK_UTF8CHAR application[] = {"My Application"};
3363 CK_BYTE dataValue[] = {...};
3364 CK_BYTE subject[] = {...};
3365 CK_BYTE id[] = {...};
3366 CK_BYTE certificateValue[] = {...};
3367 CK_BYTE modulus[] = {...};
3368 CK_BYTE exponent[] = {...};
3369 CK_BBOOL true = CK_TRUE;
3370 CK_ATTRIBUTE dataTemplate[] = {
3371     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3372     {CKA_TOKEN, &true, sizeof(true)},
3373     {CKA_APPLICATION, application, sizeof(application)-1},
3374     {CKA_VALUE, dataValue, sizeof(dataValue)}
3375 };
3376 CK_ATTRIBUTE certificateTemplate[] = {
3377     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
3378     {CKA_TOKEN, &true, sizeof(true)},
3379     {CKA_SUBJECT, subject, sizeof(subject)},
3380     {CKA_ID, id, sizeof(id)},
3381     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
3382 };
3383 CK_ATTRIBUTE keyTemplate[] = {
3384     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3385     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3386     {CKA_WRAP, &true, sizeof(true)},
3387     {CKA_MODULUS, modulus, sizeof(modulus)},
3388     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
3389 };
3390 CK_RV rv;
3391
3392 .
3393 .
3394 /* Create a data object */
3395 rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
3396 if (rv == CKR_OK) {
3397     .
3398     .
3399 }
3400
3401 /* Create a certificate object */
3402 rv = C_CreateObject(
3403     hSession, certificateTemplate, 5, &hCertificate);
3404 if (rv == CKR_OK) {

```

```

3405     .
3406     .
3407 }
3408
3409 /* Create an RSA public key object */
3410 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3411 if (rv == CKR_OK) {
3412     .
3413     .
3414 }

```

3415 5.7.2 C_CopyObject

```

3416 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject) (
3417     CK_SESSION_HANDLE hSession,
3418     CK_OBJECT_HANDLE hObject,
3419     CK_ATTRIBUTE_PTR pTemplate,
3420     CK_ULONG ulCount,
3421     CK_OBJECT_HANDLE_PTR phNewObject
3422 );

```

3423 **C_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;
3424 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number
3425 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of
3426 the object.

3427 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,
3428 in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from
3429 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's
3430 **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may
3431 specify that the new key's **CKA_SENSITIVE** attribute be CK_TRUE; the new key will have the same
3432 value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). It may also specify new values of
3433 the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object). If the
3434 template specifies a value of an attribute which is incompatible with other existing attributes of the object,
3435 the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

3436 If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without
3437 creating any object. If the object indicated by *hObject* has its CKA_COPYABLE attribute set to
3438 CK_FALSE, C_CopyObject will return CKR_ACTION_PROHIBITED.

3439 Whenever an object is copied, a new value for CKA_UNIQUE_ID is generated and assigned to the new
3440 object (See Section 4.4.1).

3441 Only session objects can be created during a read-only session. Only public objects can be created
3442 unless the normal user is logged in.

3443 Return values: , CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3444 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3445 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3446 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3447 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3448 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3449 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT,
3450 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3451 Example:

```

3452 CK_SESSION_HANDLE hSession;

```

```

3453 CK_OBJECT_HANDLE hKey, hNewKey;
3454 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
3455 CK_KEY_TYPE keyType = CKK_DES;
3456 CK_BYTE id[] = {...};
3457 CK_BYTE keyValue[] = {...};
3458 CK_BBOOL false = CK_FALSE;
3459 CK_BBOOL true = CK_TRUE;
3460 CK_ATTRIBUTE keyTemplate[] = {
3461     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3462     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3463     {CKA_TOKEN, &>false, sizeof(false)},
3464     {CKA_ID, id, sizeof(id)},
3465     {CKA_VALUE, keyValue, sizeof(keyValue)}
3466 };
3467 CK_ATTRIBUTE copyTemplate[] = {
3468     {CKA_TOKEN, &>true, sizeof(true)}
3469 };
3470 CK_RV rv;
3471
3472 .
3473 .
3474 /* Create a DES secret key session object */
3475 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3476 if (rv == CKR_OK) {
3477     /* Create a copy which is a token object */
3478     rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);
3479     .
3480     .
3481 }

```

5.7.3 C_DestroyObject

```

3483 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject)(
3484     CK_SESSION_HANDLE hSession,
3485     CK_OBJECT_HANDLE hObject
3486 );

```

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.

Certain objects may not be destroyed. Calling **C_DestroyObject** on such objects will result in the **CKR_ACTION_PROHIBITED** error code. An application can consult the object's **CKA_DESTROYABLE** attribute to determine if an object may be destroyed or not.

Return values: **CKR_ACTION_PROHIBITED**, **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**,

3497 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3498 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
3499 CKR_TOKEN_WRITE_PROTECTED.

3500 Example: see **C_GetObjectSize**.

3501 5.7.4 C_GetObjectSize

```
3502 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize) (  
3503     CK_SESSION_HANDLE hSession,  
3504     CK_OBJECT_HANDLE hObject,  
3505     CK_ULONG_PTR pulSize  
3506 );
```

3507 **C_GetObjectSize** gets the size of an object in bytes. *hSession* is the session's handle; *hObject* is the
3508 object's handle; *pulSize* points to the location that receives the size in bytes of the object.

3509 Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure
3510 of how much token memory the object takes up. If an application deletes (say) a private object of size *S*,
3511 it might be reasonable to assume that the *ulFreePrivateMemory* field of the token's **CK_TOKEN_INFO**
3512 structure increases by approximately *S*.

3513 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3514 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3515 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
3516 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3517 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3518 Example:

```
3519 CK_SESSION_HANDLE hSession;  
3520 CK_OBJECT_HANDLE hObject;  
3521 CK_OBJECT_CLASS dataClass = CKO_DATA;  
3522 CK_UTF8CHAR application[] = {"My Application"};  
3523 CK_BYTE value[] = {...};  
3524 CK_BBOOL true = CK_TRUE;  
3525 CK_ATTRIBUTE template[] = {  
3526     {CKA_CLASS, &dataClass, sizeof(dataClass)},  
3527     {CKA_TOKEN, &true, sizeof(true)},  
3528     {CKA_APPLICATION, application, sizeof(application)-1},  
3529     {CKA_VALUE, value, sizeof(value)}  
3530 };  
3531 CK_ULONG ulSize;  
3532 CK_RV rv;  
3533  
3534 .  
3535 .  
3536 rv = C_CreateObject(hSession, template, 4, &hObject);  
3537 if (rv == CKR_OK) {  
3538     rv = C_GetObjectSize(hSession, hObject, &ulSize);  
3539     if (rv != CKR_INFORMATION_SENSITIVE) {  
3540         .  
3541         .
```



```

3542     }
3543
3544     rv = C_DestroyObject(hSession, hObject);
3545     .
3546     .
3547 }

```

3548 5.7.5 C_GetAttributeValue

```

3549 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (
3550     CK_SESSION_HANDLE hSession,
3551     CK_OBJECT_HANDLE hObject,
3552     CK_ATTRIBUTE_PTR pTemplate,
3553     CK_ULONG ulCount
3554 );

```

3555 **C_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's
3556 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
3557 values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the
3558 template.

3559 For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C_GetAttributeValue** performs the following
3560 algorithm:

- 3561 1. If the specified attribute (i.e., the attribute specified by the type field) for the object cannot be revealed
3562 because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to
3563 hold the value CK_UNAVAILABLE_INFORMATION.
- 3564 2. Otherwise, if the specified value for the object is invalid (the object does not possess such an
3565 attribute), then the *ulValueLen* field in that triple is modified to hold the value
3566 CK_UNAVAILABLE_INFORMATION.
- 3567 3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold
3568 the exact length of the specified attribute for the object.
- 3569 4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified
3570 attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the
3571 *ulValueLen* field is modified to hold the exact length of the attribute.
- 3572 5. Otherwise, the *ulValueLen* field is modified to hold the value CK_UNAVAILABLE_INFORMATION.

3573 If case 1 applies to any of the requested attributes, then the call should return the value
3574 CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should
3575 return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes,
3576 then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these
3577 error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the
3578 requested attributes will CKR_OK be returned.

3579 In the special case of an attribute whose value is an array of attributes, for example
3580 CKA_WRAP_TEMPLATE, where it is passed in with *pValue* not NULL, the length specified in *ulValueLen*
3581 MUST be large enough to hold all attributes in the array. If the *pValue* of elements within the array is
3582 NULL_PTR then the *ulValueLen* of elements within the array will be set to the required length. If the
3583 *pValue* of elements within the array is not NULL_PTR, then the *ulValueLen* element of attributes within
3584 the array MUST reflect the space that the corresponding *pValue* points to, and *pValue* is filled in if there is
3585 sufficient room. Therefore it is important to initialize the contents of a buffer before calling
3586 C_GetAttributeValue to get such an array value. Note that the type element of attributes within the array
3587 MUST be ignored on input and MUST be set on output. If any *ulValueLen* within the array isn't large
3588 enough, it will be set to CK_UNAVAILABLE_INFORMATION and the function will return
3589 CKR_BUFFER_TOO_SMALL, as it does if an attribute in the *pTemplate* argument has *ulValueLen* too
3590 small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
3591 type having the CKF_ARRAY_ATTRIBUTE bit set.

3592 Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and
3593 CKR_BUFFER_TOO_SMALL do not denote true errors for **C_GetAttributeValue**. If a call to
3594 **C_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed
3595 every attribute in the template supplied to **C_GetAttributeValue**. Each attribute in the template whose
3596 value *can be* returned by the call to **C_GetAttributeValue** *will be* returned by the call to
3597 **C_GetAttributeValue**.

3598 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE,
3599 CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL,
3600 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3601 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3602 CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED,
3603 CKR_SESSION_HANDLE_INVALID.

3604 Example:

```
3605 CK_SESSION_HANDLE hSession;  
3606 CK_OBJECT_HANDLE hObject;  
3607 CK_BYTE_PTR pModulus, pExponent;  
3608 CK_ATTRIBUTE template[] = {  
3609     {CKA_MODULUS, NULL_PTR, 0},  
3610     {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}  
3611 };  
3612 CK_RV rv;  
3613  
3614 .  
3615 .  
3616 rv = C_GetAttributeValue(hSession, hObject, template, 2);  
3617 if (rv == CKR_OK) {  
3618     pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);  
3619     template[0].pValue = pModulus;  
3620     /* template[0].ulValueLen was set by C_GetAttributeValue */  
3621  
3622     pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);  
3623     template[1].pValue = pExponent;  
3624     /* template[1].ulValueLen was set by C_GetAttributeValue */  
3625  
3626     rv = C_GetAttributeValue(hSession, hObject, template, 2);  
3627     if (rv == CKR_OK) {  
3628         .  
3629         .  
3630     }  
3631     free(pModulus);  
3632     free(pExponent);  
3633 }
```

5.7.6 C_SetAttributeValue

```
CK_DECLARE_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.

Certain objects may not be modified. Calling C_SetAttributeValue on such objects will result in the CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_MODIFIABLE attribute to determine if an object may be modified or not.

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 4.1.2 for more details.

Return values: CKR_ACTION_PROHIBITED, 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) {
    .
    .
}
```

5.7.7 C_FindObjectsInit

```
CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit)(
    CK_SESSION_HANDLE hSession,
    CK_ATTRIBUTE_PTR pTemplate,
```

```

3680     CK_ULONG ulCount
3681 );

```

3682 **C_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is
3683 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;
3684 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-
3685 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

3686 After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain
3687 handles for objects matching the template, and then eventually call **C_FindObjectsFinal** to finish the
3688 active search operation. At most one search operation may be active at a given time in a given session.

3689 The object search operation will only find objects that the session can view. For example, an object
3690 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the
3691 search template specifies that the search is for private objects).

3692 If a search operation is active, and objects are created or destroyed which fit the search template for the
3693 active search operation, then those objects may or may not be found by the search operation. Note that
3694 this means that, under these circumstances, the search operation may return invalid object handles.

3695 Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and
3696 CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template
3697 with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can initialize a
3698 search operation which will match no objects and return CKR_OK.

3699 If the CKA_UNIQUE_ID attribute is present in the search template, either zero or one objects will be
3700 found, since at most one object can have any particular CKA_UNIQUE_ID value.

3701 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID,
3702 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3703 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3704 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
3705 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3706 Example: see **C_FindObjectsFinal**.

3707 5.7.8 C_FindObjects

```

3708 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects)(
3709     CK_SESSION_HANDLE hSession,
3710     CK_OBJECT_HANDLE_PTR phObject,
3711     CK_ULONG ulMaxObjectCount,
3712     CK_ULONG_PTR pulObjectCount
3713 );

```

3714 **C_FindObjects** continues a search for token and session objects that match a template, obtaining
3715 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives
3716 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles
3717 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles
3718 returned.

3719 If there are no more objects matching the template, then the location that *pulObjectCount* points to
3720 receives the value 0.

3721 The search MUST have been initialized with **C_FindObjectsInit**.

3722 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3723 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3724 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3725 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3726 Example: see **C_FindObjectsFinal**.

5.7.9 C_FindObjectsFinal

```
CK_DECLARE_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);
```

5.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

5.8.1 C_EncryptInit

```
CK_DECLARE_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.

3769 After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or
3770 call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts.
3771 The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** to
3772 *actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the
3773 application MUST call **C_EncryptInit** again.

3774 **C_EncryptInit** can be called with *pMechanism* set to NULL_PTR to terminate an active encryption
3775 operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
3776 must be returned.

3777 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3778 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
3779 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
3780 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
3781 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
3782 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3783 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
3784 CKR_OPERATION_CANCEL_FAILED.

3785 Example: see **C_EncryptFinal**.

3786 5.8.2 C_Encrypt

```
3787 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt) (  
3788     CK_SESSION_HANDLE hSession,  
3789     CK_BYTE_PTR pData,  
3790     CK_ULONG ulDataLen,  
3791     CK_BYTE_PTR pEncryptedData,  
3792     CK_ULONG_PTR pulEncryptedDataLen  
3793 );
```

3794 **C_Encrypt** encrypts single-part data. *hSession* is the session's handle; *pData* points to the data;
3795 *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the
3796 encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted
3797 data.

3798 **C_Encrypt** uses the convention described in Section 5.2 on producing output.

3799 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always
3800 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3801 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
3802 ciphertext.

3803 **C_Encrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_EncryptInit**
3804 without intervening **C_EncryptUpdate** calls.

3805 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
3806 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
3807 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
3808 **C_Encrypt** will fail with return code CKR_DATA_LEN_RANGE.

3809 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to
3810 the same location.

3811 For most mechanisms, **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** operations followed
3812 by **C_EncryptFinal**.

3813 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3814 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
3815 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3816 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3817 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
3818 CKR_SESSION_HANDLE_INVALID.

3819 Example: see **C_EncryptFinal** for an example of similar functions.

5.8.3 C_EncryptUpdate

```
CK_DECLARE_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 5.2 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**.

5.8.4 C_EncryptFinal

```
CK_DECLARE_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 5.2 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
```



```

3869
3870 CK_ULONG firstPieceLen, secondPieceLen;
3871 CK_SESSION_HANDLE hSession;
3872 CK_OBJECT_HANDLE hKey;
3873 CK_BYTE iv[8];
3874 CK_MECHANISM mechanism = {
3875     CKM_DES_CBC_PAD, iv, sizeof(iv)
3876 };
3877 CK_BYTE data[PLAINTEXT_BUF_SZ];
3878 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
3879 CK_ULONG ulEncryptedData1Len;
3880 CK_ULONG ulEncryptedData2Len;
3881 CK_ULONG ulEncryptedData3Len;
3882 CK_RV rv;
3883
3884 .
3885 .
3886 firstPieceLen = 90;
3887 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
3888 rv = C_EncryptInit(hSession, &mechanism, hKey);
3889 if (rv == CKR_OK) {
3890     /* Encrypt first piece */
3891     ulEncryptedData1Len = sizeof(encryptedData);
3892     rv = C_EncryptUpdate(
3893         hSession,
3894         &data[0], firstPieceLen,
3895         &encryptedData[0], &ulEncryptedData1Len);
3896     if (rv != CKR_OK) {
3897         .
3898         .
3899     }
3900
3901     /* Encrypt second piece */
3902     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
3903     rv = C_EncryptUpdate(
3904         hSession,
3905         &data[firstPieceLen], secondPieceLen,
3906         &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
3907     if (rv != CKR_OK) {
3908         .
3909         .
3910     }
3911

```



```

3912  /* Get last little encrypted bit */
3913  ulEncryptedData3Len =
3914      sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
3915  rv = C_EncryptFinal(
3916      hSession,
3917      &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
3918      &ulEncryptedData3Len);
3919  if (rv != CKR_OK) {
3920      .
3921      .
3922  }
3923  }

```

5.9 Message-based encryption functions

Message-based encryption refers to the process of encrypting multiple messages using the same encryption mechanism and encryption key. The encryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based encryption:

5.9.1 C_MessageEncryptInit

```

CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);

```

C_MessageEncryptInit prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. *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_MessageEncryptInit**, the application can either call **C_EncryptMessage** to encrypt a message in a single part, or call **C_EncryptMessageBegin**, followed by **C_EncryptMessageNext** one or more times, to encrypt a message in multiple parts. This may be repeated several times. The message-based encryption process is active until the application calls **C_MessageEncryptFinal** to finish the message-based encryption process.

C_MessageEncryptInit can be called with *pMechanism* set to NULL_PTR to terminate a message-based encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an active operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

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, CKR_OPERATION_CANCEL_FAILED.

5.9.2 C_EncryptMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG ulPlaintextLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG_PTR pulCiphertextLen
);
```

C_EncryptMessage encrypts a message in a single part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data; *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location that holds the length in bytes of the encrypted data.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the *pParameter* buffer.

If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

C_EncryptMessage uses the convention described in Section 5.2 on producing output.

The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**. A call to **C_EncryptMessage** begins and terminates a message encryption operation.

C_EncryptMessage cannot be called in the middle of a multi-part message encryption operation.

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_EncryptMessage** 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 *pPlaintext* and *pCiphertext* point to the same location.

For most mechanisms, **C_EncryptMessage** is equivalent to **C_EncryptMessageBegin** followed by a sequence of **C_EncryptMessageNext** operations.

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_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5.9.3 C_EncryptMessageBegin

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
);
```

C_EncryptMessageBegin begins a multiple-part message encryption operation. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the

4007 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
 4008 for an AEAD mechanism.

4009 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
 4010 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
 4011 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
 4012 generator will be output to the *pParameter* buffer.

4013 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be
 4014 set to (NULL, 0).

4015 After calling **C_EncryptMessageBegin**, the application should call **C_EncryptMessageNext** one or
 4016 more times to encrypt the message in multiple parts. The message encryption operation is active until the
 4017 application uses a call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually
 4018 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the
 4019 application MUST call **C_EncryptMessage** or **C_EncryptMessageBegin** again.

4020 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4021 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 4022 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
 4023 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 4024 CKR_USER_NOT_LOGGED_IN.

4025 5.9.4 C_EncryptMessageNext

```

4026 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (
4027     CK_SESSION_HANDLE hSession,
4028     CK_VOID_PTR pParameter,
4029     CK_ULONG ulParameterLen,
4030     CK_BYTE_PTR pPlaintextPart,
4031     CK_ULONG ulPlaintextPartLen,
4032     CK_BYTE_PTR pCiphertextPart,
4033     CK_ULONG_PTR pulCiphertextPartLen,
4034     CK_FLAGS flags
4035 );
  
```

4036 **C_EncryptMessageNext** continues a multiple-part message encryption operation, processing another
 4037 message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
 4038 mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the
 4039 plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart*
 4040 points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the
 4041 location that holds the length in bytes of the encrypted message part; flags is set to 0 if there is more
 4042 plaintext data to follow, or set to CKF_END_OF_MESSAGE if this is the last plaintext part.

4043 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
 4044 passed to **C_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For
 4045 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
 4046 generator will be output to the *pParameter* buffer.

4047 **C_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

4048 The message encryption operation MUST have been started with **C_EncryptMessageBegin**. This
 4049 function may be called any number of times in succession. A call to **C_EncryptMessageNext** with flags=0
 4050 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
 4051 encryption operation. A call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE always
 4052 terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
 4053 successful call (i.e., one which returns **CKR_OK**) to determine the length of the buffer needed to hold the
 4054 ciphertext.

4055 Although the last **C_EncryptMessageNext** call ends the encryption of a message, it does not finish the
 4056 message-based encryption process. Additional **C_EncryptMessage** or **C_EncryptMessageBegin** and
 4057 **C_EncryptMessageNext** calls may be made on the session.

4058 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*
 4059 point to the same location.

4060 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
 4061 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
 4062 are not satisfied when the final message part is supplied (i.e., with flags=CKF_END_OF_MESSAGE),
 4063 then **C_EncryptMessageNext** will fail with return code CKR_DATA_LEN_RANGE.

4064 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4065 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
 4066 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
 4067 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
 4068 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4069 5.9.5 C_MessageEncryptFinal

```
4070 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (
4071     CK_SESSION_HANDLE hSession
4072 );
```

4073 **C_MessageEncryptFinal** finishes a message-based encryption process. *hSession* is the session's
 4074 handle.

4075 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**.

4076 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4077 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4078 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4079 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4080 CKR_SESSION_HANDLE_INVALID.

4081 Example:

```
4082 #define PLAINTEXT_BUF_SZ 200
4083 #define AUTH_BUF_SZ 100
4084 #define CIPHERTEXT_BUF_SZ 256
4085
4086 CK_SESSION_HANDLE hSession;
4087 CK_OBJECT_HANDLE hKey;
4088 CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
4089 CK_BYTE tag[16];
4090 CK_GCM_MESSAGE_PARAMS gcmParams = {
4091     iv,
4092     sizeof(iv) * 8,
4093     0,
4094     CKG_NO_GENERATE,
4095     tag,
4096     sizeof(tag) * 8
4097 };
4098 CK_MECHANISM mechanism = {
4099     CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
4100 };
4101 CK_BYTE data[2][PLAINTEXT_BUF_SZ];
4102 CK_BYTE auth[2][AUTH_BUF_SZ];
```

```

4103 CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
4104 CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
4105 CK_ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;
4106
4107 /* error handling is omitted for better readability */
4108 .
4109 .
4110 C_MessageEncryptInit(hSession, &mechanism, hKey);
4111 /* encrypt message en bloc with given IV */
4112 ulEncryptedDataLen = sizeof(encryptedData[0]);
4113 C_EncryptMessage(hSession,
4114     &gcmParams, sizeof(gcmParams),
4115     &auth[0][0], sizeof(auth[0]),
4116     &data[0][0], sizeof(data[0]),
4117     &encryptedData[0][0], &ulEncryptedDataLen);
4118 /* iv and tag are set now for message */
4119
4120 /* encrypt message in two steps with generated IV */
4121 gcmParams.ivGenerator = CKG_GENERATE;
4122 C_EncryptMessageBegin(hSession,
4123     &gcmParams, sizeof(gcmParams),
4124     &auth[1][0], sizeof(auth[1])
4125 );
4126 /* encrypt first piece */
4127 ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
4128 C_EncryptMessageNext(hSession,
4129     &gcmParams, sizeof(gcmParams),
4130     &data[1][0], firstPieceLen,
4131     &encryptedData[1][0], &ulFirstEncryptedDataLen,
4132     0
4133 );
4134 /* encrypt second piece */
4135 ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
4136 C_EncryptMessageNext(hSession,
4137     &gcmParams, sizeof(gcmParams),
4138     &data[1][firstPieceLen], sizeof(data[1])-firstPieceLen,
4139     &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,
4140     CKF_END_OF_MESSAGE
4141 );
4142 /* tag is set now for message */
4143
4144 /* finalize */
4145 C_MessageEncryptFinal(hSession);

```

5.10 Decryption functions

Cryptoki provides the following functions for decrypting data:

5.10.1 C_DecryptInit

```
CK_DECLARE_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.

C_DecryptInit can be called with *pMechanism* set to NULL_PTR to terminate an active decryption operation. If an active operation cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

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, CKR_OPERATION_CANCEL_FAILED.

Example: see **C_DecryptFinal**.

5.10.2 C_Decrypt

```
CK_DECLARE_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 5.2 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 cannot be used to terminate a multi-part operation, and MUST be called after **C_DecryptInit** without intervening **C_DecryptUpdate** calls.

4194 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
4195 the same location.

4196 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4197 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4198 For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed
4199 by **C_DecryptFinal**.

4200 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4201 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4202 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4203 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4204 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4205 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4206 Example: see **C_DecryptFinal** for an example of similar functions.

4207 5.10.3 C_DecryptUpdate

```
4208 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (  
4209     CK_SESSION_HANDLE hSession,  
4210     CK_BYTE_PTR pEncryptedPart,  
4211     CK_ULONG ulEncryptedPartLen,  
4212     CK_BYTE_PTR pPart,  
4213     CK_ULONG_PTR pulPartLen  
4214 );
```

4215 **C_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data
4216 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;
4217 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the
4218 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

4219 **C_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

4220 The decryption operation MUST have been initialized with **C_DecryptInit**. This function may be called
4221 any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than
4222 CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

4223 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to
4224 the same location.

4225 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4226 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4227 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4228 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4229 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4230 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4231 Example: See **C_DecryptFinal**.

4232 5.10.4 C_DecryptFinal

```
4233 CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal) (  
4234     CK_SESSION_HANDLE hSession,  
4235     CK_BYTE_PTR pLastPart,  
4236     CK_ULONG_PTR pulLastPartLen  
4237 );
```

4238 **C_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session's handle;
4239 *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to
4240 the location that holds the length of the last recovered data part.

4241 **C_DecryptFinal** uses the convention described in Section 5.2 on producing output.

4242 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal**
4243 always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4244 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4245 plaintext.

4246 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4247 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4248 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4249 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4250 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4251 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4252 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4253 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4254 Example:

```
4255 #define CIPHERTEXT_BUF_SZ 256
4256 #define PLAINTEXT_BUF_SZ 256
4257
4258 CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
4259 CK_SESSION_HANDLE hSession;
4260 CK_OBJECT_HANDLE hKey;
4261 CK_BYTE iv[8];
4262 CK_MECHANISM mechanism = {
4263     CKM_DES_CBC_PAD, iv, sizeof(iv)
4264 };
4265 CK_BYTE data[PLAINTEXT_BUF_SZ];
4266 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
4267 CK_ULONG ulData1Len, ulData2Len, ulData3Len;
4268 CK_RV rv;
4269
4270 .
4271 .
4272 firstEncryptedPieceLen = 90;
4273 secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ-firstEncryptedPieceLen;
4274 rv = C_DecryptInit(hSession, &mechanism, hKey);
4275 if (rv == CKR_OK) {
4276     /* Decrypt first piece */
4277     ulData1Len = sizeof(data);
4278     rv = C_DecryptUpdate(
4279         hSession,
4280         &encryptedData[0], firstEncryptedPieceLen,
4281         &data[0], &ulData1Len);
4282     if (rv != CKR_OK) {
4283         .
4284         .
4285     }
4286 }
```



```

4287  /* Decrypt second piece */
4288  ulData2Len = sizeof(data)-ulData1Len;
4289  rv = C_DecryptUpdate(
4290      hSession,
4291      &encryptedData[firstEncryptedPieceLen],
4292      secondEncryptedPieceLen,
4293      &data[ulData1Len], &ulData2Len);
4294  if (rv != CKR_OK) {
4295      .
4296      .
4297  }
4298
4299  /* Get last little decrypted bit */
4300  ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
4301  rv = C_DecryptFinal(
4302      hSession,
4303      &data[ulData1Len+ulData2Len], &ulData3Len);
4304  if (rv != CKR_OK) {
4305      .
4306      .
4307  }
4308  }

```

4309 5.11 Message-based decryption functions

4310 Message-based decryption refers to the process of decrypting multiple encrypted messages using the
4311 same decryption mechanism and decryption key. The decryption mechanism can be either an
4312 authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.
4313 Cryptoki provides the following functions for message-based decryption.

4314 5.11.1 C_MessageDecryptInit

```

4315 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit)(
4316     CK_SESSION_HANDLE hSession,
4317     CK_MECHANISM_PTR pMechanism,
4318     CK_OBJECT_HANDLE hKey
4319 );

```

4320 **C_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or
4321 more decryption operations that use the same decryption mechanism and decryption key. *hSession* is
4322 the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the
4323 decryption key.

4324 The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption,
4325 MUST be CK_TRUE.

4326 After calling **C_MessageDecryptInit**, the application can either call **C_DecryptMessage** to decrypt an
4327 encrypted message in a single part; or call **C_DecryptMessageBegin**, followed by
4328 **C_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This
4329 may be repeated several times. The message-based decryption process is active until the application
4330 uses a call to **C_MessageDecryptFinal** to finish the message-based decryption process.

4331 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4332 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4333 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4334 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4335 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4336 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4337 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4338 CKR_OPERATION_CANCEL_FAILED.

4339 5.11.2 C_DecryptMessage

```
4340 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (  
4341     CK_SESSION_HANDLE hSession,  
4342     CK_VOID_PTR pParameter,  
4343     CK_ULONG ulParameterLen,  
4344     CK_BYTE_PTR pAssociatedData,  
4345     CK_ULONG ulAssociatedDataLen,  
4346     CK_BYTE_PTR pCiphertext,  
4347     CK_ULONG ulCiphertextLen,  
4348     CK_BYTE_PTR pPlaintext,  
4349     CK_ULONG_PTR pulPlaintextLen  
4350 );
```

4351 **C_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session's handle;
4352 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption
4353 operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD
4354 mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted
4355 message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to
4356 the location that holds the length of the recovered message.

4357 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
4358 **C_EncryptMessage**, *pParameter* is always an input parameter.

4359 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4360 should be set to (NULL, 0).

4361 **C_DecryptMessage** uses the convention described in Section 5.2 on producing output.

4362 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**. A call
4363 to **C_DecryptMessage** begins and terminates a message decryption operation.

4364 **C_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

4365 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to
4366 the same location.

4367 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4368 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4369 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
4370 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.

4371 For most mechanisms, **C_DecryptMessage** is equivalent to **C_DecryptMessageBegin** followed by a
4372 sequence of **C_DecryptMessageNext** operations.

4373 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4374 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4375 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4376 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4377 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4378 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4379 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4380 CKR_OPERATION_CANCEL_FAILED.

5.11.3 C_DecryptMessageBegin

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
);
```

C_DecryptMessageBegin begins a multiple-part message decryption operation. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C_EncryptMessageBegin**, *pParameter* is always an input parameter.

If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

After calling **C_DecryptMessageBegin**, the application should call **C_DecryptMessageNext** one or more times to decrypt the encrypted message in multiple parts. The message decryption operation is active until the application uses a call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or multiple parts), the application MUST call **C_DecryptMessage** or **C_DecryptMessageBegin** 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_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.11.4 C_DecryptMessageNext

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pCiphertextPart,
    CK_ULONG ulCiphertextPartLen,
    CK_BYTE_PTR pPlaintextPart,
    CK_ULONG_PTR pulPlaintextPartLen,
    CK_FLAGS flags
);
```

C_DecryptMessageNext continues a multiple-part message decryption operation, processing another encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart* points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to follow, or set to CKF_END_OF_MESSAGE if this is the last ciphertext part.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C_EncryptMessageNext**, *pParameter* is always an input parameter.

C_DecryptMessageNext uses the convention described in Section 5.2 on producing output.

The message decryption operation MUST have been started with **C_DecryptMessageBegin**. This function may be called any number of times in succession. A call to **C_DecryptMessageNext** with flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message decryption operation. A call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE always

4432 terminates the active message decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4433 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
4434 plaintext.

4435 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart*
4436 point to the same location.

4437 Although the last **C_DecryptMessageNext** call ends the decryption of a message, it does not finish the
4438 message-based decryption process. Additional **C_DecryptMessage** or **C_DecryptMessageBegin** and
4439 **C_DecryptMessageNext** calls may be made on the session.

4440 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4441 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned by
4442 the last **C_DecryptMessageNext** call.

4443 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
4444 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned by the last
4445 **C_DecryptMessageNext** call.

4446 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4447 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4448 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4449 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4450 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4451 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4452 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4453 5.11.5 C_MessageDecryptFinal

```
4454 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal) (  
4455     CK_SESSION_HANDLE hSession  
4456 );
```

4457 **C_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's
4458 handle.

4459 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**.

4460 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4461 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4462 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4463 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4464 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4465 5.12 Message digesting functions

4466 Cryptoki provides the following functions for digesting data:

4467 5.12.1 C_DigestInit

```
4468 CK_DECLARE_FUNCTION(CK_RV, C_DigestInit) (  
4469     CK_SESSION_HANDLE hSession,  
4470     CK_MECHANISM_PTR pMechanism  
4471 );
```

4472 **C_DigestInit** initializes a message-digesting operation. *hSession* is the session's handle; *pMechanism*
4473 points to the digesting mechanism.

4474 After calling **C_DigestInit**, the application can either call **C_Digest** to digest data in a single part; or call
4475 **C_DigestUpdate** zero or more times, followed by **C_DigestFinal**, to digest data in multiple parts. The
4476 message-digesting operation is active until the application uses a call to **C_Digest** or **C_DigestFinal** to
4477 *actually obtain* the message digest. To process additional data (in single or multiple parts), the
4478 application MUST call **C_DigestInit** again.

4479 **C_DigestInit** can be called with *pMechanism* set to `NULL_PTR` to terminate an active message-digesting
4480 operation. If an operation has been initialized and it cannot be cancelled,
4481 `CKR_OPERATION_CANCEL_FAILED` must be returned.

4482 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,
4483 `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`,
4484 `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`,
4485 `CKR_HOST_MEMORY`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`,
4486 `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_PIN_EXPIRED`, `CKR_SESSION_CLOSED`,
4487 `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`,
4488 `CKR_OPERATION_CANCEL_FAILED`.

4489 Example: see **C_DigestFinal**.

4490 5.12.2 C_Digest

```
4491 CK_DECLARE_FUNCTION(CK_RV, C_Digest) (  
4492     CK_SESSION_HANDLE hSession,  
4493     CK_BYTE_PTR pData,  
4494     CK_ULONG ulDataLen,  
4495     CK_BYTE_PTR pDigest,  
4496     CK_ULONG_PTR pulDigestLen  
4497 );
```

4498 **C_Digest** digests data in a single part. *hSession* is the session's handle, *pData* points to the data;
4499 *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest;
4500 *pulDigestLen* points to the location that holds the length of the message digest.

4501 **C_Digest** uses the convention described in Section 5.2 on producing output.

4502 The digest operation **MUST** have been initialized with **C_DigestInit**. A call to **C_Digest** always
4503 terminates the active digest operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful
4504 call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the message
4505 digest.

4506 **C_Digest** cannot be used to terminate a multi-part operation, and **MUST** be called after **C_DigestInit**
4507 without intervening **C_DigestUpdate** calls.

4508 The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the
4509 same location.

4510 **C_Digest** is equivalent to a sequence of **C_DigestUpdate** operations followed by **C_DigestFinal**.

4511 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
4512 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`,
4513 `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`,
4514 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`,
4515 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

4516 Example: see **C_DigestFinal** for an example of similar functions.

4517 5.12.3 C_DigestUpdate

```
4518 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (  
4519     CK_SESSION_HANDLE hSession,  
4520     CK_BYTE_PTR pPart,  
4521     CK_ULONG ulPartLen  
4522 );
```

4523 **C_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.
4524 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4525 The message-digesting operation **MUST** have been initialized with **C_DigestInit**. Calls to this function
4526 and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate**
4527 which results in an error terminates the current digest operation.

4528 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4529 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4530 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4531 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4532 CKR_SESSION_HANDLE_INVALID.

4533 Example: see **C_DigestFinal**.

4534 5.12.4 C_DigestKey

```
4535 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (  
4536     CK_SESSION_HANDLE hSession,  
4537     CK_OBJECT_HANDLE hKey  
4538 );
```

4539 **C_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret
4540 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

4541 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4542 and **C_DigestUpdate** may be interspersed any number of times in any order.

4543 If the value of the supplied key cannot be digested purely for some reason related to its length,
4544 **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

4545 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4546 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4547 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
4548 CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK,
4549 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4550 Example: see **C_DigestFinal**.

4551 5.12.5 C_DigestFinal

```
4552 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (  
4553     CK_SESSION_HANDLE hSession,  
4554     CK_BYTE_PTR pDigest,  
4555     CK_ULONG_PTR pulDigestLen  
4556 );
```

4557 **C_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.
4558 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;
4559 *pulDigestLen* points to the location that holds the length of the message digest.

4560 **C_DigestFinal** uses the convention described in Section 5.2 on producing output.

4561 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always
4562 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4563 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message
4564 digest.

4565 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4566 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4567 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4568 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4569 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4570 Example:

```
4571 CK_SESSION_HANDLE hSession;  
4572 CK_OBJECT_HANDLE hKey;  
4573 CK_MECHANISM mechanism = {  
4574     CKM_MD5, NULL_PTR, 0
```



```

4575 };
4576 CK_BYTE data[] = {...};
4577 CK_BYTE digest[16];
4578 CK_ULONG ulDigestLen;
4579 CK_RV rv;
4580
4581 .
4582 .
4583 rv = C_DigestInit(hSession, &mechanism);
4584 if (rv != CKR_OK) {
4585     .
4586     .
4587 }
4588
4589 rv = C_DigestUpdate(hSession, data, sizeof(data));
4590 if (rv != CKR_OK) {
4591     .
4592     .
4593 }
4594
4595 rv = C_DigestKey(hSession, hKey);
4596 if (rv != CKR_OK) {
4597     .
4598     .
4599 }
4600
4601 ulDigestLen = sizeof(digest);
4602 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
4603 .
4604 .

```

5.13 Signing and MACing functions

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

5.13.1 C_SignInit

```

4609 CK_DECLARE_FUNCTION(CK_RV, C_SignInit)(
4610     CK_SESSION_HANDLE hSession,
4611     CK_MECHANISM_PTR pMechanism,
4612     CK_OBJECT_HANDLE hKey
4613 );

```

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.

4617 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4618 appendix, MUST be CK_TRUE.

4619 After calling **C_SignInit**, the application can either call **C_Sign** to sign in a single part; or call
4620 **C_SignUpdate** one or more times, followed by **C_SignFinal**, to sign data in multiple parts. The signature
4621 operation is active until the application uses a call to **C_Sign** or **C_SignFinal** to *actually obtain* the
4622 signature. To process additional data (in single or multiple parts), the application MUST call **C_SignInit**
4623 again.

4624 **C_SignInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature operation.
4625 If an operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
4626 must be returned.

4627 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4628 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4629 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4630 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4631 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4632 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4633 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4634 CKR_OPERATION_CANCEL_FAILED.

4635 Example: see **C_SignFinal**.

4636 5.13.2 C_Sign

```
4637 CK_DECLARE_FUNCTION(CK_RV, C_Sign) (  
4638     CK_SESSION_HANDLE hSession,  
4639     CK_BYTE_PTR pData,  
4640     CK_ULONG ulDataLen,  
4641     CK_BYTE_PTR pSignature,  
4642     CK_ULONG_PTR pulSignatureLen  
4643 );
```

4644 **C_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the
4645 session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the
4646 location that receives the signature; *pulSignatureLen* points to the location that holds the length of the
4647 signature.

4648 **C_Sign** uses the convention described in Section 5.2 on producing output.

4649 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_Sign** always terminates
4650 the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*,
4651 one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4652 **C_Sign** cannot be used to terminate a multi-part operation, and MUST be called after **C_SignInit** without
4653 intervening **C_SignUpdate** calls.

4654 For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by
4655 **C_SignFinal**.

4656 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4657 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4658 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4659 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4660 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4661 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4662 CKR_TOKEN_RESOURCE_EXCEEDED.

4663 Example: see **C_SignFinal** for an example of similar functions.

5.13.3 C_SignUpdate

```
CK_DECLARE_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, CKR_TOKEN_RESOURCE_EXCEEDED.

Example: see **C_SignFinal**.

5.13.4 C_SignFinal

```
CK_DECLARE_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 5.2 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, CKR_TOKEN_RESOURCE_EXCEEDED.

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;
```

```

4712
4713 .
4714 .
4715 rv = C_SignInit(hSession, &mechanism, hKey);
4716 if (rv == CKR_OK) {
4717     rv = C_SignUpdate(hSession, data, sizeof(data));
4718     .
4719     .
4720     ulMacLen = sizeof(mac);
4721     rv = C_SignFinal(hSession, mac, &ulMacLen);
4722     .
4723     .
4724 }

```

5.13.5 C_SignRecoverInit

```

4725
4726 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit) (
4727     CK_SESSION_HANDLE hSession,
4728     CK_MECHANISM_PTR pMechanism,
4729     CK_OBJECT_HANDLE hKey
4730 );

```

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.

C_SignRecoverInit can be called with *pMechanism* set to NULL_PTR to terminate an active signature with data recovery operation. If an active operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

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, CKR_OPERATION_CANCEL_FAILED.

Example: see **C_SignRecover**.

5.13.6 C_SignRecover

```

4753 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (
4754     CK_SESSION_HANDLE hSession,
4755     CK_BYTE_PTR pData,
4756     CK_ULONG ulDataLen,
4757     CK_BYTE_PTR pSignature,

```

```
4758     CK_ULONG_PTR pulSignatureLen
4759 );
```

4760 **C_SignRecover** signs data in a single operation, where the data can be recovered from the signature.
4761 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4762 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that
4763 holds the length of the signature.

4764 **C_SignRecover** uses the convention described in Section 5.2 on producing output.

4765 The signing operation MUST have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover**
4766 always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4767 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4768 signature.

4769 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4770 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4771 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4772 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4773 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4774 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4775 CKR_TOKEN_RESOURCE_EXCEEDED.

4776 Example:

```
4777 CK_SESSION_HANDLE hSession;
4778 CK_OBJECT_HANDLE hKey;
4779 CK_MECHANISM mechanism = {
4780     CKM_RSA_9796, NULL_PTR, 0
4781 };
4782 CK_BYTE data[] = {...};
4783 CK_BYTE signature[128];
4784 CK_ULONG ulSignatureLen;
4785 CK_RV rv;
4786
4787 .
4788 .
4789 rv = C_SignRecoverInit(hSession, &mechanism, hKey);
4790 if (rv == CKR_OK) {
4791     ulSignatureLen = sizeof(signature);
4792     rv = C_SignRecover(
4793         hSession, data, sizeof(data), signature, &ulSignatureLen);
4794     if (rv == CKR_OK) {
4795         .
4796         .
4797     }
4798 }
4799
```

4800 5.14 Message-based signing and MACing functions

4801 Message-based signature refers to the process of signing multiple messages using the same signature
4802 mechanism and signature key.

4803 Cryptoki provides the following functions for for signing messages (for the purposes of Cryptoki, these
4804 operations also encompass message authentication codes).

4805 5.14.1 C_MessageSignInit

```
4806 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit) (  
4807     CK_SESSION_HANDLE hSession,  
4808     CK_MECHANISM_PTR pMechanism,  
4809     CK_OBJECT_HANDLE hKey  
4810 );
```

4811 **C_MessageSignInit** initializes a message-based signature process, preparing a session for one or more
4812 signature operations (where the signature is an appendix to the data) that use the same signature
4813 mechanism and signature key. *hSession* is the session's handle; *pMechanism* points to the signature
4814 mechanism; *hKey* is the handle of the signature key.

4815 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4816 appendix, MUST be CK_TRUE.

4817 After calling **C_MessageSignInit**, the application can either call **C_SignMessage** to sign a message in a
4818 single part; or call **C_SignMessageBegin**, followed by **C_SignMessageNext** one or more times, to sign
4819 a message in multiple parts. This may be repeated several times. The message-based signature process
4820 is active until the application calls **C_MessageSignFinal** to finish the message-based signature process.

4821 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4822 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4823 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4824 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4825 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4826 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4827 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4828 5.14.2 C_SignMessage

```
4829 CK_DECLARE_FUNCTION(CK_RV, C_SignMessage) (  
4830     CK_SESSION_HANDLE hSession,  
4831     CK_VOID_PTR pParameter,  
4832     CK_ULONG ulParameterLen,  
4833     CK_BYTE_PTR pData,  
4834     CK_ULONG ulDataLen,  
4835     CK_BYTE_PTR pSignature,  
4836     CK_ULONG_PTR pulSignatureLen  
4837 );
```

4838 **C_SignMessage** signs a message in a single part, where the signature is an appendix to the message.
4839 **C_MessageSignInit** must previously been called on the session. *hSession* is the session's handle;
4840 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature
4841 operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location
4842 that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

4843 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
4844 input or an output parameter.

4845 **C_SignMessage** uses the convention described in Section 5.2 on producing output.

4846 The message-based signing process MUST have been initialized with **C_MessageSignInit**. A call to
4847 **C_SignMessage** begins and terminates a message signing operation unless it returns

4848 CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a
 4849 successful call (i.e., one which returns CKR_OK).

4850 **C_SignMessage** cannot be called in the middle of a multi-part message signing operation.

4851 **C_SignMessage** does not finish the message-based signing process. Additional **C_SignMessage** or
 4852 **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

4853 For most mechanisms, **C_SignMessage** is equivalent to **C_SignMessageBegin** followed by a sequence
 4854 of **C_SignMessageNext** operations.

4855 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4856 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
 4857 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4858 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4859 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4860 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
 4861 CKR_TOKEN_RESOURCE_EXCEEDED.

4862 5.14.3 C_SignMessageBegin

```
4863 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (
4864     CK_SESSION_HANDLE hSession,
4865     CK_VOID_PTR pParameter,
4866     CK_ULONG ulParameterLen
4867 );
```

4868 **C_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an
 4869 appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is
 4870 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
 4871 the message signature operation.

4872 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
 4873 input or an output parameter.

4874 After calling **C_SignMessageBegin**, the application should call **C_SignMessageNext** one or more times
 4875 to sign the message in multiple parts. The message signature operation is active until the application
 4876 uses a call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.
 4877 To process additional messages (in single or multiple parts), the application MUST call **C_SignMessage**
 4878 or **C_SignMessageBegin** again.

4879 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4880 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4881 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4882 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
 4883 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
 4884 CKR_TOKEN_RESOURCE_EXCEEDED.

4885 5.14.4 C_SignMessageNext

```
4886 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (
4887     CK_SESSION_HANDLE hSession,
4888     CK_VOID_PTR pParameter,
4889     CK_ULONG ulParameterLen,
4890     CK_BYTE_PTR pDataPart,
4891     CK_ULONG ulDataPartLen,
4892     CK_BYTE_PTR pSignature,
4893     CK_ULONG_PTR pulSignatureLen
```

4894);

4895 **C_SignMessageNext** continues a multiple-part message signature operation, processing another data
4896 part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the
4897 session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any
4898 mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the
4899 data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the
4900 location that holds the length of the signature.

4901 The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NULL
4902 value (to receive the signature length) if this is the last data part.

4903 **C_SignMessageNext** uses the convention described in Section 5.2 on producing output.

4904 The message signing operation MUST have been started with **C_SignMessageBegin**. This function may
4905 be called any number of times in succession. A call to **C_SignMessageNext** with a NULL
4906 *pulSignatureLen* which results in an error terminates the current message signature operation. A call to
4907 **C_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing
4908 operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to
4909 hold the signature, or is a successful call (i.e., one which returns CKR_OK).

4910 Although the last **C_SignMessageNext** call ends the signing of a message, it does not finish the
4911 message-based signing process. Additional **C_SignMessage** or **C_SignMessageBegin** and
4912 **C_SignMessageNext** calls may be made on the session.

4913 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4914 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4915 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4916 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4917 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4918 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4919 CKR_TOKEN_RESOURCE_EXCEEDED.

4920 5.14.5 C_MessageSignFinal

```
4921 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) (  
4922     CK_SESSION_HANDLE hSession  
4923 );
```

4924 **C_MessageSignFinal** finishes a message-based signing process. *hSession* is the session's handle.

4925 The message-based signing process MUST have been initialized with **C_MessageSignInit**.

4926 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4927 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4928 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4929 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4930 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4931 CKR_TOKEN_RESOURCE_EXCEEDED.

4932 5.15 Functions for verifying signatures and MACs

4933 Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki,
4934 these operations also encompass message authentication codes):

4935 5.15.1 C_VerifyInit

```
4936 CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (  
4937     CK_SESSION_HANDLE hSession,  
4938     CK_MECHANISM_PTR pMechanism,  
4939     CK_OBJECT_HANDLE hKey  
4940 );
```


4941 **C_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*
4942 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
4943 *hKey* is the handle of the verification key.

4944 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
4945 where the signature is an appendix to the data, MUST be CK_TRUE.

4946 After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single
4947 part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data
4948 in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.
4949 To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

4950 **C_VerifyInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
4951 operation. If an active operation has been initialized and it cannot be cancelled,
4952 CKR_OPERATION_CANCEL_FAILED must be returned.

4953 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4954 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4955 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4956 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4957 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4958 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4959 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4960 CKR_OPERATION_CANCEL_FAILED.

4961 Example: see **C_VerifyFinal**.

4962 5.15.2 C_Verify

```
4963 CK_DECLARE_FUNCTION(CK_RV, C_Verify) (  
4964     CK_SESSION_HANDLE hSession,  
4965     CK_BYTE_PTR pData,  
4966     CK_ULONG ulDataLen,  
4967     CK_BYTE_PTR pSignature,  
4968     CK_ULONG ulSignatureLen  
4969 );
```

4970 **C_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.
4971 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4972 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

4973 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_Verify** always
4974 terminates the active verification operation.

4975 A successful call to **C_Verify** should return either the value CKR_OK (indicating that the supplied
4976 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
4977 signature can be seen to be invalid purely on the basis of its length, then
4978 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active signing operation
4979 is terminated.

4980 **C_Verify** cannot be used to terminate a multi-part operation, and MUST be called after **C_VerifyInit**
4981 without intervening **C_VerifyUpdate** calls.

4982 For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by
4983 **C_VerifyFinal**.

4984 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
4985 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4986 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4987 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4988 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
4989 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

4990 Example: see **C_VerifyFinal** for an example of similar functions.

4991 5.15.3 C_VerifyUpdate

```
4992 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate) (  
4993     CK_SESSION_HANDLE hSession,  
4994     CK_BYTE_PTR pPart,  
4995     CK_ULONG ulPartLen  
4996 );
```

4997 **C_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*
4998 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4999 The verification operation MUST have been initialized with **C_VerifyInit**. This function may be called any
5000 number of times in succession. A call to **C_VerifyUpdate** which results in an error terminates the current
5001 verification operation.

5002 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5003 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5004 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5005 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5006 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5007 CKR_TOKEN_RESOURCE_EXCEEDED.

5008 Example: see **C_VerifyFinal**.

5009 5.15.4 C_VerifyFinal

```
5010 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal) (  
5011     CK_SESSION_HANDLE hSession,  
5012     CK_BYTE_PTR pSignature,  
5013     CK_ULONG ulSignatureLen  
5014 );
```

5015 **C_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the
5016 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5017 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always
5018 terminates the active verification operation.

5019 A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied
5020 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
5021 signature can be seen to be invalid purely on the basis of its length, then
5022 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying
5023 operation is terminated.

5024 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5025 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5026 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5027 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5028 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5029 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5030 Example:

```
5031 CK_SESSION_HANDLE hSession;  
5032 CK_OBJECT_HANDLE hKey;  
5033 CK_MECHANISM mechanism = {  
5034     CKM_DES_MAC, NULL_PTR, 0  
5035 };  
5036 CK_BYTE data[] = {...};  
5037 CK_BYTE mac[4];  
5038 CK_RV rv;
```



```

5039
5040 .
5041 .
5042 rv = C_VerifyInit(hSession, &mechanism, hKey);
5043 if (rv == CKR_OK) {
5044     rv = C_VerifyUpdate(hSession, data, sizeof(data));
5045     .
5046     .
5047     rv = C_VerifyFinal(hSession, mac, sizeof(mac));
5048     .
5049     .
5050 }

```

5051 5.15.5 C_VerifyRecoverInit

```

5052 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit)(
5053     CK_SESSION_HANDLE hSession,
5054     CK_MECHANISM_PTR pMechanism,
5055     CK_OBJECT_HANDLE hKey
5056 );

```

5057 **C_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the
5058 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
5059 verification mechanism; *hKey* is the handle of the verification key.

5060 The **CKA_VERIFY_RECOVER** attribute of the verification key, which indicates whether the key supports
5061 verification where the data is recovered from the signature, **MUST** be CK_TRUE.

5062 After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on
5063 data in a single part. The verification operation is active until the application uses a call to
5064 **C_VerifyRecover** to actually obtain the recovered message.

5065 **C_VerifyRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5066 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,
5067 CKR_OPERATION_CANCEL_FAILED must be returned.

5068 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5069 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5070 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5071 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5072 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5073 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5074 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5075 CKR_OPERATION_CANCEL_FAILED.

5076 Example: see **C_VerifyRecover**.

5077 5.15.6 C_VerifyRecover

```

5078 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover)(
5079     CK_SESSION_HANDLE hSession,
5080     CK_BYTE_PTR pSignature,
5081     CK_ULONG ulSignatureLen,
5082     CK_BYTE_PTR pData,
5083     CK_ULONG_PTR pulDataLen
5084 );

```

5085 **C_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the
 5086 signature. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the
 5087 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*
 5088 points to the location that holds the length of the recovered data.

5089 **C_VerifyRecover** uses the convention described in Section 5.2 on producing output.

5090 The verification operation MUST have been initialized with **C_VerifyRecoverInit**. A call to
 5091 **C_VerifyRecover** always terminates the active verification operation unless it returns
 5092 CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the
 5093 length of the buffer needed to hold the recovered data.

5094 A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the
 5095 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
 5096 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
 5097 CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID
 5098 and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code
 5099 CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never
 5100 return CKR_BUFFER_TOO_SMALL.

5101 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 5102 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
 5103 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 5104 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 5105 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 5106 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID,
 5107 CKR_TOKEN_RESOURCE_EXCEEDED.

5108 Example:

```

5109 CK_SESSION_HANDLE hSession;
5110 CK_OBJECT_HANDLE hKey;
5111 CK_MECHANISM mechanism = {
5112     CKM_RSA_9796, NULL_PTR, 0
5113 };
5114 CK_BYTE data[] = {...};
5115 CK_ULONG ulDataLen;
5116 CK_BYTE signature[128];
5117 CK_RV rv;
5118
5119 .
5120 .
5121 rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
5122 if (rv == CKR_OK) {
5123     ulDataLen = sizeof(data);
5124     rv = C_VerifyRecover(
5125         hSession, signature, sizeof(signature), data, &ulDataLen);
5126     .
5127     .
5128 }

```

5.16 Message-based functions for verifying signatures and MACs

Message-based verification refers to the process of verifying signatures on multiple messages using the same verification mechanism and verification key.

Cryptoki provides the following functions for verifying signatures on messages (for the purposes of Cryptoki, these operations also encompass message authentication codes).

5.16.1 C_MessageVerifyInit

```
CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit) (
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_MessageVerifyInit initializes a message-based verification process, preparing a session for one or more verification operations (where the signature is an appendix to the data) that use the same verification mechanism and verification key. *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_MessageVerifyInit**, the application can either call **C_VerifyMessage** to verify a signature on a message in a single part; or call **C_VerifyMessageBegin**, followed by **C_VerifyMessageNext** one or more times, to verify a signature on a message in multiple parts. This may be repeated several times. The message-based verification process is active until the application calls **C_MessageVerifyFinal** to finish the message-based verification process.

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**.

5.16.2 C_VerifyMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG ulSignatureLen
);
```

C_VerifyMessage verifies a signature on a message in a single part operation, where the signature is an appendix to the data. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

Unlike the *pParameter* parameter of **C_SignMessage**, *pParameter* is always an input parameter.

5174 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**. A call to
5175 **C_VerifyMessage** starts and terminates a message verification operation.

5176 A successful call to **C_VerifyMessage** should return either the value CKR_OK (indicating that the
5177 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5178 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5179 CKR_SIGNATURE_LEN_RANGE should be returned.

5180 **C_VerifyMessage** does not finish the message-based verification process. Additional **C_VerifyMessage**
5181 or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

5182 For most mechanisms, **C_VerifyMessage** is equivalent to **C_VerifyMessageBegin** followed by a
5183 sequence of **C_VerifyMessageNext** operations.

5184 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
5185 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5186 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5187 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5188 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5189 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5190 5.16.3 C_VerifyMessageBegin

```
5191 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (  
5192     CK_SESSION_HANDLE hSession,  
5193     CK_VOID_PTR pParameter,  
5194     CK_ULONG ulParameterLen  
5195 );
```

5196 **C_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an
5197 appendix to the message. **C_MessageVerifyInit** must previously been called on the session. *hSession* is
5198 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
5199 the message verification operation.

5200 Unlike the *pParameter* parameter of **C_SignMessageBegin**, *pParameter* is always an input parameter.

5201 After calling **C_VerifyMessageBegin**, the application should call **C_VerifyMessageNext** one or more
5202 times to verify a signature on a message in multiple parts. The message verification operation is active
5203 until the application calls **C_VerifyMessageNext** with a non-NULL *pSignature*. To process additional
5204 messages (in single or multiple parts), the application MUST call **C_VerifyMessage** or
5205 **C_VerifyMessageBegin** again.

5206 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5207 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5208 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5209 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5210 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5211 5.16.4 C_VerifyMessageNext

```
5212 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (  
5213     CK_SESSION_HANDLE hSession,  
5214     CK_VOID_PTR pParameter,  
5215     CK_ULONG ulParameterLen,  
5216     CK_BYTE_PTR pDataPart,  
5217     CK_ULONG ulDataPartLen,  
5218     CK_BYTE_PTR pSignature,  
5219     CK_ULONG ulSignatureLen
```

5220);

5221 **C_VerifyMessageNext** continues a multiple-part message verification operation, processing another data
5222 part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the
5223 session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5224 message verification operation, *pPart* points to the data part; *ulPartLen* is the length of the data part;
5225 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5226 The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value
5227 (pointing to the signature to verify) if this is the last data part.

5228 The message verification operation MUST have been started with **C_VerifyMessageBegin**. This function
5229 may be called any number of times in succession. A call to **C_VerifyMessageNext** with a NULL
5230 *pSignature* which results in an error terminates the current message verification operation. A call to
5231 **C_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification
5232 operation.

5233 A successful call to **C_VerifyMessageNext** with a non-NULL *pSignature* should return either the value
5234 CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that
5235 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its
5236 length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active
5237 message verifying operation is terminated.

5238 Although the last **C_VerifyMessageNext** call ends the verification of a message, it does not finish the
5239 message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and
5240 **C_VerifyMessageNext** calls may be made on the session.

5241 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5242 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5243 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5244 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5245 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5246 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5247 5.16.5 C_MessageVerifyFinal

```
5248 CK_DECLARE_FUNCTION(CK_RV,C_MessageVerifyFinal) (  
5249     CK_SESSION_HANDLE hSession  
5250 );
```

5251 **C_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session's handle.

5252 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**.

5253 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5254 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5255 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5256 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5257 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5258 CKR_TOKEN_RESOURCE_EXCEEDED.

5259 5.17 Dual-function cryptographic functions

5260 Cryptoki provides the following functions to perform two cryptographic operations "simultaneously" within
5261 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and
5262 from a token.

5263 5.17.1 C_DigestEncryptUpdate

```
5264 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (  
5265     CK_SESSION_HANDLE hSession,  
5266     CK_BYTE_PTR pPart,
```

```

5267     CK_ULONG ulPartLen,
5268     CK_BYTE_PTR pEncryptedPart,
5269     CK_ULONG_PTR pulEncryptedPartLen
5270 );

```

5271 **C_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another
5272 data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the
5273 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;
5274 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5275 **C_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5276 **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5277 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5278 encrypted part output), then no plaintext is passed to the active digest operation.

5279 Digest and encryption operations **MUST** both be active (they **MUST** have been initialized with
5280 **C_DigestInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5281 succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls
5282 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5283 **C_DigestKey**, however).

5284 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5285 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5286 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
5287 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,
5288 `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

5289 Example:

```

5290 #define BUF_SZ 512
5291
5292 CK_SESSION_HANDLE hSession;
5293 CK_OBJECT_HANDLE hKey;
5294 CK_BYTE iv[8];
5295 CK_MECHANISM digestMechanism = {
5296     CKM_MD5, NULL_PTR, 0
5297 };
5298 CK_MECHANISM encryptionMechanism = {
5299     CKM_DES_ECB, iv, sizeof(iv)
5300 };
5301 CK_BYTE encryptedData[BUF_SZ];
5302 CK_ULONG ulEncryptedDataLen;
5303 CK_BYTE digest[16];
5304 CK_ULONG ulDigestLen;
5305 CK_BYTE data[(2*BUF_SZ)+8];
5306 CK_RV rv;
5307 int i;
5308
5309 .
5310 .
5311 memset(iv, 0, sizeof(iv));
5312 memset(data, 'A', ((2*BUF_SZ)+5));
5313 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);

```

```

5314 if (rv != CKR_OK) {
5315     .
5316     .
5317 }
5318 rv = C_DigestInit(hSession, &digestMechanism);
5319 if (rv != CKR_OK) {
5320     .
5321     .
5322 }
5323
5324 ulEncryptedDataLen = sizeof(encryptedData);
5325 rv = C_DigestEncryptUpdate(
5326     hSession,
5327     &data[0], BUF_SZ,
5328     encryptedData, &ulEncryptedDataLen);
5329 .
5330 .
5331 ulEncryptedDataLen = sizeof(encryptedData);
5332 rv = C_DigestEncryptUpdate(
5333     hSession,
5334     &data[BUF_SZ], BUF_SZ,
5335     encryptedData, &ulEncryptedDataLen);
5336 .
5337 .
5338
5339 /*
5340  * The last portion of the buffer needs to be
5341  * handled with separate calls to deal with
5342  * padding issues in ECB mode
5343  */
5344
5345 /* First, complete the digest on the buffer */
5346 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5347 .
5348 .
5349 ulDigestLen = sizeof(digest);
5350 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5351 .
5352 .
5353
5354 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
5355 for(i=0;i<3;i++)
5356     data[((BUF_SZ*2)+5)+i] = 0x00;

```

```

5357
5358 /* Now, get second-to-last piece of ciphertext */
5359 ulEncryptedDataLen = sizeof(encryptedData);
5360 rv = C_EncryptUpdate(
5361     hSession,
5362     &data[BUF_SZ*2], 8,
5363     encryptedData, &ulEncryptedDataLen);
5364 .
5365 .
5366
5367 /* Get last piece of ciphertext (should have length 0, here) */
5368 ulEncryptedDataLen = sizeof(encryptedData);
5369 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5370 .
5371 .

```

5372 5.17.2 C_DecryptDigestUpdate

```

5373 CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
5374     CK_SESSION_HANDLE hSession,
5375     CK_BYTE_PTR pEncryptedPart,
5376     CK_ULONG ulEncryptedPartLen,
5377     CK_BYTE_PTR pPart,
5378     CK_ULONG_PTR pulPartLen
5379 );

```

5380 **C_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation,
5381 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5382 data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that
5383 receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered
5384 data part.

5385 **C_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a
5386 **C_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because
5387 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire decrypted part
5388 output), then no plaintext is passed to the active digest operation.

5389 Decryption and digesting operations **MUST** both be active (they **MUST** have been initialized with
5390 **C_DecryptInit** and **C_DigestInit**, respectively). This function may be called any number of times in
5391 succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls
5392 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5393 **C_DigestKey**, however).

5394 Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using
5395 **C_DigestEncryptUpdate**, the "inverse function" of **C_DecryptDigestUpdate**. This is because when
5396 **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting
5397 operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the
5398 input passed to the active digesting operation is the *output* of the active decryption operation. This issue
5399 comes up only when the mechanism used for decryption performs padding.

5400 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
5401 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
5402 ciphertext and digest the original plaintext thereby obtained.

5403 After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES
5404 blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** returns exactly 16 bytes of plaintext,

5405 since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
5406 ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

5407 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's
5408 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
5409 decryption and digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2
5410 bytes of plaintext are *not* passed on to be digested.

5411 A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the*
5412 *plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called,
5413 the last 2 bytes of plaintext get passed into the active digesting operation via a **C_DigestUpdate** call.

5414 Because of this, it is critical that when an application uses a padded decryption mechanism with
5415 **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting
5416 operation. *Extreme caution is warranted when using a padded decryption mechanism with*
5417 **C_DecryptDigestUpdate**.

5418 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5419 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5420 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
5421 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5422 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5423 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5424 Example:

```
5425 #define BUF_SZ 512
5426
5427 CK_SESSION_HANDLE hSession;
5428 CK_OBJECT_HANDLE hKey;
5429 CK_BYTE iv[8];
5430 CK_MECHANISM decryptionMechanism = {
5431     CKM_DES_ECB, iv, sizeof(iv)
5432 };
5433 CK_MECHANISM digestMechanism = {
5434     CKM_MD5, NULL_PTR, 0
5435 };
5436 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5437 CK_BYTE digest[16];
5438 CK_ULONG ulDigestLen;
5439 CK_BYTE data[BUF_SZ];
5440 CK_ULONG ulDataLen, ulLastUpdateSize;
5441 CK_RV rv;
5442
5443 .
5444 .
5445 memset(iv, 0, sizeof(iv));
5446 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5447 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
5448 if (rv != CKR_OK) {
5449     .
5450     .
```

```

5451 }
5452 rv = C_DigestInit(hSession, &digestMechanism);
5453 if (rv != CKR_OK) {
5454     .
5455     .
5456 }
5457
5458 ulDataLen = sizeof(data);
5459 rv = C_DecryptDigestUpdate(
5460     hSession,
5461     &encryptedData[0], BUF_SZ,
5462     data, &ulDataLen);
5463 .
5464 .
5465 ulDataLen = sizeof(data);
5466 rv = C_DecryptDigestUpdate(
5467     hSession,
5468     &encryptedData[BUF_SZ], BUF_SZ,
5469     data, &ulDataLen);
5470 .
5471 .
5472
5473 /*
5474  * The last portion of the buffer needs to be handled with
5475  * separate calls to deal with padding issues in ECB mode
5476  */
5477
5478 /* First, complete the decryption of the buffer */
5479 ulLastUpdateSize = sizeof(data);
5480 rv = C_DecryptUpdate(
5481     hSession,
5482     &encryptedData[BUF_SZ*2], 8,
5483     data, &ulLastUpdateSize);
5484 .
5485 .
5486 /* Get last piece of plaintext (should have length 0, here) */
5487 ulDataLen = sizeof(data)-ulLastUpdateSize;
5488 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5489 if (rv != CKR_OK) {
5490     .
5491     .
5492 }
5493

```

```

5494 /* Digest last bit of plaintext */
5495 rv = C_DigestUpdate(hSession, data, 5);
5496 if (rv != CKR_OK) {
5497     .
5498     .
5499 }
5500 ulDigestLen = sizeof(digest);
5501 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5502 if (rv != CKR_OK) {
5503     .
5504     .
5505 }

```

5506 5.17.3 C_SignEncryptUpdate

```

5507 CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate)(
5508     CK_SESSION_HANDLE hSession,
5509     CK_BYTE_PTR pPart,
5510     CK_ULONG ulPartLen,
5511     CK_BYTE_PTR pEncryptedPart,
5512     CK_ULONG_PTR pulEncryptedPartLen
5513 );

```

5514 **C_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation,
5515 processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is
5516 the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted
5517 data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5518 **C_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5519 **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5520 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5521 encrypted part output), then no plaintext is passed to the active signing operation.

5522 Signature and encryption operations **MUST** both be active (they **MUST** have been initialized with
5523 **C_SignInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5524 succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

5525 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5526 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5527 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
5528 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,
5529 `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`,
5530 `CKR_USER_NOT_LOGGED_IN`.

5531 Example:

```

5532 #define BUF_SZ 512
5533
5534 CK_SESSION_HANDLE hSession;
5535 CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
5536 CK_BYTE iv[8];
5537 CK_MECHANISM signMechanism = {
5538     CKM_DES_MAC, NULL_PTR, 0
5539 };

```

```

5540 CK_MECHANISM encryptionMechanism = {
5541     CKM_DES_ECB, iv, sizeof(iv)
5542 };
5543 CK_BYTE encryptedData[BUF_SZ];
5544 CK_ULONG ulEncryptedDataLen;
5545 CK_BYTE MAC[4];
5546 CK_ULONG ulMacLen;
5547 CK_BYTE data[(2*BUF_SZ)+8];
5548 CK_RV rv;
5549 int i;
5550
5551 .
5552 .
5553 memset(iv, 0, sizeof(iv));
5554 memset(data, 'A', ((2*BUF_SZ)+5));
5555 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
5556 if (rv != CKR_OK) {
5557     .
5558     .
5559 }
5560 rv = C_SignInit(hSession, &signMechanism, hMacKey);
5561 if (rv != CKR_OK) {
5562     .
5563     .
5564 }
5565
5566 ulEncryptedDataLen = sizeof(encryptedData);
5567 rv = C_SignEncryptUpdate(
5568     hSession,
5569     &data[0], BUF_SZ,
5570     encryptedData, &ulEncryptedDataLen);
5571 .
5572 .
5573 ulEncryptedDataLen = sizeof(encryptedData);
5574 rv = C_SignEncryptUpdate(
5575     hSession,
5576     &data[BUF_SZ], BUF_SZ,
5577     encryptedData, &ulEncryptedDataLen);
5578 .
5579 .
5580
5581 /*
5582  * The last portion of the buffer needs to be handled with

```

```

5583  * separate calls to deal with padding issues in ECB mode
5584  */
5585
5586  /* First, complete the signature on the buffer */
5587  rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
5588  .
5589  .
5590  ulMacLen = sizeof(MAC);
5591  rv = C_SignFinal(hSession, MAC, &ulMacLen);
5592  .
5593  .
5594
5595  /* Then pad last part with 3 0x00 bytes, and complete encryption */
5596  for(i=0;i<3;i++)
5597      data[((BUF_SZ*2)+5)+i] = 0x00;
5598
5599  /* Now, get second-to-last piece of ciphertext */
5600  ulEncryptedDataLen = sizeof(encryptedData);
5601  rv = C_EncryptUpdate(
5602      hSession,
5603      &data[BUF_SZ*2], 8,
5604      encryptedData, &ulEncryptedDataLen);
5605  .
5606  .
5607
5608  /* Get last piece of ciphertext (should have length 0, here) */
5609  ulEncryptedDataLen = sizeof(encryptedData);
5610  rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5611  .
5612  .

```

5613 5.17.4 C_DecryptVerifyUpdate

```

5614  CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate)(
5615      CK_SESSION_HANDLE hSession,
5616      CK_BYTE_PTR pEncryptedPart,
5617      CK_ULONG ulEncryptedPartLen,
5618      CK_BYTE_PTR pPart,
5619      CK_ULONG_PTR pulPartLen
5620  );

```

5621 **C_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation,
5622 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5623 data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the
5624 recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

5625 **C_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a
5626 **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because

5627 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part
5628 output), then no plaintext is passed to the active verification operation.

5629 Decryption and signature operations **MUST** both be active (they **MUST** have been initialized with
5630 **C_DecryptInit** and **C_VerifyInit**, respectively). This function may be called any number of times in
5631 succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

5632 Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using
5633 **C_SignEncryptUpdate**, the “inverse function” of **C_DecryptVerifyUpdate**. This is because when
5634 **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation
5635 and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed
5636 to the active verifying operation is the *output* of the active decryption operation. This issue comes up only
5637 when the mechanism used for decryption performs padding.

5638 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
5639 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
5640 ciphertext and verify a signature on the original plaintext thereby obtained.

5641 After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3
5642 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of
5643 plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
5644 ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

5645 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's
5646 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
5647 decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2
5648 bytes of plaintext are *not* passed on to the verification mechanism.

5649 A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature
5650 on *the first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C_VerifyFinal** is
5651 called, the last 2 bytes of plaintext get passed into the active verification operation via a **C_VerifyUpdate**
5652 call.

5653 Because of this, it is critical that when an application uses a padded decryption mechanism with
5654 **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active
5655 verification operation. *Extreme caution is warranted when using a padded decryption mechanism with*
5656 **C_DecryptVerifyUpdate**.

5657 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5658 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5659 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_ENCRYPTED_DATA_INVALID`,
5660 `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`,
5661 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`,
5662 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

5663 Example:

```

5664 #define BUF_SZ 512
5665
5666 CK_SESSION_HANDLE hSession;
5667 CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
5668 CK_BYTE iv[8];
5669 CK_MECHANISM decryptionMechanism = {
5670     CKM_DES_ECB, iv, sizeof(iv)
5671 };
5672 CK_MECHANISM verifyMechanism = {
5673     CKM_DES_MAC, NULL_PTR, 0
5674 };
5675 CK_BYTE encryptedData[(2*BUF_SZ)+8];

```

```

5676 CK_BYTE MAC[4];
5677 CK_ULONG ulMacLen;
5678 CK_BYTE data[BUF_SZ];
5679 CK_ULONG ulDataLen, ulLastUpdateSize;
5680 CK_RV rv;
5681
5682 .
5683 .
5684 memset(iv, 0, sizeof(iv));
5685 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5686 rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
5687 if (rv != CKR_OK) {
5688     .
5689     .
5690 }
5691 rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
5692 if (rv != CKR_OK){
5693     .
5694     .
5695 }
5696
5697 ulDataLen = sizeof(data);
5698 rv = C_DecryptVerifyUpdate(
5699     hSession,
5700     &encryptedData[0], BUF_SZ,
5701     data, &ulDataLen);
5702 .
5703 .
5704 ulDataLen = sizeof(data);
5705 rv = C_DecryptVerifyUpdate(
5706     hSession,
5707     &encryptedData[BUF_SZ], BUF_SZ,
5708     data, &ulDataLen);
5709 .
5710 .
5711
5712 /*
5713  * The last portion of the buffer needs to be handled with
5714  * separate calls to deal with padding issues in ECB mode
5715  */
5716
5717 /* First, complete the decryption of the buffer */
5718 ulLastUpdateSize = sizeof(data);

```

```

5719 rv = C_DecryptUpdate(
5720     hSession,
5721     &encryptedData[BUF_SZ*2], 8,
5722     data, &ulLastUpdateSize);
5723 .
5724 .
5725 /* Get last little piece of plaintext. Should have length 0 */
5726 ulDataLen = sizeof(data)-ulLastUpdateSize;
5727 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5728 if (rv != CKR_OK) {
5729     .
5730     .
5731 }
5732
5733 /* Send last bit of plaintext to verification operation */
5734 rv = C_VerifyUpdate(hSession, data, 5);
5735 if (rv != CKR_OK) {
5736     .
5737     .
5738 }
5739 rv = C_VerifyFinal(hSession, MAC, ulMacLen);
5740 if (rv == CKR_SIGNATURE_INVALID) {
5741     .
5742     .
5743 }

```

5744 5.18 Key management functions

5745 Cryptoki provides the following functions for key management:

5746 5.18.1 C_GenerateKey

```

5747 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey) (
5748     CK_SESSION_HANDLE hSession
5749     CK_MECHANISM_PTR pMechanism,
5750     CK_ATTRIBUTE_PTR pTemplate,
5751     CK_ULONG ulCount,
5752     CK_OBJECT_HANDLE_PTR phKey
5753 );

```

5754 **C_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is
5755 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template
5756 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*
5757 points to the location that receives the handle of the new key or set of domain parameters.

5758 If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the
5759 value CKO_DOMAIN_PARAMETERS; otherwise, it will have the value CKO_SECRET_KEY.

5760 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the
5761 template does not need to supply a key type. If it does supply a key type which is inconsistent with the

5762 generation mechanism, **C_GenerateKey** fails and returns the error code
5763 CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

5764 If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without
5765 creating an object.

5766 The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to
5767 CK_TRUE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
5768 assigned (See Section 4.4.1).

5769 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5770 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5771 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
5772 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5773 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
5774 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5775 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5776 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5777 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5778 CKR_USER_NOT_LOGGED_IN.

5779 Example:

```
5780 CK_SESSION_HANDLE hSession;
5781 CK_OBJECT_HANDLE hKey;
5782 CK_MECHANISM mechanism = {
5783     CKM_DES_KEY_GEN, NULL_PTR, 0
5784 };
5785 CK_RV rv;
5786
5787 .
5788 .
5789 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
5790 if (rv == CKR_OK) {
5791     .
5792     .
5793 }
```

5794 5.18.2 C_GenerateKeyPair

```
5795 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair) (
5796     CK_SESSION_HANDLE hSession,
5797     CK_MECHANISM_PTR pMechanism,
5798     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
5799     CK_ULONG ulPublicKeyAttributeCount,
5800     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
5801     CK_ULONG ulPrivateKeyAttributeCount,
5802     CK_OBJECT_HANDLE_PTR phPublicKey,
5803     CK_OBJECT_HANDLE_PTR phPrivateKey
5804 );
```

5805 **C_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the
5806 session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to
5807 the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key
5808 template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is
5809 the number of attributes in the private-key template; *phPublicKey* points to the location that receives the

5810 handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new
5811 private key.

5812 Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates
5813 do not need to supply key types. If one of the templates does supply a key type which is inconsistent with
5814 the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code
5815 CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

5816 If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return
5817 without creating any key objects.

5818 A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no
5819 keys; or it can succeed, and create a matching public/private key pair.

5820 The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL**
5821 attributes set to CK_TRUE. In addition, the key objects created will both have values for
5822 CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).

5823 *Note carefully the order of the arguments to C_GenerateKeyPair. The last two arguments do not have*
5824 *the same order as they did in the original Cryptoki Version 1.0 document. The order of these two*
5825 *arguments has caused some unfortunate confusion.*

5826 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5827 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5828 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
5829 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
5830 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5831 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
5832 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5833 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5834 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5835 CKR_USER_NOT_LOGGED_IN.

5836 Example:

```

5837 CK_SESSION_HANDLE hSession;
5838 CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
5839 CK_MECHANISM mechanism = {
5840     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
5841 };
5842 CK_ULONG modulusBits = 3072;
5843 CK_BYTE publicExponent[] = { 3 };
5844 CK_BYTE subject[] = {...};
5845 CK_BYTE id[] = {123};
5846 CK_BBOOL true = CK_TRUE;
5847 CK_ATTRIBUTE publicKeyTemplate[] = {
5848     {CKA_ENCRYPT, &true, sizeof(true)},
5849     {CKA_VERIFY, &true, sizeof(true)},
5850     {CKA_WRAP, &true, sizeof(true)},
5851     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
5852     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
5853 };
5854 CK_ATTRIBUTE privateKeyTemplate[] = {
5855     {CKA_TOKEN, &true, sizeof(true)},
5856     {CKA_PRIVATE, &true, sizeof(true)},

```

```

5857     {CKA_SUBJECT, subject, sizeof(subject)},
5858     {CKA_ID, id, sizeof(id)},
5859     {CKA_SENSITIVE, &true, sizeof(true)},
5860     {CKA_DECRYPT, &true, sizeof(true)},
5861     {CKA_SIGN, &true, sizeof(true)},
5862     {CKA_UNWRAP, &true, sizeof(true)}
5863 };
5864 CK_RV rv;
5865
5866 rv = C_GenerateKeyPair(
5867     hSession, &mechanism,
5868     publicKeyTemplate, 5,
5869     privateKeyTemplate, 8,
5870     &hPublicKey, &hPrivateKey);
5871 if (rv == CKR_OK) {
5872     .
5873     .
5874 }

```

5.18.3 C_WrapKey

```

5876 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
5877     CK_SESSION_HANDLE hSession,
5878     CK_MECHANISM_PTR pMechanism,
5879     CK_OBJECT_HANDLE hWrappingKey,
5880     CK_OBJECT_HANDLE hKey,
5881     CK_BYTE_PTR pWrappedKey,
5882     CK_ULONG_PTR pulWrappedKeyLen
5883 );

```

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 5.2 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.

5902 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
5903 CKA_WRAP_TEMPLATE can be used on the wrapping key to specify an attribute set that will be
5904 compared against the attributes of the key to be wrapped. If all attributes match according to the
5905 C_FindObject rules of attribute matching then the wrap will proceed. The value of this attribute is an
5906 attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If
5907 this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
5908 checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
5909 CKR_KEY_HANDLE_INVALID.

5910 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5911 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5912 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5913 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
5914 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
5915 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5916 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5917 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5918 CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE,
5919 CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

5920 Example:

```
5921 CK_SESSION_HANDLE hSession;  
5922 CK_OBJECT_HANDLE hWrappingKey, hKey;  
5923 CK_MECHANISM mechanism = {  
5924     CKM_DES3_ECB, NULL_PTR, 0  
5925 };  
5926 CK_BYTE wrappedKey[8];  
5927 CK_ULONG ulWrappedKeyLen;  
5928 CK_RV rv;  
5929  
5930 .  
5931 .  
5932 ulWrappedKeyLen = sizeof(wrappedKey);  
5933 rv = C_WrapKey(  
5934     hSession, &mechanism,  
5935     hWrappingKey, hKey,  
5936     wrappedKey, &ulWrappedKeyLen);  
5937 if (rv == CKR_OK) {  
5938     .  
5939     .  
5940 }
```

5941 5.18.4 C_UnwrapKey

```
5942 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (  
5943     CK_SESSION_HANDLE hSession,  
5944     CK_MECHANISM_PTR pMechanism,  
5945     CK_OBJECT_HANDLE hUnwrappingKey,  
5946     CK_BYTE_PTR pWrappedKey,  
5947     CK_ULONG ulWrappedKeyLen,  
5948     CK_ATTRIBUTE_PTR pTemplate,
```

```

5949     CK_ULONG ulAttributeCount,
5950     CK_OBJECT_HANDLE_PTR phKey
5951 );

```

5952 **C_UnwrapKey** unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or secret key object.
5953 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is
5954 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the
5955 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the
5956 number of attributes in the template; *phKey* points to the location that receives the handle of the
5957 recovered key.

5958 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
5959 unwrapping, MUST be CK_TRUE.

5960 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
5961 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
5962 default set to CK_TRUE.

5963 Some mechanisms may modify, or attempt to modify, the contents of the *pMechanism* structure at the
5964 same time that the key is unwrapped.

5965 If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without
5966 creating any key object.

5967 The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to
5968 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
5969 assigned (See Section 4.4.1).

5970 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
5971 CKA_UNWRAP_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be
5972 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
5973 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
5974 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
5975 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
5976 conflict occurs on an attempt to unwrap a key then the function SHALL return
5977 CKR_TEMPLATE_INCONSISTENT.

5978 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5979 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5980 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
5981 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5982 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
5983 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
5984 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5985 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5986 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5987 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5988 CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE,
5989 CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN,
5990 CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.

5991 Example:

```

5992 CK_SESSION_HANDLE hSession;
5993 CK_OBJECT_HANDLE hUnwrappingKey, hKey;
5994 CK_MECHANISM mechanism = {
5995     CKM_DES3_ECB, NULL_PTR, 0
5996 };
5997 CK_BYTE wrappedKey[8] = {...};
5998 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
5999 CK_KEY_TYPE keyType = CKK_DES;

```

```

6000 CK_BBOOL true = CK_TRUE;
6001 CK_ATTRIBUTE template[] = {
6002     {CKA_CLASS, &keyClass, sizeof(keyClass)},
6003     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6004     {CKA_ENCRYPT, &true, sizeof(true)},
6005     {CKA_DECRYPT, &true, sizeof(true)}
6006 };
6007 CK_RV rv;
6008
6009 .
6010 .
6011 rv = C_UnwrapKey(
6012     hSession, &mechanism, hUnwrappingKey,
6013     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);
6014 if (rv == CKR_OK) {
6015     .
6016     .
6017 }

```

6018 5.18.5 C_DeriveKey

```

6019 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey)(
6020     CK_SESSION_HANDLE hSession,
6021     CK_MECHANISM_PTR pMechanism,
6022     CK_OBJECT_HANDLE hBaseKey,
6023     CK_ATTRIBUTE_PTR pTemplate,
6024     CK_ULONG ulAttributeCount,
6025     CK_OBJECT_HANDLE_PTR phKey
6026 );

```

6027 **C_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's
6028 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the
6029 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number
6030 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

6031 The values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and
6032 **CKA_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold
6033 for the newly-derived key. See the description of each particular key-derivation mechanism in Section
6034 5.21.2 for any constraints of this type.

6035 If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without
6036 creating any key object.

6037 The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to
6038 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
6039 assigned (See Section 4.4.1).

6040 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6041 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6042 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
6043 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
6044 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6045 CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE,
6046 CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,

6047 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
6048 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
6049 CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
6050 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

6051 Example:

```
6052 CK_SESSION_HANDLE hSession;  
6053 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;  
6054 CK_MECHANISM keyPairMechanism = {  
6055     CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0  
6056 };  
6057 CK_BYTE prime[] = {...};  
6058 CK_BYTE base[] = {...};  
6059 CK_BYTE publicValue[128];  
6060 CK_BYTE otherPublicValue[128];  
6061 CK_MECHANISM mechanism = {  
6062     CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)  
6063 };  
6064 CK_ATTRIBUTE template[] = {  
6065     {CKA_VALUE, &publicValue, sizeof(publicValue)}  
6066 };  
6067 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
6068 CK_KEY_TYPE keyType = CKK_DES;  
6069 CK_BBOOL true = CK_TRUE;  
6070 CK_ATTRIBUTE publicKeyTemplate[] = {  
6071     {CKA_PRIME, prime, sizeof(prime)},  
6072     {CKA_BASE, base, sizeof(base)}  
6073 };  
6074 CK_ATTRIBUTE privateKeyTemplate[] = {  
6075     {CKA_DERIVE, &true, sizeof(true)}  
6076 };  
6077 CK_ATTRIBUTE derivedKeyTemplate[] = {  
6078     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
6079     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
6080     {CKA_ENCRYPT, &true, sizeof(true)},  
6081     {CKA_DECRYPT, &true, sizeof(true)}  
6082 };  
6083 CK_RV rv;  
6084  
6085 .  
6086 .  
6087 rv = C_GenerateKeyPair(  
6088     hSession, &keyPairMechanism,  
6089     publicKeyTemplate, 2,
```

```

6090     privateKeyTemplate, 1,
6091     &hPublicKey, &hPrivateKey);
6092 if (rv == CKR_OK) {
6093     rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
6094     if (rv == CKR_OK) {
6095         /* Put other guy's public value in otherPublicValue */
6096         .
6097         .
6098         rv = C_DeriveKey(
6099             hSession, &mechanism,
6100             hPrivateKey, derivedKeyTemplate, 4, &hKey);
6101         if (rv == CKR_OK) {
6102             .
6103             .
6104         }
6105     }
6106 }

```

5.19 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

5.19.1 C_SeedRandom

```

6110 CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
6111     CK_SESSION_HANDLE hSession,
6112     CK_BYTE_PTR pSeed,
6113     CK_ULONG ulSeedLen
6114 );

```

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**.

5.19.2 C_GenerateRandom

```

6126 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (
6127     CK_SESSION_HANDLE hSession,
6128     CK_BYTE_PTR pRandomData,
6129     CK_ULONG ulRandomLen
6130 );

```


6131 **C_GenerateRandom** generates random or pseudo-random data. *hSession* is the session's handle;
6132 *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in
6133 bytes of the random or pseudo-random data to be generated.

6134 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6135 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6136 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6137 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG,
6138 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6139 Example:

```
6140 CK_SESSION_HANDLE hSession;  
6141 CK_BYTE seed[] = {...};  
6142 CK_BYTE randomData[] = {...};  
6143 CK_RV rv;  
6144  
6145 .  
6146 .  
6147 rv = C_SeedRandom(hSession, seed, sizeof(seed));  
6148 if (rv != CKR_OK) {  
6149     .  
6150     .  
6151 }  
6152 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));  
6153 if (rv == CKR_OK) {  
6154     .  
6155     .  
6156 }
```

6157 5.20 Parallel function management functions

6158 Cryptoki provides the following functions for managing parallel execution of cryptographic functions.
6159 These functions exist only for backwards compatibility.

6160 5.20.1 C_GetFunctionStatus

```
6161 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus)(  
6162     CK_SESSION_HANDLE hSession  
6163 );
```

6164 In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in
6165 parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should
6166 simply return the value CKR_FUNCTION_NOT_PARALLEL.

6167 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6168 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6169 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6170 5.20.2 C_CancelFunction

```
6171 CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction)(  
6172     CK_SESSION_HANDLE hSession  
6173 );
```

6174 In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an
6175 application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value
6176 CKR_FUNCTION_NOT_PARALLEL.

6177 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
6178 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6179 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_CLOSED.

6180 5.21 Callback functions

6181 Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain
6182 events.

6183 5.21.1 Surrender callbacks

6184 Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions;
6185 decryption functions; message digesting functions; signing and MACing functions; functions for verifying
6186 signatures and MACs; dual-purpose cryptographic functions; key management functions; random number
6187 generation functions) executing in Cryptoki sessions can periodically surrender control to the application
6188 who called them if the session they are executing in had a notification callback function associated with it
6189 when it was opened. They do this by calling the session's callback with the arguments (hSession,
6190 CKN_SURRENDER, pApplication), where hSession is the session's handle and pApplication was
6191 supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the
6192 value CKR_OK (to indicate that Cryptoki should continue executing the function) or the value
6193 CKR_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before
6194 returning one of these values, the callback function can perform some computation, if desired.

6195 A typical use of a surrender callback might be to give an application user feedback during a lengthy key
6196 pair generation operation. Each time the application receives a callback, it could display an additional "."
6197 to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the
6198 key pair generation operation (probably by returning the value CKR_CANCEL) if the user hit <ESCAPE>.

6199 A Cryptoki library is not *required* to make *any* surrender callbacks.

6200 5.21.2 Vendor-defined callbacks

6201 Library vendors can also define additional types of callbacks. Because of this extension capability,
6202 application-supplied notification callback routines should examine each callback they receive, and if they
6203 are unfamiliar with the type of that callback, they should immediately give control back to the library by
6204 returning with the value CKR_OK.

6 PKCS #11 Implementation Conformance

6205

6206 An implementation is a conforming implementation if it meets the conditions specified in one or more
6207 server profiles specified in **[PKCS #11-Prof]**.

6208 If a PKCS #11 implementation claims support for a particular profile, then the implementation SHALL
6209 conform to all normative statements within the clauses specified for that profile and for any subclauses to
6210 each of those clauses .

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

List needs to be pasted in here

Gil Abel, Athena Smartcard Solutions, Inc.

Warren Armstrong, QuintessenceLabs

Jeff Bartell, Semper Foris Solutions LLC

Peter Bartok, Venafi, Inc.

Anthony Berglas, Cryptsoft

Joseph Brand, Semper Fortis Solutions LLC

Kelley Burgin, National Security Agency

Robert Burns, Thales e-Security

Wan-Teh Chang, Google Inc.

Hai-May Chao, Oracle

Janice Cheng, Vormetric, Inc.

Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)

Doron Cohen, SafeNet, Inc.

Fadi Cotran, Futurex

Tony Cox, Cryptsoft

Christopher Duane, EMC

Chris Dunn, SafeNet, Inc.

Valerie Fenwick, Oracle

Terry Fletcher, SafeNet, Inc.

Susan Gleeson, Oracle

Sven Gossel, Charismathics

John Green, QuintessenceLabs

Robert Griffin, EMC

Paul Grojean, Individual

Peter Gutmann, Individual

Dennis E. Hamilton, Individual

Thomas Hardjono, M.I.T.

Tim Hudson, Cryptsoft

Gershon Janssen, Individual

Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)

Wang Jingman, Feitan Technologies

Andrey Jivsov, Symantec Corp.

Mark Joseph, P6R

Stefan Kaesar, Infineon Technologies

Greg Kazmierczak, Wave Systems Corp.

6252 Mark Knight, Thales e-Security
6253 Darren Krahn, Google Inc.
6254 Alex Krasnov, Infineon Technologies AG
6255 Dina Kurktchi-Nimeh, Oracle
6256 Mark Lambiase, SecureAuth Corporation
6257 Lawrence Lee, GoTrust Technology Inc.
6258 John Leiseboer, QuintessenceLabs
6259 Sean Leon, Infineon Technologies
6260 Geoffrey Li, Infineon Technologies
6261 Howie Liu, Infineon Technologies
6262 Hal Lockhart, Oracle
6263 Robert Lockhart, Thales e-Security
6264 Dale Moberg, Axway Software
6265 Darren Moffat, Oracle
6266 Valery Osheter, SafeNet, Inc.
6267 Sean Parkinson, EMC
6268 Rob Philpott, EMC
6269 Mark Powers, Oracle
6270 Ajai Puri, SafeNet, Inc.
6271 Robert Relyea, Red Hat
6272 Saikat Saha, Oracle
6273 Subhash Sankuratipati, NetApp
6274 Anthony Scarpino, Oracle
6275 Johann Schoetz, Infineon Technologies AG
6276 Rayees Shamsuddin, Wave Systems Corp.
6277 Radhika Siravara, Oracle
6278 Brian Smith, Mozilla Corporation
6279 David Smith, Venafi, Inc.
6280 Ryan Smith, Futurex
6281 Jerry Smith, US Department of Defense (DoD)
6282 Oscar So, Oracle
6283 Graham Steel, Cryptosense
6284 Michael Stevens, QuintessenceLabs
6285 Michael StJohns, Individual
6286 Jim Susoy, P6R
6287 Sander Temme, Thales e-Security
6288 Kiran Thota, VMware, Inc.
6289 Walter-John Turnes, Gemini Security Solutions, Inc.
6290 Stef Walter, Red Hat
6291 James Wang, Vormetric
6292 Jeff Webb, Dell
6293 Peng Yu, Feitian Technologies

- 6294 Magda Zdunkiewicz, Cryptsoft
- 6295 Chris Zimman, Individual

Appendix B. Manifest constants

6296

6297

6298

The definitions for manifest constants specified in this document can be found in the following normative computer language definition files:

6299

- [include/pkcs11-v3.00/pkcs11.h](#)

6300

- [include/pkcs11-v3.00/pkcs11t.h](#)

6301

- [include/pkcs11-v3.00/pkcs11f.h](#)

Appendix C. Revision History

Revision	Date	Editor	Changes Made
csprd 02 wd01	Oct 8 2019	Dieter Bong	Created csprd02 based on csprd01
csprd 02 wd02	Nov 8 2019	Dieter Bong	Item #26 as per “PKCS11 mechanisms review” document
csprd 02 wd03	Dec 3 2019	Dieter Bong	Changes as per “PKCS11 base spec review” document