# OASIS

# **PKCS #11 Cryptographic Token Interface Base Specification Version 3.0**

## **OASIS Standard**

# 15 June 2020

This stage:

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

#### **Previous 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

#### 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/os/include/pkcs11-v3.0/
- ALERT: Due to a clerical error when publishing the Committee Specification, the header files listed above are outdated and may contain serious flaws. The TC is addressing this in the next round of edits. Meanwhile, users of the standard can find the correct header files at https://github.com/oasistcs/pkcs11/tree/master/working/3-00-current.

#### **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-basev2.40.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/pkcs11hist/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 membership of OASIS 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. 15 June 2020. OASIS Standard. https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/os/pkcs11-base-v3.0-os.html. Latest stage: https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html.

# **Notices**

Copyright © OASIS Open 2020. 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	
5.5.9 C_SetPIN	
5.6 Session management functions	
5.6.1 C_OpenSession	
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	
5.6.10 C_Logout	
5.7 Object management functions	
5.7.1 C_CreateObject	
5.7.2 C_CopyObject	
5.7.3 C_DestroyObject	
5.7.4 C_GetObjectSize	
5.7.5 C_GetAttributeValue	
5.7.6 C_SetAttributeValue	
5.7.7 C_FindObjectsInit	
5.7.8 C_FindObjects	
5.7.9 C_FindObjectsFinal	
5.8 Encryption functions	
5.8.1 C_EncryptInit	
5.8.2 C_Encrypt	
5.8.3 C_EncryptUpdate	
5.8.4 C_EncryptFinal	
5.9 Message-based encryption functions	
5.9.1 C_MessageEncryptInit	
5.9.2 C_EncryptMessage	
5.9.3 C_EncryptMessageBegin	
5.9.4 C_EncryptMessageNext	115
5.9.5 C_ MessageEncryptFinal	
5.10 Decryption functions	
5.10.1 C_DecryptInit	
5.10.2 C_Decrypt	
5.10.3 C_DecryptUpdate	
5.10.4 C_DecryptFinal	
5.11 Message-based decryption functions	
5.11.1 C_MessageDecryptInit	
5.11.2 C_DecryptMessage	
5.11.3 C_DecryptMessageBegin	
5.11.4 C_DecryptMessageNext	
5.11.5 C_MessageDecryptFinal	
5.12 Message digesting functions	

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_0	GetFunctionStatus	161
5.20.2 C_0	CancelFunction	161
5.21 Callbacl	k functions	162
5.21.1 Sur	render callbacks	162
5.21.2 Ver	ndor-defined callbacks	162
6 PKCS #11	. Implementation Conformance	163
Appendix A.	Acknowledgments	164
Appendix B.	Manifest constants	166
Appendix C.	Revision History	167

## 1 **1 Introduction**

- 2 This document describes the basic PKCS#11 token interface and token behavior.
- 3 The PKCS#11 standard specifies an application programming interface (API), called "Cryptoki," for
- 4 devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a
- 5 simple object based approach, addressing the goals of technology independence (any kind of device) and
- 6 resource sharing (multiple applications accessing multiple devices), presenting to applications a common,
- 7 logical view of the device called a "cryptographic token".
- 8 This document specifies the data types and functions available to an application requiring cryptographic
- 9 services using the ANSI C programming language. The supplier of a Cryptoki library implementation
   10 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
- 12 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.
- 15 Cryptoki isolates an application from the details of the cryptographic device. The application does not
- 16 have to change to interface to a different type of device or to run in a different environment; thus, the
- 17 application is portable. How Cryptoki provides this isolation is beyond the scope of this document,
- 18 although some conventions for the support of multiple types of device will be addressed here and
- 19 possibly in a separate document.
- Details of cryptographic mechanisms (algorithms) may be found in the associated PKCS#11 Mechanisms
   documents.

#### 22 **1.1 IPR Policy**

- 23 This specification is provided under the RF on RAND Terms Mode of the OASIS IPR Policy, the mode
- 24 chosen when the Technical Committee was established. For information on whether any patents have
- 25 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-
- 27 open.org/committees/pkcs11/ipr.php).

### 28 **1.2 Terminology**

- 29 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]**.

### 32 1.3 Definitions

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

34	API	Application programming interface.	
35	Application	Any computer program that calls the Cryptoki interface.	
36	ASN.1	Abstract Syntax Notation One, as defined in X.680.	
37	Attribute	A characteristic of an object.	
38	BER	Basic Encoding Rules, as defined in X.690.	
39	CBC	Cipher-Block Chaining mode, as defined in FIPS PUB 81.	
40 41	Certificate	A signed message binding a subject name and a public key, or a subject name and a set of attributes.	
42	CMS	Cryptographic Message Syntax (see RFC 5652)	

43 44 45 46	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.
47	Cryptoki	The Cryptographic Token Interface defined in this standard.
48	Cryptoki library	A library that implements the functions specified in this standard.
49	DER	Distinguished Encoding Rules, as defined in X.690.
50	DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
51	DSA	Digital Signature Algorithm, as defined in FIPS PUB 186-4.
52	EC	Elliptic Curve
53	ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
54	IV	Initialization Vector.
55	MAC	Message Authentication Code.
56	Mechanism	A process for implementing a cryptographic operation.
57 58	Object	An item that is stored on a token. May be data, a certificate, or a key.
59	PIN	Personal Identification Number.
60	PKCS	Public-Key Cryptography Standards.
61	PRF	Pseudo random function.
62	PTD	Personal Trusted Device, as defined in MeT-PTD
63	RSA	The RSA public-key cryptosystem.
64	Reader	The means by which information is exchanged with a device.
65	Session	A logical connection between an application and a token.
66	Slot	A logical reader that potentially contains a token.
67	SSL	The Secure Sockets Layer 3.0 protocol.
68 69	Subject Name	The X.500 distinguished name of the entity to which a key is assigned.
70	SO	A Security Officer user.
71	TLS	Transport Layer Security.
72	Token	The logical view of a cryptographic device defined by Cryptoki.
73	User	The person using an application that interfaces to Cryptoki.
74 75 76	UTF-8	Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.
77	WIM	Wireless Identification Module.
78	WTLS	Wireless Transport Layer Security.

## 79 **1.4 Symbols and abbreviations**

80 The following symbols are used in this standard:

81 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		
р	a pointer		
pb	a pointer to a CK_BYTE		
ph	a pointer to a handle		
pul	a pointer to a CK_ULONG		

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

```
86
87
88
89
90
91
```

```
/* an unsigned 8-bit value */
typedef unsigned char CK_BYTE;
/* an unsigned 8-bit character */
typedef CK_BYTE CK_CHAR;
/* an 8-bit UTF-8 character */
typedef CK_BYTE CK_UTF8CHAR;
/* a BYTE-sized Boolean flag */
typedef CK_BYTE CK_BBOOL;
/* an unsigned value, at least 32 bits long */
```

100	typedef unsigned long int CK ULONG;
101	
102	/* a signed value, the same size as a CK ULONG */
103	typedef long int CK LONG;
104	··· _
105	<pre>/* at least 32 bits; each bit is a Boolean flag */</pre>
106	typedef CK ULONG CK FLAGS;
107	

108 Cryptoki also uses pointers to some of these data types, as well as to the type void, which are 109 implementation-dependent. These pointer types are:

116 Cryptoki also defines a pointer to a CK\_VOID\_PTR, which is implementation-dependent:

```
117
118
```

CK\_VOID\_PTR\_PTR /\* Pointer to a CK\_VOID\_PTR \*/

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

```
120
121
```

/\* A NULL pointer \*/

122 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

127 which case they are hexadecimal values.

- 128 The **CK\_CHAR** data type holds characters from the following table, taken from ANSI C:
- 129 Table 3, Character Set

NULL PTR

Category	Characters
Letters	ABCDEFGHIJKLMNOPQRSTUVWXYZabcd efghijklmnopqrstuvwxyz
Numbers	0 1 2 3 4 5 6 7 8 9
Graphic characters	! " # % & ' ( ) * + , / : ; < = > ? [ \ ] ^ {   }~
Blank character	" "

130 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

- 136 CK BBOOL:
- 137
   #define CK\_FALSE 0

   138
   #define CK\_TRUE 1

   139
   139
- 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):
- 142 #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		

## 152 **1.5 Normative References**

153 154	[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
155 156	[FIPS PUB 81]	NIST. <i>FIPS 81: DES Modes of Operation</i> . December 1980. URL: http://csrc.nist.gov/publications/fips/fips81/fips81.htm
157 158	[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
159 160 161 162 163	[PKCS11-Curr]	<i>PKCS #11 Cryptographic Token Interface Current Mechanisms Specification</i> <i>Version 2.40.</i> 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.
164 165 166 167 168	[PKCS11-Hist]	<i>PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification</i> <i>Version 2.40.</i> 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.
169 170 171 172 173	[PKCS11-Prof]	<i>PKCS #11 Cryptographic Token Interface Profiles Version 2.40.</i> 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.
174 175	[PKCS #1]	RSA Laboratories. <i>RSA Cryptography Standard.</i> v2.1, June 14, 2002. URL: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf
176 177 178	[PKCS #3]	RSA Laboratories. <i>Diffie-Hellman Key-Agreement Standard.</i> v1.4, November 1993. URL: ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-3.doc
179 180	[PKCS #5]	RSA Laboratories. <i>Password-Based Encryption Standard</i> . v2.0, March 25, 1999 URL: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf
181 182 183	[PKCS #7]	RSA Laboratories. <i>Cryptographic Message Syntax Standard.</i> v1.5, November 1993 URL : ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-7.doc
184 185 186	[PKCS #8]	RSA Laboratories. <i>Private-Key Information Syntax Standard</i> . v1.2, November 1993. URL: ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-8.doc
187 188 189 190 191	[PKCS11-UG]	<i>PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40.</i> Edited by John Leiseboer and Robert Griffin. 16 November 2014. OASIS Committee Note 02. http://docs.oasis-open.org/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.
192 193	[PKCS #12]	RSA Laboratories. <i>Personal Information Exchange Syntax Standard</i> . v1.0, June 1999.

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

## 228 **1.6 Non-Normative References**

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

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

# 282 2 Platform- and compiler-dependent directives for C 283 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.
- 290 Plattform specific implementation hints can be found in the pkcs11.h header file.

#### 291 **2.1 Structure packing**

292 Cryptoki structures are packed to occupy as little space as is possible. Cryptoki structures SHALL be 293 packed with 1-byte alignment.

#### 294 **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:

#### 297 **• CK\_PTR**

- 298 CK\_PTR is the "indirection string" a given platform and compiler uses to make a pointer to an object. It is 299 used in the following fashion:
- 300 typedef CK\_BYTE CK\_PTR CK\_BYTE\_PTR;

#### 301 • CK\_DECLARE\_FUNCTION

302 CK\_DECLARE\_FUNCTION (returnType, name), when followed by a parentheses-enclosed
 303 list of arguments and a semicolon, declares a Cryptoki API function in a Cryptoki library. returnType is
 304 the return type of the function, and name is its name. It SHALL be used in the following fashion:

```
305CK_DECLARE_FUNCTION(CK_RV, C_Initialize)(306CK_VOID_PTR pReserved307);
```

#### 308 • CK\_DECLARE\_FUNCTION\_POINTER

309 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):

```
315
316
317
```

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

318

320

319 or:

typedef CK DECLARE FUNCTION POINTER(CK RV, myC InitializeType)(

321	CK VOID PTR pReserved
322	);
323	<pre>myC_InitializeType myC_Initialize;</pre>

#### 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):

332 333 CK CALLBACK FUNCTION(CK RV, myCallback)(args);

334

or:

335	<pre>typedef CK_CALLBACK_FUNCTION(CK_RV, myCallbackType)(args);</pre>
336	myCallbackType myCallback;

#### 337 • NULL\_PTR

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

## **340 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

343 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 toplevel 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.

349 When including either of these header files, a source file MUST specify the preprocessor directives 350 indicated in Section 2.

## 351 **3.1 General information**

352 Cryptoki represents general information with the following types:

### 353 • CK\_VERSION; CK\_VERSION\_PTR

354 CK\_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an
 355 SSL or TLS implementation, or the hardware or firmware version of a slot or token. It is defined as
 356 follows:

357 358 359

360

361

typedef struct CK\_VERSION {
 CK\_BYTE major;
 CK\_BYTE minor;
} CK\_VERSION;

362 The fields of the structure have the following meanings:

363 *major* major version number (the integer portion of the version)

- 364 *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.
- 367 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

- 368 Minor revisions of the Cryptoki standard are always upwardly compatible within the same major version 369 number.
- 370 **CK\_VERSION\_PTR** is a pointer to a **CK\_VERSION**.

#### 371 • CK\_INFO; CK\_INFO\_PTR

372 **CK\_INFO** provides general information about Cryptoki. It is defined as follows:

373 374 375 376 377 378 379 380	<pre>typedef struct CK_INFO {     CK_VERSION cryptokiVers:     CK_UTF8CHAR manufacture:     CK_FLAGS flags;     CK_UTF8CHAR libraryDesc:     CK_VERSION libraryVersio } CK_INF0;</pre>	ion; rID[32]; ription[32]; pn;
381	The fields of the structure have the f	ollowing meanings:
382 383	cryptokiVersion	Cryptoki interface version number, for compatibility with future revisions of this interface
384 385	manufacturerID	ID of the Cryptoki library manufacturer. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
386	flags	bit flags reserved for future versions. MUST be zero for this version
387 388	libraryDescription	character-string description of the library. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
389	libraryVersion	Cryptoki library version number

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

392 **CK\_INFO\_PTR** is a pointer to a **CK\_INFO**.

#### 

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

396 397	typede	f CK_ULONG CK_NOTIE	TICATION;
398	For this vers	sion of Cryptoki, the foll	owing types of notifications are defined:
399 400	CKN_SU	RRENDER	
401	The notificat	tions have the following	meanings:
402 403 404		CKN_SURRENDER	Cryptoki is surrendering the execution of a function executing in a session so that the application may perform other operations. After performing any desired operations, the application should indicate

to Cryptoki whether to continue or cancel the function (see Section

#### 407 **3.2 Slot and token types**

405

406

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:

5.21.1).

411	typedef	CK	ULONG	CK	SLOT	ID;	
412		_	-	_		_	

- 413 A list of CK\_SLOT\_IDs is returned by C\_GetSlotList. A priori, any value of CK\_SLOT\_ID can be a valid
- 414 slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a 415 slot, however.
- 416 **CK\_SLOT\_ID\_PTR** is a pointer to a **CK\_SLOT\_ID**.

#### 417 • CK\_SLOT\_INFO; CK\_SLOT\_INFO\_PTR

418 **CK\_SLOT\_INFO** provides information about a slot. It is defined as follows:

427	The fields of the structure have the	following meanings:
428 429	slotDescription	character-string description of the slot. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
430 431	manufacturerID	ID of the slot manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
432 433	flags	bits flags that provide capabilities of the slot. The flags are defined below
434	hardwareVersion	version number of the slot's hardware
435	firmwareVersion	version number of the slot's firmware

- 436 The following table defines the *flags* field:
- 437 Table 5, Slot Information Flags

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x00000001	True if a token is present in the slot ( <i>e.g.</i> , 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.

441 **CK\_SLOT\_INFO\_PTR** is a pointer to a **CK\_SLOT\_INFO**.

#### 442 • CK\_TOKEN\_INFO; CK\_TOKEN\_INFO\_PTR

- 443 CK\_TOKEN\_INFO provides information about a token. It is defined as follows:
- 444 typedef struct CK\_TOKEN\_INFO { 445 CK UTF8CHAR label[32];

446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464	CK_UTF8CHAR manufacturer CK_UTF8CHAR model[16]; CK_CHAR serialNumber[16] CK_FLAGS flags; CK_ULONG ulMaxSessionCoun CK_ULONG ulMaxRwSessionCoun CK_ULONG ulMaxRwSessionCoun CK_ULONG ulMaxPinLen; CK_ULONG ulMaxPinLen; CK_ULONG ulMaxPinLen; CK_ULONG ulTotalPublicMen CK_ULONG ulTotalPublicMen CK_ULONG ulTotalPrivateMen CK_ULONG ulTotalPrivateMen CK_ULONG ulFreePrivateMen CK_VERSION hardwareVersi CK_VERSION firmwareVersi CK_CHAR utcTime[16]; } CK_TOKEN_INFO;	<pre>rID[32]; ; int; Count; nt; emory; Memory; Memory; con; con;</pre>
465	The fields of the structure have the fe	ollowing meanings:
466 467 468	label	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
469 470	manufacturerID	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
471 472	model	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
473 474	serialNumber	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
475 476	flags	bit flags indicating capabilities and status of the device as defined below
477 478 479	ulMaxSessionCount	maximum number of sessions that can be opened with the token at one time by a single application (see <b>CK_TOKEN_INFO Note</b> below)
480 481	ulSessionCount	number of sessions that this application currently has open with the token (see <b>CK_TOKEN_INFO Note</b> below)
482 483 484	ulMaxRwSessionCount	maximum number of read/write sessions that can be opened with the token at one time by a single application (see <b>CK_TOKEN_INFO Note</b> below)
485 486	ulRwSessionCount	number of read/write sessions that this application currently has open with the token (see <b>CK_TOKEN_INFO Note</b> below)
487	ulMaxPinLen	maximum length in bytes of the PIN
488	ulMinPinLen	minimum length in bytes of the PIN
489 490	ulTotalPublicMemory	the total amount of memory on the token in bytes in which public objects may be stored (see <b>CK_TOKEN_INFO Note</b> below)

491 492	ulFreePublicMemory	the amount of free (unused) memory on the token in bytes for public objects (see <b>CK_TOKEN_INFO Note</b> below)
493 494	ulTotalPrivateMemory	the total amount of memory on the token in bytes in which private objects may be stored (see <b>CK_TOKEN_INFO Note</b> below)
495 496	ulFreePrivateMemory	the amount of free (unused) memory on the token in bytes for private objects (see <b>CK_TOKEN_INFO Note</b> below)
497	hardwareVersion	version number of hardware
498	firmwareVersion	version number of firmware
499 500 501 502 503 504	utcTime	current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters). The value of this field only makes sense for tokens equipped with a clock, as indicated in the token information flags (see below)

- 505 The following table defines the *flags* field:
- 506 Table 6, Token Information Flags

Bit Flag	Mask	Meaning
CKF_RNG	0x00000001	True if the token has its own random number generator
CKF_WRITE_PROTECTED	0x00000002	True if the token is write- protected (see below)
CKF_LOGIN_REQUIRED	0x00000004	True if there are some cryptographic functions that a user MUST be logged in to perform
CKF_USER_PIN_INITIALIZED	0x0000008	True if the normal user's PIN has been initialized
CKF_RESTORE_KEY_NOT_NEEDED	0x00000020	True if a successful save of a session's cryptographic operations state <i>always</i> contains all keys needed to restore the state of the session
CKF_CLOCK_ON_TOKEN	0x00000040	True if token has its own hardware clock
CKF_PROTECTED_AUTHENTICATION_PA TH	0x00000100	True if token has a "protected authentication path", whereby a user can log into the token without passing a PIN through the Cryptoki library
CKF_DUAL_CRYPTO_OPERATIONS	0x00000200	True if a single session with the token can perform dual cryptographic operations (see Section 5.14)

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	0x0000800	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.

507 Exactly what the **CKF\_WRITE\_PROTECTED** flag means is not specified in Cryptoki. An application may 508 be unable to perform certain actions on a write-protected token; these actions can include any of the 509 following, among others:

- Creating/modifying/deleting any object on the token.
- Creating/modifying/deleting a token object on the token.

- 512 Changing the SO's PIN.
- Changing the normal user's PIN.

514 The token may change the value of the **CKF\_WRITE\_PROTECTED** flag depending on the session state 515 to implement its object management policy. For instance, the token may set the

516 **CKF\_WRITE\_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy

517 that does not allow any objects, public or private, to be created, modified, or deleted unless the user has 518 successfully called C Login.

519 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.

522 The CKF\_USER\_PIN\_TO\_BE\_CHANGED and CKF\_SO\_PIN\_TO\_BE\_CHANGED flags may always be

523 set to false if the token does not support the functionality. If a PIN is set to the default value, or has 524 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,

- 525 but only the C SetPIN function can be called. Calling any other function that required the user to be
- 527 logged in will cause CKR PIN EXPIRED to be returned until C SetPIN is called successfully.
- 528 **CK\_TOKEN\_INFO Note**: The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount,
- 529 ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and
- 530 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
- 532 ulMaxSessionCount and ulMaxRwSessionCount can have the special value
- 533 CK\_EFFECTIVELY\_INFINITE, which means that there is no practical limit on the number of sessions
- 534 (resp. R/W sessions) an application can have open with the token.
- 535 It is important to check these fields for these special values. This is particularly true for
- 536 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.
- 539 The upshot of all this is that the correct way to interpret (for example) the ulMaxSessionCount field is 540 something along the lines of the following:

```
541
          CK TOKEN INFO info;
542
543
544
          if ((CK LONG) info.ulMaxSessionCount
545
              == CK UNAVAILABLE INFORMATION) {
            /* Token refuses to give value of ulMaxSessionCount */
546
547
548
549
          } else if (info.ulMaxSessionCount == CK EFFECTIVELY INFINITE) {
550
            /* Application can open as many sessions as it wants */
551
            .
552
553
          } else {
554
            /* ulMaxSessionCount really does contain what it should */
555
556
557
          }
558
```

559 CK\_TOKEN\_INFO\_PTR is a pointer to a CK\_TOKEN\_INFO.

#### 560 **3.3 Session types**

561 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 context-specific type described in Section 4.9. It is defined as follows:

574 575	typedef CK_ULONG CK_USER_TYPE;
576	For this version of Cryptoki, the following types of users are defined:
577 578 579	CKU_SO CKU_USER CKU_CONTEXT_SPECIFIC

#### 

581 **CK\_STATE** holds the session state, as described in **[PKCS11-UG]**. It is defined as follows:

582 583	typedef CK_ULONG CK_STATE;	
584	For this version of Cryptoki, the following session states are defined:	
585 586 587 588	CKS_RO_PUBLIC_SESSION CKS_RO_USER_FUNCTIONS CKS_RW_PUBLIC_SESSION 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

- 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.

#### 3.4 Object types 609

610 Cryptoki represents object information with the following types:

#### CK OBJECT HANDLE; CK OBJECT HANDLE PTR 611

#### 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 handle which is fixed for the lifetime of the object; however, if a particular session can use a particular 617 handle to access a particular object, then that session will continue to be able to use that handle to 618 619 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:
- 623 624

CK INVALID HANDLE

625 CK\_OBJECT\_HANDLE\_PTR is a pointer to a CK\_OBJECT\_HANDLE.

#### CK\_OBJECT\_CLASS; CK\_OBJECT\_CLASS\_PTR 626

- CK\_OBJECT\_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes. 627 It is defined as follows: 628
- 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:

642 643 typedef CK\_ULONG CK\_HW\_FEATURE\_TYPE;

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.
- 647 648

CKH\_VENDOR\_DEFINED

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:

664 665	typedef CK_ULONG CK_CERTIFICATE_TYPE;
666 667	Certificate types are defined with the objects and mechanisms that use them. The certificate type is specified on an object through the CKA_CERTIFICATE_TYPE attribute of the object.
668	Vendor defined values for this type may also be specified.
669 670	CKC_VENDOR_DEFINED

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	0x0000000UL	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		-	-	_		-

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

- 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:

```
692typedef struct CK_ATTRIBUTE {693CK_ATTRIBUTE_TYPE type;694CK_VOID_PTR pValue;695CK_ULONG ulValueLen;696} CK_ATTRIBUTE;697697
```

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

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

708

The constant CK\_UNAVAILABLE\_INFORMATION is used in the ulValueLen field to denote an invalid or 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:

The fields of the structure have the following meanings:

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

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

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

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

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 unsigend ulong value represting a specific token profile. It is defined as follows:

733 734	<pre>typedef CK_ULONG CK_PROFILE_ID;</pre>
735 736 737	Profiles are defines in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater than 0xfffffff may cause compatibility issues on platforms that have CK_ULONG values of 32 bits, and should be avoided.
738	Vendor defined values for this type may also be specified.

739 740	CKP_VENDOR_DEFINED
741 742	Profile IDs <b>CKP_VENDOR_DEFINED</b> and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.
743	
744 745	Valid Profile IDs in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:
746	CKP_INVALID_ID
747	CK_PROFILE_ID_PTR is a pointer to a CK_PROFILE_ID.

#### 748 • CK\_JAVA\_MIDP\_SECURITY\_DOMAIN

749 **CK\_JAVA\_MIDP\_SECURITY\_DOMAIN** is a value that identifies the Java MIDP security domain of a certificate. It is defined as follows:

751 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

754

#### 755 **3.5 Data types for mechanisms**

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

#### 757 • CK\_MECHANISM\_TYPE; CK\_MECHANISM\_TYPE\_PTR

758 **CK\_MECHANISM\_TYPE** is a value that identifies a mechanism type. It is defined as follows:

759 760	typedef CK_ULONG CK_MECHANISM_TYPE;
761 762	Mechanism types are defined with the objects and mechanism descriptions that use them. Vendor defined values for this type may also be specified.
763 764	CKM_VENDOR_DEFINED
765	Mechanism types CKM VENDOR DEFINED and above are permanently reserved for token vendors

- 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.
- 767 **CK\_MECHANISM\_TYPE\_PTR** is a pointer to a **CK\_MECHANISM\_TYPE**.

#### 768 • CK\_MECHANISM; CK\_MECHANISM\_PTR

769 CK\_MECHANISM is a structure that specifies a particular mechanism and any parameters it requires. It
 770 is defined as follows:

771	typedef struct CK MECHANISM {
772	CK MECHANISM TYPE mechanism;
773	CK_VOID_PTR pParameter;
774	CK_ULONG ulParameterLen;
775	} CK MECHANISM;
776	—

- The fields of the structure have the following meanings:
- 778

*mechanism* the type of mechanism

pParameter pointer to the parameter if required by the mechanism

780 *ulParameterLen* length in bytes of the parameter

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

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 defined as follows:

788 789 790 791 792 793	<pre>typedef struct CK_MECHANIS     CK_ULONG ulMinKeySize;     CK_ULONG ulMaxKeySize;     CK_FLAGS flags; } CK_MECHANISM_INFO;</pre>	M_INFO {
794	The fields of the structure have the fo	ollowing meanings:
795	ulMinKeySize	the minimum size of the key for the mechanism (whether this is
796		measured in bits or in bytes is mechanism-dependent)
797	ulMaxKeySize	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	0x0000008	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 <b>C_*MessageBegin</b> . One of CKF_MESSAGE_* flag must also be set.
CKF_FIND_OBJECTS	0x00000040	This flag can be passed in as a parameter to <b>C_SessionCancel</b> 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 <b>C_EncryptInit</b>
CKF_DECRYPT	0x00000200	True if the mechanism can be used with <b>C_DecryptInit</b>
CKF_DIGEST	0x00000400	True if the mechanism can be used with <b>C_DigestInit</b>
CKF_SIGN	0x0000800	True if the mechanism can be used with <b>C_SignInit</b>
CKF_SIGN_RECOVER	0x00001000	True if the mechanism can be used with <b>C_SignRecoverInit</b>
CKF_VERIFY	0x00002000	True if the mechanism can be used with <b>C_VerifyInit</b>
CKF_VERIFY_RECOVER	0x00004000	True if the mechanism can be used with <b>C_VerifyRecoverInit</b>
CKF_GENERATE	0x00008000	True if the mechanism can be used with <b>C_GenerateKey</b>
CKF_GENERATE_KEY_PAIR	0x00010000	True if the mechanism can be used with <b>C_GenerateKeyPair</b>
CKF_WRAP	0x00020000	True if the mechanism can be used with <b>C_WrapKey</b>
CKF_UNWRAP	0x00040000	True if the mechanism can be used with <b>C_UnwrapKey</b>
CKF_DERIVE	0x00080000	True if the mechanism can be used with <b>C_DeriveKey</b>
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.

#### **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:

808 809		typedef CK_ULONG CK_RV;
810	Vend	for defined values for this type may also be specified.
811 812		CKR_VENDOR_DEFINED
813	Sect	ion 5.1 defines the meaning of each CK_RV value. Return values CKR_VENDOR_DEFINED and

- above are permanently reserved for token vendors. For interoperability, vendors should register their
- 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
```

The arguments to a notification call	back function have the following meanings:
hSession	The handle of the session performing the callback
event	The type of notification callback
pApplication	An application-defined value. This is the same value as was passed to <b>C_OpenSession</b> to open the session performing the callback
	The arguments to a notification call hSession event pApplication

#### 830 + CK\_C\_XXX

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

# 836 • CK\_FUNCTION\_LIST; 837 CK\_FUNCTION\_LIST\_PTR\_PTR

CK\_FUNCTION\_LIST\_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 <sup>-</sup> C Initialize C Initialize;
843	CK <sup>C</sup> Finalize C Finalize;
844	CK <sup>C</sup> GetInfo C GetInfo;
845	CK <sup>C</sup> GetFunctionList C GetFunctionList;
846	CK <sup>C</sup> GetSlotList C GetSlotList;
847	CK <sup>-</sup> C <sup>-</sup> GetSlotInfo C <sup>-</sup> GetSlotInfo;
848	CK <sup>C</sup> GetTokenInfo <sup>C</sup> GetTokenInfo;
849	CK <sup>C</sup> GetMechanismList C GetMechanismList;
850	CK <sup>C</sup> GetMechanismInfo C <sup>G</sup> etMechanismInfo;
851	CK <sup>C</sup> InitToken C InitToken;
852	CK <sup>C</sup> InitPIN C InitPIN;
853	CK <sup>C</sup> SetPIN C SetPIN;
854	CK <sup>C</sup> OpenSession C OpenSession;
855	CK <sup>C</sup> 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 CetObjectSize C CetObjectSize.
967	
060	
000	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 DecryptUndate C DecryptUndate:
879	CK C DecryptFinal C DecryptFinal:
880	CK C DigestInit C DigestInit.
881	CK_C_Digost_C_Digost.
882	CK_C_DigestUpdate.C_DigestUpdate.
883	CK_C_Digestbalte C_Digestopdate,
003	CK_C_DigestRey C_DigestRey;
004	
000	ck_c_signinic c_signinic;
880	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 GenerateKev C GenerateKev;
902	CK C GenerateKevPair C GenerateKevPair:
903	CK C WrapKey C WrapKey:
904	CK C HowranKey C HowranKey:
905	CK C DeriveKey C DeriveKey.
906	CK_C_SeedBandom_C_SeedBandom.
907	CK_C_ConcrateBandom C_ConcrateBandom;
908	CK_CConstantionStatus_C_ConstantionStatus.
900	CK_C_GeneralFunction C_Getrunction;
010	CK_C_Ualcerfulction C_Calcerfulction;
011	CK_C_waitfolsiotevent C_waitfolsiotevent;
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 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API 916 MUST have an entry point defined in the Cryptoki library's CK\_FUNCTION\_LIST structure. If a particular 917 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the 918 919 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 1222 the set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for

- this version of the specification may be returned via **C\_GetInterfaceList** or **C\_GetInterface.**
- 924

- 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 <sup>-</sup> C Initialize C Initialize;
941	CK <sup>C</sup> Finalize C Finalize;
942	CK <sup>-</sup> C <sup>-</sup> GetInfo C GetInfo;
943	CK_C_GetFunctionList C_GetFunctionList;
944	CK <sup>-</sup> C <sup>-</sup> GetSlotList C GetSlotList;
945	CK <sup>C</sup> GetSlotInfo C <sup>G</sup> etSlotInfo;
946	CK <sup>-</sup> C <sup>-</sup> GetTokenInfo <sup>-</sup> C GetTokenInfo;
947	CK <sup>C</sup> GetMechanismList C GetMechanismList;
948	CK <sup>C</sup> GetMechanismInfo C <sup>G</sup> etMechanismInfo;
949	CK_C_InitToken C_InitToken;
950	CK C InitPIN C InitPIN;
951	CK_C_SetPIN C_SetPIN;
952	CK <sup>-</sup> C <sup>-</sup> OpenSession C OpenSession;
953	CK_C_CloseSession C_CloseSession;
954	CK C CloseAllSessions C CloseAllSessions;
955	CK_C_GetSessionInfo C_GetSessionInfo;
956	CK <sup>-</sup> C <sup>-</sup> GetOperationState C GetOperationState;
957	CK <sup>C</sup> SetOperationState C <sup>S</sup> SetOperationState;
958	CKC Login C Login;
959	CK <sup>C</sup> Logout C Logout;
960	CK C CreateObject C CreateObject;
961	CK <sup>C</sup> C <sup>C</sup> CopyObject C CopyObject;
962	CK <sup>C</sup> C <sup>D</sup> estroyObject 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;
9//	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 SignBecover C SignBecover	
988	CK C Verifulnit C Verifulnit.	
989	CK_C Vorify C Vorify	
900	CK_C_VerifyUndeta_C_VerifyUndeta.	
001	CK_C_verifyoptate C_verifyoptate,	
991	CK_C_VerilyFinal C_VerilyFinal;	
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 MessageDecrymtInit C MessageDecrymtInit.	
1018	CK C DecryptMessage C DecryptMessage	
1019	CK C DecryptMessageBegin C DecryptMessageBegin.	
1020	CK_C_DecryptMessageNeyt_C_DecryptMessageNeyt.	
1020	CK_C_DestageDecryptFinal_C_DestageDecryptFinal.	
1022	CK C MossageSignThit C MossageSignThit.	
1022	CK_C_ignMagagaga	
1023	CK_C_SignMessage C_SignMessage;	
1024		
1025	CK_C_SIGNMESSAGENEXL C_SIGNMESSAGENEXL;	
1020	CK_C_MessageSignFinal_C_MessageSignFinal;	
1027	CK_C_MessageverifyInit;	
1020 1020	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 *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 1040

1041 CK\_INTERFACE is a structure which contains an interface name with a function list and flag.

1042 It is defined as follows:

1043	typedef struct CK	INTERFACE {
1044	CK UTF8CHAR PTR	_ pInterfaceName;
1045	CK VOID PTR	pFunctionList;
1046	CK FLAGS	flags;
1047	} CK INTERFACE;	-

1048

1049 The fields of the structure have the following meanings:

- 1050 pInterfaceName the name of the interface 1051 pFunctionList the interface function list which must always begin with a 1052 CK VERSION structure as the first field
- 1053 bit flags specifying interface capabilities flags

The interface name "PKCS 11" is reserved for use by interfaces defined within the cryptoki specification. 1054

1055 Interfaces starting with the string: "Vendor" are reserved for vendor use and will not oetherwise be defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface 1056 names of "Vendor {vendor name}". For example "Vendor ACME Inc". 1057

- 1058
- 1059 The following table defines the flags field.
- 1060 Table 9, CK\_INTERFACE Flags

Bit Flag	Mask	Meaning
CKF_INTERFACE_FORK_SAFE	0x0000001	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.

1061

- 1062 CK\_INTERFACE\_PTR is a pointer to a CK\_INTERFACE.
- 1063 CK\_INTERFACE\_PTR\_PTR is a pointer to a CK\_INTERFACE\_PTR.

#### 3.7 Locking-related types 1064

1065 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. 1066

#### CK\_CREATEMUTEX 1067

CK\_CREATEMUTEX is the type of a pointer to an application-supplied function which creates a new 1068 1069 mutex object and returns a pointer to it. It is defined as follows:

1070 1071 1072 1073	<pre>typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX)(         CK_VOID_PTR_PTR ppMutex );</pre>	
1074	Calling a CK_CREATEMUTEX function returns the pointer to the new mutex object in the location pointer	ed

1074 Calling a CK\_CREATEMOTEX function returns the pointer to the new mutex object in the location po 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 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	HOSI	_MEN	IORY -	-
1089	CKR	MUTE	EX_BA	AD	

# 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:

- If a CK\_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a lock on that mutex and returns.
- If a CK\_LOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- If a CK\_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the behavior of the function call is undefined.
- If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:
- 1102oIf exactly one thread was blocking on that particular mutex, then that thread stops blocking,1103obtains a lock on that mutex, and its CK\_LOCKMUTEX call returns.
- 1104oIf more than one thread was blocking on that particular mutex, then exactly one of the1105blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on1106the mutex, and its CK\_LOCKMUTEX call returns. All other threads blocking on that particular1107mutex continue to block.
- If a CK\_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call returns the error code CKR\_MUTEX\_NOT\_LOCKED.
- If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the behavior of the function call is undefined.

### 1112 **CK\_LOCKMUTEX** is defined as follows:

1113	typedef CK CALLBACK FUNCTION(CK RV, CK LOCKMUTEX)(
1114	CK_VOID_PTR pMutex
1115	);

		EX function is a pointer to the mutex object to be locked. Such a
117 118	The argument to a CK_LOCKMUTE function should return one of the fo	llowing values:
119 120 121 122	CKR_OK, CKR_GENERAL_ERROF CKR_HOST_MEMORY, CKR_MUTEX_BAD	ξ
123	CK_UNLOCKMUTEX is defined as	s follows:
124 125 126 127	<pre>typedef CK_CALLBACK_FUNCT CK_VOID_PTR pMutex );</pre>	TION(CK_RV, CK_UNLOCKMUTEX)(
128 129	The argument to a CK_UNLOCKM function should return one of the fo	UTEX function is a pointer to the mutex object to be unlocked. Such llowing values:
130 131 132 133	CKR_OK, CKR_GENERAL_ERROF CKR_HOST_MEMORY CKR_MUTEX_BAD CKR_MUTEX_NOT_LOCKED	2
134		S; CK_C_INITIALIZE_ARGS_PTR
135	CK_C_INITIALIZE_ARGS is a stru	icture containing the optional arguments for the <b>C. Initialize</b> function
136 137	For this version of Cryptoki, these c with threads. <b>CK_C_INITIALIZE_A</b>	ARGS is defined as follows:
136 137 138 139 140 141 142 143 144 145 146	For this version of Cryptoki, these c with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMu CK_FLAGS flags; CK_VOID_PTR pReserved; } CK_C_INITIALIZE_ARGS;	<pre>potional arguments are all concerned with the way the library deals ARGS is defined as follows: IALIZE_ARGS {     trex;     //utex;     itex;</pre>
136 137 138 139 140 141 142 143 144 145 146 147	For this version of Cryptoki, these of with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMu CK_FLAGS flags; CK_VOID_PTR pReserved; } CK_C_INITIALIZE_ARGS; The fields of the structure have the	following meanings:
136 137 138 139 140 141 142 143 144 145 146 147 148	For this version of Cryptoki, these of with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMu CK_FLAGS flags; CK_VOID_PTR pReserved; } CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex	following meanings: pointer to a function to use for creating mutex objects
136 137 138 139 140 141 142 143 144 145 146 147 148 149	For this version of Cryptoki, these c with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UNLOCKMU CK_FLAGS flags; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for destroying mutex objects
136 137 138 139 140 141 142 143 144 145 146 147 148 149 150	For this version of Cryptoki, these c with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMutex; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex LockMutex	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for locking mutex objects
136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151	For this version of Cryptoki, these of with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMu CK_FLAGS flags; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex LockMutex	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects
136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 151 152 153	For this version of Cryptoki, these c with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMutex; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex LockMutex UnlockMutex	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for unlocking mutex objects bit flags specifying options for <b>C_Initialize</b> ; the flags are defined below
136 137 138 139 140 141 142 143 144 145 146 147 148 147 148 149 150 151 152 153 154 155	For this version of Cryptoki, these of with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMutex; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex LockMutex flags pReserved	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for creating mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for unlocking mutex objects bit flags specifying options for <b>C_Initialize</b> ; the flags are defined below reserved for future use. Should be NULL_PTR for this version of Cryptoki
136 137 138 139 140 141 142 143 144 145 146 147 148 147 148 149 150 151 152 153 154 155 156	For this version of Cryptoki, these c with threads. CK_C_INITIALIZE_A typedef struct CK_C_INITI CK_CREATEMUTEX CreateMu CK_DESTROYMUTEX Destroy CK_LOCKMUTEX LockMutex; CK_UNLOCKMUTEX UnlockMutex; CK_VOID_PTR pReserved; CK_C_INITIALIZE_ARGS; The fields of the structure have the CreateMutex DestroyMutex LockMutex flags pReserved The following table defines the flags	following meanings: pointer to a function to use for creating mutex objects pointer to a function to use for destroying mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for locking mutex objects pointer to a function to use for unlocking mutex objects pointer to a function to use for unlocking mutex objects bit flags specifying options for <b>C_Initialize</b> ; the flags are defined below reserved for future use. Should be NULL_PTR for this version of Cryptoki s field:

Bit Flag	Mask	Meaning
CKF_LIBRARY_CANT_CREATE_OS_THREADS	0x0000001	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	0x0000002	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.

# 1159 **4 Objects**

1160 Cryptoki recognizes a number of classes of objects, as defined in the **CK\_OBJECT\_CLASS** data type.

1161 An object consists of a set of attributes, each of which has a given value. Each attribute that an object

1162 possesses has precisely one value. The following figure illustrates the high-level hierarchy of the 1163 Cryptoki objects and some of the attributes they support:



1164

1165 Figure 1, Object Attribute Hierarchy

1166 Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and 1167 modifying the values of their attributes. Some of the cryptographic functions (*e.g.*, **C\_GenerateKey**) also 1168 create key objects to hold their results.

1169 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

1171 contrasts with some object-based paradigms where an object has no attributes other than perhaps a

1172 class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

1173 Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the

1174 attribute value and the meaning of the attribute, which may include a default initial value. Some of the

1175 data types are defined explicitly by Cryptoki (*e.g.*, **CK\_OBJECT\_CLASS**). Attribute values may also take 1176 the following types:

1177	Byte array	an arbitrary string (array) of <b>CK_BYTE</b> s
1178 1179 1180	Big integer	a string of <b>CK_BYTE</b> s representing an unsigned integer of arbitrary size, most-significant byte first ( <i>e.g.</i> , the integer 32768 is represented as the 2-byte string 0x80 0x00)
1181 1182	Local string	an unpadded string of <b>CK_CHAR</b> s (see Table 3) with no null- termination
1183	RFC2279 string	an unpadded string of <b>CK_UTF8CHAR</b> s with no null-termination

- 1184 A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the 1185 same values for all their attributes.
- 1186 In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of
- 1187 Cryptoki attributes. Some of these attributes possess default values, and need not be specified when
- 1188 creating an object; some of these default values may even be the empty string (""). Nonetheless, the
- 1189 object possesses these attributes. A given object has a single value for each attribute it possesses, even
- 1190 if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.
- 1191 In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes 1192 whose meanings and values are not specified by Cryptoki.

# 1193 **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.

## 1201 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.
- 1206 Attempting to create an object with any of these functions requires an appropriate template to be 1207 supplied.
- 12081.If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the<br/>error code CKR\_ATTRIBUTE\_TYPE\_INVALID. An attribute is valid if it is either one of the attributes<br/>described in the Cryptoki specification or an additional vendor-specific attribute supported by the library<br/>and token.
- 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.
- 12153. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the<br/>error code CKR\_ATTRIBUTE\_READ\_ONLY. Whether or not a given Cryptoki attribute is read-only is<br/>explicitly stated in the Cryptoki specification; however, a particular library and token may be even more<br/>restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may<br/>nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of<br/>other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-<br/>only is obviously outside the scope of Cryptoki.
- 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.
- 1226 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.
- 1236 6. If the supplied template specifies the same value for a particular attribute more than once (or the 1237 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 1238 1239 create an object can either succeed—thereby creating the same object that would have been created 1240 if the multiply-specified attribute had only appeared once—or it can fail with error code 1241 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 1242 1243 encouraged never to put a particular attribute into a particular template more than once.
- 1244 If more than one of the situations listed above applies to an attempt to create an object, then the error 1245 code returned from the attempt can be any of the error codes from above that applies.

# 1246 **4.1.2 Modifying objects**

1247 Objects may be modified with the Cryptoki function **C\_SetAttributeValue** (see Section 5.7). The 1248 template supplied to **C\_SetAttributeValue** can contain new values for attributes which the object already 1249 possesses; values for attributes which the object does not yet possess; or both.

- 1250 Some attributes of an object may be modified after the object has been created, and some may not. In 1251 addition, attributes which Cryptoki specifies are modifiable may actually not be modifiable on some 1252 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 1253 support modification of some such attributes. Furthermore, whether or not a particular attribute of an 1254 object on a particular token is modifiable might depend on the values of certain attributes of the object. 1255 1256 For example, a secret key object's CKA SENSITIVE attribute can be changed from CK FALSE to 1257 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
- 1259 **C\_SetAttributeValue**, except for the possibility of a template being incomplete.

# 1260 **4.1.3 Copying objects**

1261 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-
- 1264 supplied template.

1265 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 1266 CKA\_TOKEN, CKA\_PRIVATE, CKA\_MODIFIABLE and CKA\_DESTROYABLE. To be more precise, 1267 1268 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 1269 attributes during the course of a C CopyObject operation. Furthermore, whether or not a particular 1270 attribute of an object on a particular token is modifiable during the course of a C CopyObject operation 1271 might depend on the values of certain attributes of the object. For example, a secret key object's 1272 CKA\_SENSITIVE attribute can be changed from CK FALSE to CK TRUE during the course of a 1273 1274 **C CopyObject** operation, but not the other way around.

- 1275 If the CKA\_COPYABLE attribute of the object to be copied is set to CK\_FALSE, C\_CopyObject returns
- 1276 CKR\_ACTION\_PROHIBITED. Otherwise, the scenarios described in 10.1.1 and the error codes they
- 1277 return apply to copying objects with C\_CopyObject, except for the possibility of a template being
- incomplete.

# 1279 4.2 Common attributes

1280 Table 11, Common footnotes for object attribute tables

<sup>1</sup> MUST be specified when object is created with **C\_CreateObject**.

<sup>2</sup> MUST *not* be specified when object is created with **C\_CreateObject**.

<sup>3</sup> MUST be specified when object is generated with **C\_GenerateKey** or **C\_GenerateKeyPair**.

 $^4$  MUST not be specified when object is generated with  $\textbf{C}\_\textbf{GenerateKey}$  or

### C\_GenerateKeyPair.

<sup>5</sup> MUST be specified when object is unwrapped with **C\_UnwrapKey**.

<sup>6</sup> MUST *not* be specified when object is unwrapped with **C\_UnwrapKey**.

<sup>7</sup> Cannot be revealed if object has its **CKA\_SENSITIVE** attribute set to CK\_TRUE or its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.

<sup>8</sup> 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.

<sup>9</sup> Default value is token-specific, and may depend on the values of other attributes.

<sup>10</sup> Can only be set to CK\_TRUE by the SO user.

<sup>11</sup> Attribute cannot be changed once set to CK\_TRUE. It becomes a read only attribute.

<sup>12</sup> Attribute cannot be changed once set to CK\_FALSE. It becomes a read only attribute.

### 1281

1282 Table 12, Common Object Attributes

Attribute	Data Type	Meaning
CKA_CLASS <sup>1</sup>	CK_OBJECT_CLASS	Object class (type)

1283 Refer to Table 11 for footnotes

1284 The above table defines the attributes common to all objects.

# 1285 **4.3 Hardware Feature Objects**

### 1286 **4.3.1 Definitions**

1287 This section defines the object class CKO\_HW\_FEATURE for type CK\_OBJECT\_CLASS as used in the 1288 CKA\_CLASS attribute of objects.

### 1289 **4.3.2 Overview**

1290 Hardware feature objects (**CKO\_HW\_FEATURE**) represent features of the device. They provide an easily 1291 expandable method for introducing new value-based features to the Cryptoki interface.

1292 When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, hardware feature objects are 1293 not returned unless the **CKA\_CLASS** attribute in the template has the value **CKO\_HW\_FEATURE**. This 1294 protects applications written to previous versions of Cryptoki from finding objects that they do not 1295 understand.

- 1296 Table 13, Hardware Feature Common Attributes

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE <sup>1</sup>	CK_HW_FEATURE_TYPE	Hardware feature (type)

1297 <sup>-</sup> Refer to Table 11 for footnotes

### 1298 **4.3.3 Clock**

### 1299 **4.3.3.1 Definition**

- 1300 The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_CLOCK of type
- 1301 CK\_HW\_FEATURE\_TYPE.

### 1302 4.3.3.2 Description

1303 Clock objects represent real-time clocks that exist on the device. This represents the same clock source 1304 as the **utcTime** field in the **CK\_TOKEN\_INFO** structure.

1305 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).

1306 The CKA\_VALUE attribute may be set using the C\_SetAttributeValue function if permitted by the 1307 device. The session used to set the time MUST be logged in. The device may require the SO to be the 1308 user logged in to modify the time value. C\_SetAttributeValue will return the error

1309 CKR\_USER\_NOT\_LOGGED\_IN to indicate that a different user type is required to set the value.

### 1310 4.3.4 Monotonic Counter Objects

### 1311 4.3.4.1 Definition

- 1312 The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_MONOTONIC\_COUNTER of type
- 1313 CK\_HW\_FEATURE\_TYPE.

### 1314 **4.3.4.2 Description**

1315 Monotonic counter objects represent hardware counters that exist on the device. The counter is

- 1316 guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an 1317 application for generating serial numbers to get some assurance of uniqueness per token.
- 1318 Table 15, Monotonic Counter Attributes

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

- 1319 <sup>1</sup>Read Only
- 1320 The **CKA\_VALUE** attribute may not be set by the client.

### 1321 **4.3.5 User Interface Objects**

### 1322 **4.3.5.1 Definition**

1323The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_USER\_INTERFACE of type1324CK\_HW\_FEATURE\_TYPE.

### 1325 **4.3.5.2 Description**

- 1326 User interface objects represent the presentation capabilities of the device.
- 1327 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".

- 1328 The selection of attributes, and associated data types, has been done in an attempt to stay as aligned 1329 with RFC 2534 and CC/PP Struct as possible. The special value CK\_UNAVAILABLE\_INFORMATION 1330 may be used for CK\_ULONG-based attributes when information is not available or applicable.
- 1331 None of the attribute values may be set by an application.
- 1332 The value of the **CKA\_ENCODING\_METHODS** attribute may be used when the application needs to 1333 send MIME objects with encoded content to the token.

# 1334 4.4 Storage Objects

1335 This is not an object class; hence no CKO\_ definition is required. It is a category of object classes with 1336 common attributes for the object classes that follow.

### 1337 Table 17, Common Storage Object Attributes

Attribute	Data Type	Meaning
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_COPYABLE	CK_BBOOL	CK_TRUE if object can be copied using C_CopyObject. Defaults to CK_TRUE. Can't be set to TRUE once it is set to FALSE.
CKA_DESTROYABLE	CK_BBOOL	CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.
CKA_UNIQUE_ID <sup>246</sup>	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**

- Any time a new object is created, a value for CKA\_UNIQUE\_ID MUST be generated by the token and stored with the object. The specific algorithm used to generate unique ID values for objects is tokenspecific, but values generated MUST be unique across all objects visible to any particular session, and SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling C InitToken, MAY cause reuse of CKA\_UNIQUE\_ID values.
- Any attempt to modify the CKA\_UNIQUE\_ID attribute of an existing object or to specify the value of the 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

# 1361 **4.5 Data objects**

### 1362 **4.5.1 Definitions**

1363 This section defines the object class CKO\_DATA for type CK\_OBJECT\_CLASS as used in the 1364 CKA CLASS attribute of objects.

### 1365 **4.5.2 Overview**

1366 Data objects (object class **CKO\_DATA**) hold information defined by an application. Other than providing 1367 access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the 1368 attributes supported by data objects, in addition to the common attributes defined for this object class:

1369 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)

1370 The **CKA\_APPLICATION** attribute provides a means for applications to indicate ownership of the data 1371 objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has

1372 access to a data object, however.

1373 The **CKA\_OBJECT\_ID** attribute provides an application independent and expandable way to indicate the 1374 type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier 1375 matches the data value.

1376 The following is a sample template containing attributes for creating a data object:

```
1377
           CK OBJECT CLASS class = CKO DATA;
1378
           CK_UTF8CHAR label[] = "A data object";
1379
           CK_UTF8CHAR application[] = "An application";
1380
           CK BYTE data[] = "Sample data";
1381
           CK BBOOL true = CK TRUE;
1382
           CK ATTRIBUTE template[] = {
1383
              {CKA_CLASS, &class, sizeof(class)},
1384
              {CKA_TOKEN, &true, sizeof(true)},
1385
              {CKA_LABEL, label, sizeof(label)-1},
1386
              {CKA_APPLICATION, application, sizeof(application)-1},
1387
              {CKA VALUE, data, sizeof(data)}
1388
            };
```

1389

# 1390 **4.6 Certificate objects**

### 1391 **4.6.1 Definitions**

1392 This section defines the object class CKO\_CERTIFICATE for type CK\_OBJECT\_CLASS as used in the 1393 CKA\_CLASS attribute of objects.

### 1394 **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:

### 1399 Table 19, Common Certificate Object Attributes

Attribute	Data type	Meaning
CKA_CERTIFICATE_TYPE <sup>1</sup>	CK_CERTIFICATE_TYPE	Type of certificate
CKA_TRUSTED <sup>10</sup>	CK_BBOOL	The certificate can be trusted for the application that it was created.
CKA_CERTIFICATE_CATEGORY	CKA_CERTIFICATE_CATEGORY	(default CK_CERTIFICATE_ CATEGORY_UNSP ECIFIED)
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 SubjectPublicKeyInf o for the public key contained in this certificate (default empty)

1400 <sup>-</sup> 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:
- X.509 public key certificate
- WTLS public key certificate
- 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
- attach any special meaning to them. When present, the application is responsible to set them to valuesthat match the certificate's encoded "not before" and "not after" fields (if any).

# 1421 4.6.3 X.509 public key certificate objects

1422 X.509 certificate objects (certificate type CKC\_X\_509) hold X.509 public key certificates. The following
 1423 table defines the X.509 certificate object attributes, in addition to the common attributes defined for this
 1424 object class:

1425 Table 20, X.509 Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT <sup>1</sup>	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 <sup>2</sup>	Byte array	BER-encoding of the certificate
CKA_URL <sup>3</sup>	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
CKA_HASH_OF_SUBJECT_PUB LIC_KEY <sup>4</sup>	Byte array	Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLI C_KEY <sup>4</sup>	Byte array	Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_JAVA_MIDP_SECURITY_D OMAIN	CK_JAVA_ MIDP_SEC URITY_DO MAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECI FIED)
CKA_NAME_HASH_ALGORITH M	CK_MECH ANISM_TY PE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC _KEY and CKA_HASH_OF_ISSUER_PUBLIC_K EY. If the attribute is not present then the type defaults to SHA-1.

- <sup>1</sup>426 <sup>1</sup>MUST be specified when the object is created.
- <sup>2</sup>MUST be specified when the object is created. MUST be non-empty if CKA\_URL is empty.
- <sup>3</sup>MUST be non-empty if CKA\_VALUE is empty.
- 1429 <sup>4</sup>Can only be empty if CKA\_URL is empty.
- 1430 Only the **CKA\_ID**, **CKA\_ISSUER**, and **CKA\_SERIAL\_NUMBER** attributes may be modified after the 1431 object is created.
- 1432 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
- 1435 **CKA\_ID** value without introducing any ambiguity.)
- 1436 It is intended in the interests of interoperability that the subject name and key identifier for a certificate will 1437 be the same as those for the corresponding public and private keys (though it is not required that all be

1438 stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness 1439 of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

1440 The CKA\_ISSUER and CKA\_SERIAL\_NUMBER attributes are for compatibility with PKCS #7 and 1441 Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key 1442 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. 1443

1444 The CKA URL attribute enables the support for storage of the URL where the certificate can be found 1445 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile 1446 environments.

1447 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 1448

particularly important when only the URL is available to be able to correlate a certificate with a private key 1449 and when searching for the certificate of the issuer. The hash algorithm is defined by 1450

1451 CKA NAME HASH ALGORITHM.

1452 The CKA JAVA MIDP SECURITY DOMAIN attribute associates a certificate with a Java MIDP security 1453 domain.

1454 The following is a sample template for creating an X.509 certificate object:

```
CK OBJECT CLASS class = CKO CERTIFICATE;
1455
1456
           CK CERTIFICATE TYPE certType = CKC X 509;
1457
           CK_UTF8CHAR label[] = "A certificate object";
1458
           CK BYTE subject[] = {...};
1459
           CK BYTE id[] = {123};
1460
           CK BYTE certificate[] = {...};
1461
           CK BBOOL true = CK TRUE;
1462
           CK ATTRIBUTE template[] = {
1463
              {CKA CLASS, &class, sizeof(class)},
1464
              {CKA CERTIFICATE TYPE, &certType, sizeof(certType)};
1465
              {CKA TOKEN, &true, sizeof(true)},
1466
              {CKA_LABEL, label, sizeof(label)-1},
1467
              {CKA SUBJECT, subject, sizeof(subject)},
1468
              {CKA ID, id, sizeof(id)},
1469
              {CKA VALUE, certificate, sizeof(certificate)}
1470
           };
```

#### 4.6.4 WTLS public key certificate objects 1471

WTLS certificate objects (certificate type CKC\_WTLS) hold WTLS public key certificates. The following 1472 table defines the WTLS certificate object attributes, in addition to the common attributes defined for this 1473 1474 object class.

Attribute	Data type	Meaning
CKA_SUBJECT <sup>1</sup>	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 <sup>2</sup>	Byte array	WTLS-encoding of the certificate
CKA_URL <sup>3</sup>	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained
CKA_HASH_OF_SUBJECT_PU BLIC_KEY⁴	Byte array	SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM

1475 Table 21: WTLS Certificate Object Attributes

Attribute	Data type	Meaning
CKA_HASH_OF_ISSUER_PUB LIC_KEY <sup>4</sup>	Byte array	SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_NAME_HASH_ALGORITH M	CK_MECHANI SM_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.

- <sup>1</sup>MUST be specified when the object is created. Can only be empty if CKA\_VALUE is empty.
- <sup>2</sup>MUST be specified when the object is created. MUST be non-empty if CKA\_URL is empty.
- <sup>3</sup>MUST be non-empty if CKA\_VALUE is empty.
- <sup>4</sup>Can only be empty if CKA\_URL is empty.
- 1480

1481 Only the **CKA\_ISSUER** attribute may be modified after the object has been created.

- 1482 The encoding for the CKA\_SUBJECT, CKA\_ISSUER, and CKA\_VALUE attributes can be found in 1483 [WTLS].
- 1484 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 mobileenvironments.
- 1487 The CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY and CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY
- 1488 attributes are used to store the hashes of the public keys of the subject and the issuer. They are
- 1489 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
- 1491 CKA\_NAME\_HASH\_ALGORITHM.
- 1492 The following is a sample template for creating a WTLS certificate object:

```
1493
            CK OBJECT CLASS class = CKO CERTIFICATE;
1494
            CK CERTIFICATE TYPE certType = CKC WTLS;
1495
            CK_UTF8CHAR label[] = "A certificate object";
1496
            CK BYTE subject[] = { ... };
1497
            CK BYTE certificate[] = {...};
1498
            CK BBOOL true = CK TRUE;
1499
            CK ATTRIBUTE template[] =
1500
1501
              {CKA CLASS, &class, sizeof(class)},
1502
              {CKA CERTIFICATE TYPE, &certType, sizeof(certType)};
1503
              {CKA TOKEN, &true, sizeof(true)},
1504
              {CKA LABEL, label, sizeof(label)-1},
1505
              {CKA SUBJECT, subject, sizeof(subject)},
1506
              {CKA VALUE, certificate, sizeof(certificate)}
1507
           };
```

### 1508 **4.6.5 X.509 attribute certificate objects**

- 1509 X.509 attribute certificate objects (certificate type **CKC\_X\_509\_ATTR\_CERT**) hold X.509 attribute 1510 certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the
- 1511 common attributes defined for this object class:

### 1512 Table 22, X.509 Attribute Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_OWNER <sup>1</sup>	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 <sup>1</sup>	Byte Array	BER-encoding of the certificate.

<sup>1</sup>MUST 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:

```
1517
           CK_OBJECT_CLASS class = CKO CERTIFICATE;
1518
           CK CERTIFICATE TYPE certType = CKC X 509 ATTR CERT;
1519
           CK_UTF8CHAR label[] = "An attribute certificate object";
1520
           CK BYTE owner[] = {...};
1521
           CK BYTE certificate[] = {...};
1522
           CK_BBOOL true = CK TRUE;
1523
           CK ATTRIBUTE template[] = {
1524
              {CKA CLASS, &class, sizeof(class)},
1525
             {CKA CERTIFICATE TYPE, &certType, sizeof(certType)};
1526
             {CKA_TOKEN, &true, sizeof(true)},
1527
             {CKA LABEL, label, sizeof(label)-1},
1528
             {CKA OWNER, owner, sizeof(owner)},
1529
              {CKA VALUE, certificate, sizeof(certificate)}
1530
           };
```

# 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 <sup>1,5</sup>	CK_KEY_TYPE	Type of key
CKA_ID <sup>8</sup>	Byte array	Key identifier for key (default empty)
CKA_START_DATE <sup>8</sup>	CK_DATE	Start date for the key (default empty)
CKA_END_DATE <sup>8</sup>	CK_DATE	End date for the key (default empty)
CKA_DERIVE <sup>8</sup>	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 <sup>2,4,6</sup>	CK_BBOOL	CK_TRUE only if key was either
		<ul> <li>generated locally (<i>i.e.</i>, on the token) with a C_GenerateKey or C_GenerateKeyPair call</li> </ul>
		<ul> <li>created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE</li> </ul>
CKA_KEY_GEN_ MECHANISM <sup>2,4,6</sup>	CK_MECHANISM _TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANI SMS	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 <sup>-</sup> 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 attributes common to all public keys, in addition to the common attributes defined for this object class:

### 1563 Table 24, Common Public Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT <sup>8</sup>	Byte array	DER-encoding of the key subject name (default empty)
CKA_ENCRYPT <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports encryption <sup>9</sup>
CKA_VERIFY <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports verification where the signature is an appendix to the data <sup>9</sup>
CKA_VERIFY_RECOVER <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports verification where the data is recovered from the signature <sup>9</sup>
CKA_WRAP <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports wrapping ( <i>i.e.</i> , can be used to wrap other keys) <sup>9</sup>
CKA_TRUSTED <sup>10</sup>	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 <sup>-</sup> 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 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	subjectPublicK	ey 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 <sup>8</sup>	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SENSITIVE <sup>8,11</sup>	CK_BBOOL	CK_TRUE if key is sensitive <sup>9</sup>
CKA_DECRYPT <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports decryption <sup>9</sup>
CKA_SIGN <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports signatures where the signature is an appendix to the data <sup>9</sup>
CKA_SIGN_RECOVER <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports signatures where the data can be recovered from the signature <sup>9</sup>
CKA_UNWRAP <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports unwrapping ( <i>i.e.</i> , can be used to unwrap other keys) <sup>9</sup>
CKA_EXTRACTABLE <sup>8,12</sup>	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped <sup>9</sup>
CKA_ALWAYS_SENSITIVE <sup>2,4,6</sup>	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE <sup>2,4,6</sup>	CK_BBOOL	CK_TRUE if key has <i>never</i> had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_WRAP_WITH_TRUSTED <sup>11</sup>	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 <sup>8</sup>	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)

1581 <sup>-</sup> Refer to Table 11 for footnotes

1582 It is intended in the interests of interoperability that the subject name and key identifier for a private key 1583 will be the same as those for the corresponding certificate and public key. However, this is not enforced 1584 by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

1585 If the CKA\_SENSITIVE attribute is CK\_TRUE, or if the CKA\_EXTRACTABLE attribute is CK\_FALSE,
1586 then certain attributes of the private key cannot be revealed in plaintext outside the token. Which
1587 attributes these are is specified for each type of private key in the attribute table in the section describing
1588 that type of key.

1589 The **CKA\_ALWAYS\_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user 1590 to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such 1591 as sign or decrypt. This attribute may only be set to CK\_TRUE when **CKA\_PRIVATE** is also CK\_TRUE.

1592 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
- 1603 CKR\_USER\_NOT\_LOGGED\_IN to be returned from calls using the key. C\_Login will return
- 1604 CKR\_OPERATION\_NOT\_INITIALIZED, but the active cryptographic operation will not be affected, if an
- 1605 attempt is made to re-authenticate when CKA\_ALWAYS\_AUTHENTICATE is set to CK\_FALSE.

- 1606 The CKA\_PUBLIC\_KEY\_INFO attribute represents the public key associated with this private key. The 1607 data it represents may either be stored as part of the private key data, or regenerated as needed from the 1608 private kev.
- 1609 If this attribute is supplied as part of a template for C CreateObject, C CopyObject or
- 1610 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 1611
- 1612 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 1613
- token MAY choose not to support the CKA PUBLIC KEY INFO attribute for commands which create 1614
- 1615 new private keys. If it does not support the attribute, the command SHALL return

#### 1616 CKR ATTRIBUTE TYPE INVALID.

1617 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 1618 1619 the CKA PUBLIC EXPONENT to the list of attributes required for an RSA private key. All other private 1620 key types described in this specification contain sufficient information to recover the associated public 1621 key.

#### 4.9.1 RSA private key objects 1622

- 1623 RSA private key objects (object class CKO\_PRIVATE\_KEY, key type CKK\_RSA) hold RSA private keys. 1624 The following table defines the RSA private key object attributes, in addition to the common attributes
- 1625 defined for this object class:
- Table 27, RSA Private Key Object Attributes 1626

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

- 1627 Refer to Table 11 for footnotes
- 1628 Depending on the token, there may be limits on the length of the key components. See PKCS #1 for more information on RSA keys.
- 1629
- 1630 Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above 1631 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 1632 1633 also store CKA PUBLIC EXPONENT. This permits the retrieval of sufficient data to reconstitute the associated public key. 1634
- 1635 Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an 1636 RSA private key, it stores whichever of the fields in Table 27 it keeps track of. Later, if an application 1637 asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose 1638 values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request 1639 fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various 1640 attributes of RSA private keys which are not actually stored on the token. E.g., if a particular token stores 1641 values only for the CKA PRIVATE EXPONENT, CKA PUBLIC EXPONENT, CKA PRIME 1, and 1642 CKA PRIME 2 attributes, then Cryptoki is certainly able to report values for all the attributes above
- 1643 (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 1644

- 1645 implementation is *required* to be able to return values are **CKA\_MODULUS**,
- 1646 CKA\_PRIVATE\_EXPONENT, and CKA\_PUBLIC\_EXPONENT. A token SHOULD also be able to return
- 1647 **CKA\_PUBLIC\_KEY\_INFO** for an RSA private key. See the general guidance for Private Keys above.

# 1648 **4.10 Secret key objects**

- 1649 Secret key objects (object class **CKO\_SECRET\_KEY**) hold secret keys. The following table defines the 1650 attributes common to all secret keys, in addition to the common attributes defined for this object class:
- 1651 Table 28, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE <sup>8,11</sup>	CK_BBOOL	CK_TRUE if object is sensitive (default CK_FALSE)
CKA_ENCRYPT <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports encryption <sup>9</sup>
CKA_DECRYPT <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports decryption <sup>9</sup>
CKA_SIGN <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports signatures ( <i>i.e.</i> , authentication codes) where the signature is an appendix to the data <sup>9</sup>
CKA_VERIFY <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports verification ( <i>i.e.</i> , of authentication codes) where the signature is an appendix to the data <sup>9</sup>
CKA_WRAP <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports wrapping ( <i>i.e.</i> , can be used to wrap other keys) <sup>9</sup>
CKA_UNWRAP <sup>8</sup>	CK_BBOOL	CK_TRUE if key supports unwrapping ( <i>i.e.</i> , can be used to unwrap other keys) <sup>9</sup>
CKA_EXTRACTABLE <sup>8,12</sup>	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped <sup>9</sup>
CKA_ALWAYS_SENSITIVE <sup>2,4,6</sup>	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE <sup>2,4,6</sup>	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 <sup>11</sup>	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 <sup>10</sup>	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 <sup>-</sup> 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.
- Properties: 1662
- 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.

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

1670 If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST match what the library calculates it to be or the library returns a CKR ATTRIBUTE VALUE INVALID. If 1671 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.
- The generation of the KCV may be prevented by the application supplying the attribute in the template as 1675 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.
- Unless otherwise specified for the object definition, the value of this attribute is derived from the key 1678
- 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.

# 1681 4.11 Domain parameter objects

### 1682 **4.11.1 Definitions**

1683 This section defines the object class CKO\_DOMAIN\_PARAMETERS for type CK\_OBJECT\_CLASS as 1684 used in the CKA\_CLASS attribute of objects.

### 1685 **4.11.2 Overview**

- 1686 This object class was created to support the storage of certain algorithm's extended parameters. DSA 1687 and DH both use domain parameters in the key-pair generation step. In particular, some libraries support 1688 the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.
- 1689 To use a domain parameter object you MUST extract the attributes into a template and supply them (still 1690 in the template) to the corresponding key-pair generation function.
- 1691 Domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**) hold public domain parameters.
- 1692 The following table defines the attributes common to domain parameter objects in addition to the common 1693 attributes defined for this object class:
- 1694 Table 29, Common Domain Parameter Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE <sup>1</sup>	CK_KEY_TYPE	Type of key the domain parameters can be used to generate.
CKA_LOCAL <sup>2,4</sup>	CK_BBOOL	CK_TRUE only if domain parameters were either
		with a C_GenerateKey
		<ul> <li>created with a C_CopyObject call as a copy of domain parameters which had its CKA_LOCAL attribute set to CK_TRUE</li> </ul>

- 1695 <sup>-</sup> Refer to Table 11 for footnotes
- 1696 The **CKA\_LOCAL** attribute has the value CK\_TRUE if and only if the values of the domain parameters 1697 were originally generated on the token by a **C\_GenerateKey** call.

# 1698 **4.12 Mechanism objects**

### 1699 **4.12.1 Definitions**

1700 This section defines the object class CKO\_MECHANISM for type CK\_OBJECT\_CLASS as used in the 1701 CKA\_CLASS attribute of objects.

### 1702 **4.12.2 Overview**

- 1703 Mechanism objects provide information about mechanisms supported by a device beyond that given by 1704 the **CK\_MECHANISM\_INFO** structure.
- 1705 When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, mechanism objects are not
- 1706 returned unless the CKA\_CLASS attribute in the template has the value CKO\_MECHANISM. This
- 1707 protects applications written to previous versions of Cryptoki from finding objects that they do not
- 1708 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**

Profile objects (object class CKO\_PROFILE) describe which PKCS #11 profiles the token implements.
Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given
token can contain more than one profile ID. The following table lists the attributes supported by profile
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.

# 1723 **5 Functions**

- 1724 Cryptoki's functions are organized into the following categories:
- 1725 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)
- 1733 signing and MACing functions (6 functions)
- 1734 functions for verifying signatures and MACs (6 functions)
- dual-purpose cryptographic functions (4 functions)
- 1736 key management functions (5 functions)
- random number generation functions (2 functions)
- parallel function management functions (2 functions)
- 1739

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.

- 1743 The Cryptoki API functions are presented in the following table:
- 1744 Table 32, Summary of Cryptoki Functions

Category	Function	Description
General	C_Initialize	initializes Cryptoki
purpose functions	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	C_GetSlotList	obtains a list of slots in the system
management	C_GetSlotInfo	obtains information about a particular slot
functions	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	C_CreateObject	creates an object
management	C_CopyObject	creates a copy of an object
functions	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	C_EncryptInit	initializes an encryption operation
functions	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	C_MessageEncryptInit	initializes a message-based encryption process
Functions	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	C_DecryptInit	initializes a decryption operation
Functions	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	C_MessageDecryptInit	initializes a message decryption operation
Decryption	C_DecryptMessage	decrypts single-part data
Functions	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	C_DigestInit	initializes a message-digesting operation
Digesting	C_Digest	digests single-part data
Functions	C_DigestUpdate	continues a multiple-part digesting operation
	C_DigestKey	digests a key
	C_DigestFinal	finishes a multiple-part digesting operation
Signing	C_SignInit	initializes a signature operation
and MACing	C_Sign	signs single-part data
functions	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	C_MessageSignInit	initializes a message signature operation
Signature	C_SignMessage	signs single-part data
Tunctions	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	C_VerifyInit	initializes a verification operation
signatures	C_Verify	verifies a signature on single-part data
and MACs	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	C_MessageVerifyInit	initializes a message verification operation
Functions for	C_VerifyMessage	verifies single-part data
signatures and	C_VerifyMessageBegin	starts a multiple-part message verification operation
MAGS	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation
	C_MessageVerifyFinal	finishes a message verification operation
Dual-purpose cryptographic	C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations
functions	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
Кеу	C_GenerateKey	generates a secret key
management	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	C_SeedRandom	mixes in additional seed material to the random number generator
functions	C_GenerateRandom	generates random data
Parallel function management	C_GetFunctionStatus	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
functions	C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
Callback function		application-supplied function to process notifications from Cryptoki

1745

- 1746 Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes 1747 either its entire goal, or nothing at all.
- 1748 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.
- 1759 A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported 1760 function MUST have a "stub" in the library which simply returns the value
- 1761 CKR FUNCTION NOT SUPPORTED. The function's entry in the library's **CK FUNCTION LIST**
- 1762 structure (as obtained by **C\_GetFunctionList**) should point to this stub function (see Section 3.6).

# 1763 **5.1 Function return values**

1764 The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we 1765 enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1 1766 details the behavior of Cryptoki functions, including what values each of them may return.

1767 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 1768 specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are 1769 presumably some gaps. For example, it is possible that a particular error code which might apply to a 1770 particular Cryptoki function is unfortunately not actually listed in the description of that function as a 1771 possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit 1772 his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful 1773 1774 if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that 1775 error code for that function. It would be far preferable for the application to examine the function's return 1776 value, see that it indicates some sort of error (even if the application doesn't know precisely what kind of 1777 error), and behave accordingly.

1778 See Section 5.1.8 for some specific details on how a developer might attempt to make an application that 1779 accommodates a range of behaviors from Cryptoki libraries.

# 1780 **5.1.1 Universal Cryptoki function return values**

- 1781 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.
- 1797 The relative priorities of these errors are in the order listed above, *e.g.*, if either of
- 1798 CKR\_GENERAL\_ERROR or CKR\_HOST\_MEMORY would be an appropriate error return, then
- 1799 CKR\_GENERAL\_ERROR should be returned.

# 1800 5.1.2 Cryptoki function return values for functions that use a session 1801 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,
- 1904 C\_OPENSESSION and C. Close All Sessions) can return the following values:
- $1805 \qquad {\sf C\_OpenSession, and C\_CloseAllSessions) can return the following values:}$
- 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.
- 1816 The relative priorities of these errors are in the order listed above, *e.g.*, if either of
- 1817 CKR\_SESSION\_HANDLE\_INVALID or CKR\_DEVICE\_REMOVED would be an appropriate error return, 1818 then CKR\_SESSION\_HANDLE\_INVALID should be returned.
- 1819 In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction 1820 between a token being removed *before* a function invocation and a token being removed *during* a
- 1821 function execution.

### 1822 **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:

 1826 • CKR\_DEVICE\_MEMORY: The token does not have sufficient memory to perform the requested function.

- CKR\_DEVICE\_ERROR: Some problem has occurred with the token and/or slot. This error code can be returned by more than just the functions mentioned above; in particular, it is possible for
   C\_GetSlotInfo to return CKR\_DEVICE\_ERROR.
- 1831 CKR\_TOKEN\_NOT\_PRESENT: The token was not present in its slot at the time that the function was 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:
- CKR\_CANCEL: When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a CKN\_SURRENDER
   callback (see Section 5.21). If the callback returns the value CKR\_CANCEL, then the function aborts and returns CKR\_FUNCTION\_CANCELED.

# 1847 **5.1.5** Special return values for mutex-handling functions

- There are two other special-purpose return values which are not returned by any actual Cryptoki
  functions. These values may be returned by application-supplied mutex-handling functions, and they may
  safely be ignored by application developers who are not using their own threading model. They are:
- CKR\_MUTEX\_BAD: This error code can be returned by mutex-handling functions that are passed a bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a bad mutex object. There is therefore no guarantee that such a function will successfully detect bad mutex objects and return this value.
- CKR\_MUTEX\_NOT\_LOCKED: This error code can be returned by mutex-unlocking functions. It indicates that the mutex supplied to the mutex-unlocking function was not locked.

# 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.
- CKR\_ACTION\_PROHIBITED: This value can only be returned by C\_CopyObject,
   C\_SetAttributeValue and C\_DestroyObject. It denotes that the action may not be taken, either
   because of underlying policy restrictions on the token, or because the object has the relevant
   CKA\_COPYABLE, CKA\_MODIFIABLE or CKA\_DESTROYABLE policy attribute set to CK\_FALSE.
- CKR\_ARGUMENTS\_BAD: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate.
- CKR\_ATTRIBUTE\_READ\_ONLY: An attempt was made to set a value for an attribute which may not be set by the application, or which may not be modified by the application. See Section 4.1 for more information.
- CKR\_ATTRIBUTE\_SENSITIVE: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or un-extractable.

- 1873 CKR\_ATTRIBUTE\_TYPE\_INVALID: An invalid attribute type was specified in a template. See 1874 Section 4.1 for more information.
- CKR\_ATTRIBUTE\_VALUE\_INVALID: An invalid value was specified for a particular attribute in a 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.
- CKR\_CANT\_LOCK: This value can only be returned by C\_Initialize. It means that the type of locking requested by the application for thread-safety is not available in this library, and so the application cannot make use of this library in the specified fashion.
- CKR\_CRYPTOKI\_ALREADY\_INITIALIZED: This value can only be returned by C\_Initialize. It
   means that the Cryptoki library has already been initialized (by a previous call to C\_Initialize which
   did not have a matching C\_Finalize call).
- CKR\_CRYPTOKI\_NOT\_INITIALIZED: This value can be returned by any function other than
   C\_Initialize, C\_GetFunctionList, C\_GetInterfaceList and C\_GetInterface. It indicates that the
   function cannot be executed because the Cryptoki library has not yet been initialized by a call to
   C\_Initialize.
- CKR\_CURVE\_NOT\_SUPPORTED: This curve is not supported by this token. Used with Elliptic
   Curve mechanisms.
- CKR\_DATA\_INVALID: The plaintext input data to a cryptographic operation is invalid. This return value has lower priority than CKR\_DATA\_LEN\_RANGE.
- CKR\_DATA\_LEN\_RANGE: The plaintext input data to a cryptographic operation has a bad length.
   Depending on the operation's mechanism, this could mean that the plaintext data is too short, too
   long, or is not a multiple of some particular block size. This return value has higher priority than
   CKR\_DATA\_INVALID.
- CKR\_DOMAIN\_PARAMS\_INVALID: Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.
- CKR\_ENCRYPTED\_DATA\_INVALID: The encrypted input to a decryption operation has been determined to be invalid ciphertext. This return value has lower priority than CKR\_ENCRYPTED\_DATA\_LEN\_RANGE.
- CKR\_ENCRYPTED\_DATA\_LEN\_RANGE: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular block size. This return value has higher priority than CKR\_ENCRYPTED\_DATA\_INVALID.
- CKR\_EXCEEDED\_MAX\_ITERATIONS: An iterative algorithm (for key pair generation, domain parameter generation etc.) failed because we have exceeded the maximum number of iterations.
   This error code has precedence over CKR\_FUNCTION\_FAILED. Examples of iterative algorithms include DSA signature generation (retry if either r = 0 or s = 0) and generation of DSA primes p and q specified in FIPS 186-4.
- CKR\_FIPS\_SELF\_TEST\_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed. The token entered an error state. Future calls to cryptographic functions on the token will return
   CKR\_GENERAL\_ERROR. CKR\_FIPS\_SELF\_TEST\_FAILED has a higher precedence over
   CKR\_GENERAL\_ERROR. This error may be returned by C\_Initialize, if a power-up self-test failed,
   by C\_GenerateRandom or C\_SeedRandom, if the continuous random number generator test failed,
   or by C\_GenerateKeyPair, if the pair-wise consistency test failed.
- CKR\_FUNCTION\_CANCELED: The function was canceled in mid-execution. This happens to a cryptographic function if the function makes a CKN\_SURRENDER application callback which returns CKR\_CANCEL (see CKR\_CANCEL). It also happens to a function that performs PIN entry through a protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- CKR\_FUNCTION\_NOT\_PARALLEL: There is currently no function executing in parallel in the specified session. This is a legacy error code which is only returned by the legacy functions
   C\_GetFunctionStatus and C\_CancelFunction.

- CKR\_FUNCTION\_NOT\_SUPPORTED: The requested function is not supported by this Cryptoki library. Even unsupported functions in the Cryptoki API should have a "stub" in the library; this stub should simply return the value CKR\_FUNCTION\_NOT\_SUPPORTED.
- 1927 CKR\_FUNCTION\_REJECTED: The signature request is rejected by the user.
- CKR\_INFORMATION\_SENSITIVE: The information requested could not be obtained because the token considers it sensitive, and is not able or willing to reveal it.
- CKR\_KEY\_CHANGED: This value is only returned by C\_SetOperationState. It indicates that one of the keys specified is not the same key that was being used in the original saved session.
- CKR\_KEY\_FUNCTION\_NOT\_PERMITTED: An attempt has been made to use a key for a cryptographic purpose that the key's attributes are not set to allow it to do. For example, to use a key for performing encryption, that key MUST have its CKA\_ENCRYPT attribute set to CK\_TRUE (the fact that the key MUST have a CKA\_ENCRYPT attribute implies that the key cannot be a private key). This return value has lower priority than CKR\_KEY\_TYPE\_INCONSISTENT.
- CKR\_KEY\_HANDLE\_INVALID: The specified key handle is not valid. It may be the case that the specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a valid key handle.
- CKR\_KEY\_INDIGESTIBLE: This error code can only be returned by C\_DigestKey. It indicates that the value of the specified key cannot be digested for some reason (perhaps the key isn't a secret key, or perhaps the token simply can't digest this kind of key).
- CKR\_KEY\_NEEDED: This value is only returned by C\_SetOperationState. It indicates that the session state cannot be restored because C\_SetOperationState needs to be supplied with one or more keys that were being used in the original saved session.
- CKR\_KEY\_NOT\_NEEDED: An extraneous key was supplied to C\_SetOperationState. For
   example, an attempt was made to restore a session that had been performing a message digesting
   operation, and an encryption key was supplied.
- CKR\_KEY\_NOT\_WRAPPABLE: Although the specified private or secret key does not have its
   CKA\_EXTRACTABLE attribute set to CK\_FALSE, Cryptoki (or the token) is unable to wrap the key as
   requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping
   key specified is not one of these types). Compare with CKR\_KEY\_UNEXTRACTABLE.
- CKR\_KEY\_SIZE\_RANGE: Although the requested keyed cryptographic operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the range of key sizes that it can handle.
- CKR\_KEY\_TYPE\_INCONSISTENT: The specified key is not the correct type of key to use with the specified mechanism. This return value has a higher priority than
   CKR\_KEY\_FUNCTION\_NOT\_PERMITTED.
- CKR\_KEY\_UNEXTRACTABLE: The specified private or secret key can't be wrapped because its
   CKA\_EXTRACTABLE attribute is set to CK\_FALSE. Compare with CKR\_KEY\_NOT\_WRAPPABLE.
- **•** CKR\_LIBRARY\_LOAD\_FAILED: The Cryptoki library could not load a dependent shared library.
- CKR\_MECHANISM\_INVALID: An invalid mechanism was specified to the cryptographic operation.
   This error code is an appropriate return value if an unknown mechanism was specified or if the mechanism specified cannot be used in the selected token with the selected function.
- CKR\_MECHANISM\_PARAM\_INVALID: Invalid parameters were supplied to the mechanism specified to the cryptographic operation. Which parameter values are supported by a given mechanism can vary from token to token.
- CKR\_NEED\_TO\_CREATE\_THREADS: This value can only be returned by C\_Initialize. It is returned when two conditions hold:
- The application called **C\_Initialize** in a way which tells the Cryptoki library that application threads executing calls to the library cannot use native operating system methods to spawn new threads.

- 19732. The library cannot function properly without being able to spawn new threads in the above<br/>fashion.
- CKR\_NO\_EVENT: This value can only be returned by C\_WaitForSlotEvent. It is returned when
   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 is never a valid object handle.
- CKR\_OPERATION\_ACTIVE: There is already an active operation (or combination of active operations) which prevents Cryptoki from activating the specified operation. For example, an active object-searching operation would prevent Cryptoki from activating an encryption operation with
   C\_EncryptInit. Or, an active digesting operation and an active encryption operation would prevent Cryptoki from activating as ignature operation. Or, on a token which doesn't support simultaneous dual cryptographic operations in a session (see the description of the
   CKF\_DUAL\_CRYPTO\_OPERATIONS flag in the CK\_TOKEN\_INFO structure), an active signature
- 1985CKF\_DUAL\_CRYPTO\_OPERATIONS flag in the CK\_TOKEN\_INFO structure), an active signature1986operation would prevent Cryptoki from activating an encryption operation.
- CKR\_OPERATION\_NOT\_INITIALIZED: There is no active operation of an appropriate type in the specified session. For example, an application cannot call C\_Encrypt in a session without having called C\_EncryptInit first to activate an encryption operation.
- CKR\_PIN\_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried out unless C\_SetPIN is called to change the PIN value. Whether or not the normal user's PIN on a token ever expires varies from token to token.
- CKR\_PIN\_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the token. More generally-- when authentication to the token involves something other than a PIN-- the attempt to authenticate the user has failed.
- CKR\_PIN\_INVALID: The specified PIN has invalid characters in it. This return code only applies to functions which attempt to set a PIN.
- CKR\_PIN\_LEN\_RANGE: The specified PIN is too long or too short. This return code only applies to functions which attempt to set a PIN.
- CKR\_PIN\_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some particular number of failed authentication attempts has been reached, the token is unwilling to permit further attempts at authentication. Depending on the token, the specified PIN may or may not remain locked indefinitely.
- CKR\_PIN\_TOO\_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is too short, CKR\_PIN\_LEN\_RANGE should be returned instead. This return code only applies to functions which attempt to set a PIN.
- CKR\_PUBLIC\_KEY\_INVALID: The public key fails a public key validation. For example, an EC public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may be returned by C\_CreateObject, when the public key is created, or by C\_VerifyInit or C\_VerifyRecoverInit, when the public key is used. It may also be returned by C\_DeriveKey, in preference to CKR\_MECHANISM\_PARAM\_INVALID, if the other party's public key specified in the mechanism's parameters is invalid.
- CKR\_RANDOM\_NO\_RNG: This value can be returned by C\_SeedRandom and
   C\_GenerateRandom. It indicates that the specified token doesn't have a random number generator.
   This return value has higher priority than CKR\_RANDOM\_SEED\_NOT\_SUPPORTED.
- CKR\_RANDOM\_SEED\_NOT\_SUPPORTED: This value can only be returned by C\_SeedRandom.
   It indicates that the token's random number generator does not accept seeding from an application.
   This return value has lower priority than CKR\_RANDOM\_NO\_RNG.
- CKR\_SAVED\_STATE\_INVALID: This value can only be returned by **C\_SetOperationState**. It indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be restored to the specified session.

- CKR\_SESSION\_COUNT: This value can only be returned by **C\_OpenSession**. It indicates that the attempt to open a session failed, either because the token has too many sessions already open, or because the token has too many read/write sessions already open.
- CKR\_SESSION\_EXISTS: This value can only be returned by **C\_InitToken**. It indicates that a session with the token is already open, and so the token cannot be initialized.
- CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED: The specified token does not support parallel sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel sessions. CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED can only be returned by
   C\_OpenSession, and it is only returned when C\_OpenSession is called in a particular [deprecated] way.
- CKR\_SESSION\_READ\_ONLY: The specified session was unable to accomplish the desired action because it is a read-only session. This return value has lower priority than CKR\_TOKEN\_WRITE\_PROTECTED.
- CKR\_SESSION\_READ\_ONLY\_EXISTS: A read-only session already exists, and so the SO cannot be logged in.
- CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS: A read/write SO session already exists, and so a read-only session cannot be opened.
- CKR\_SIGNATURE\_LEN\_RANGE: The provided signature/MAC can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR\_SIGNATURE\_INVALID.
- CKR\_SIGNATURE\_INVALID: The provided signature/MAC is invalid. This return value has lower priority than CKR\_SIGNATURE\_LEN\_RANGE.
- CKR\_SLOT\_ID\_INVALID: The specified slot ID is not valid.
- CKR\_STATE\_UNSAVEABLE: The cryptographic operations state of the specified session cannot be saved for some reason (possibly the token is simply unable to save the current state). This return value has lower priority than CKR\_OPERATION\_NOT\_INITIALIZED.
- CKR\_TEMPLATE\_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes. See Section 4.1 for more information.
- CKR\_TEMPLATE\_INCONSISTENT: The template specified for creating an object has conflicting attributes. See Section 4.1 for more information.
- CKR\_TOKEN\_NOT\_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in the slot.
- CKR\_TOKEN\_WRITE\_PROTECTED: The requested action could not be performed because the token is write-protected. This return value has higher priority than CKR\_SESSION\_READ\_ONLY.
- CKR\_UNWRAPPING\_KEY\_HANDLE\_INVALID: This value can only be returned by **C\_UnwrapKey**. 2056 It indicates that the key handle specified to be used to unwrap another key is not valid.
- CKR\_UNWRAPPING\_KEY\_SIZE\_RANGE: This value can only be returned by C\_UnwrapKey. It indicates that although the requested unwrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the range of key sizes that it can handle.
- CKR\_UNWRAPPING\_KEY\_TYPE\_INCONSISTENT: This value can only be returned by
   C\_UnwrapKey. It indicates that the type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.
- CKR\_USER\_ALREADY\_LOGGED\_IN: This value can only be returned by C\_Login. It indicates that the specified user cannot be logged into the session, because it is already logged into the session.
   For example, if an application has an open SO session, and it attempts to log the SO into it, it will receive this error code.
- CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN: This value can only be returned by **C\_Login**. It indicates that the specified user cannot be logged into the session, because another user is already
- logged into the session. For example, if an application has an open SO session, and it attempts tolog the normal user into it, it will receive this error code.
- CKR\_USER\_NOT\_LOGGED\_IN: The desired action cannot be performed because the appropriate user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out unless it is logged in. Another example is that a private object cannot be created on a token unless the session attempting to create it is logged in as the normal user. A final example is that cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- CKR\_USER\_PIN\_NOT\_INITIALIZED: This value can only be returned by **C\_Login**. It indicates that the normal user's PIN has not yet been initialized with **C\_InitPIN**.
- CKR\_USER\_TOO\_MANY\_TYPES: An attempt was made to have more distinct users simultaneously logged into the token than the token and/or library permits. For example, if some application has an open SO session, and another application attempts to log the normal user into a session, the attempt may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be supported does C\_Login have to return this value. Note that this error code generalizes to true multi-user tokens.
- CKR\_USER\_TYPE\_INVALID: An invalid value was specified as a CK\_USER\_TYPE. Valid types are
   CKU\_SO, CKU\_USER, and CKU\_CONTEXT\_SPECIFIC.
- CKR\_WRAPPED\_KEY\_INVALID: This value can only be returned by C\_UnwrapKey. It indicates that the provided wrapped key is not valid. If a call is made to C\_UnwrapKey to unwrap a particular type of key (*i.e.*, some particular key type is specified in the template provided to C\_UnwrapKey), and the wrapped key provided to C\_UnwrapKey is recognizably not a wrapped key of the proper type, then C\_UnwrapKey should return CKR\_WRAPPED\_KEY\_INVALID. This return value has lower priority than CKR\_WRAPPED\_KEY\_LEN\_RANGE.
- CKR\_WRAPPED\_KEY\_LEN\_RANGE: This value can only be returned by C\_UnwrapKey. It indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR\_WRAPPED\_KEY\_INVALID.
- CKR\_WRAPPING\_KEY\_HANDLE\_INVALID: This value can only be returned by **C\_WrapKey**. It indicates that the key handle specified to be used to wrap another key is not valid.
- CKR\_WRAPPING\_KEY\_SIZE\_RANGE: This value can only be returned by C\_WrapKey. It indicates that although the requested wrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range of key sizes that it can handle.
- CKR\_WRAPPING\_KEY\_TYPE\_INCONSISTENT: This value can only be returned by **C\_WrapKey**. It indicates that the type of the key specified to wrap another key is not consistent with the mechanism specified for wrapping.
- 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**

- In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than CKR\_OK) take
   precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
   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
- these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
- applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
- 2116 descriptions of functions.

#### 2117 **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:
- 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.
- 2123
   2. As mentioned in Section 5.1.2, an application should never count on getting a
   2124
   CKR\_SESSION\_CLOSED error.
- 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.
- 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:
- 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.
- 21. 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.

2157 Cryptographic functions which return output in a variable-length buffer should always return as much 2158 output as can be computed from what has been passed in to them thus far. As an example, consider a 2159 session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode 2160 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the C DecryptUpdate 2161 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 2162 least 8 bytes. Hence the call to C DecryptUpdate should return 0 bytes of plaintext. If a single 2163 additional byte of ciphertext is supplied by a subsequent call to C DecryptUpdate, then that call should 2164 2165 return 8 bytes of plaintext (one full DES block).

# 2166 **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.

# 2171 **5.4 General-purpose functions**

2172 Cryptoki provides the following general-purpose functions:

# 2173 **5.4.1 C\_Initialize**

2174	CK_DECLARE_FUNCTION(CK_RV, C_Initialize) {
2175	CK_VOID_PTR pInitArgs
2176	);
2177 2178 2179 2180 2181	<b>C_Initialize</b> initializes the Cryptoki library. <i>plnitArgs</i> either has the value NULL_PTR or points to a <b>CK_C_INITIALIZE_ARGS</b> 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 <b>C_Initialize</b> (the consequences of supplying this value will be explained below).
2182 2183 2184 2185 2186	If <i>plnitArgs</i> is non-NULL_PTR, <b>C_Initialize</b> should cast it to a <b>CK_C_INITIALIZE_ARGS_PTR</b> and then dereference the resulting pointer to obtain the <b>CK_C_INITIALIZE_ARGS</b> fields <i>CreateMutex</i> , <i>DestroyMutex</i> , <i>LockMutex</i> , <i>UnlockMutex</i> , <i>flags</i> , and <i>pReserved</i> . For this version of Cryptoki, the value of <i>pReserved</i> thereby obtained MUST be NULL_PTR; if it's not, then <b>C_Initialize</b> should return with the value CKR_ARGUMENTS_BAD.
2187 2188 2189 2190 2191	If the <b>CKF_LIBRARY_CANT_CREATE_OS_THREADS</b> flag in the <i>flags</i> 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, <b>C_Initialize</b> should return with the value CKR_NEED_TO_CREATE_THREADS.
2192 2193 2194	A call to <b>C_Initialize</b> specifies one of four different ways to support multi-threaded access via the value of the <b>CKF_OS_LOCKING_OK</b> flag in the <i>flags</i> field and the values of the <i>CreateMutex</i> , <i>DestroyMutex</i> , <i>LockMutex</i> , and <i>UnlockMutex</i> function pointer fields:
2195 2196 2197	<ol> <li>If the flag <i>isn't</i> set, and the function pointer fields <i>aren't</i> supplied (<i>i.e.</i>, they all have the value NULL_PTR), that means that the application <i>won't</i> be accessing the Cryptoki library from multiple threads simultaneously.</li> </ol>
2198 2199 2200 2201	2. If the flag <i>is</i> set, and the function pointer fields <i>aren't</i> supplied ( <i>i.e.</i> , they all have the value NULL_PTR), that means that the application <i>will</i> 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, <b>C_Initialize</b> should return with the value CKR_CANT_LOCK.
2202 2203 2204 2205 2206	3. If the flag <i>isn't</i> set, and the function pointer fields <i>are</i> supplied ( <i>i.e.</i> , they all have non-NULL_PTR values), that means that the application <i>will</i> 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, <b>C_Initialize</b> should return with the value CKR_CANT_LOCK.
2207 2208 2209 2210 2211	4. If the flag <i>is</i> set, and the function pointer fields <i>are</i> supplied ( <i>i.e.</i> , they all have non-NULL_PTR values), that means that the application <i>will</i> 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, <b>C_Initialize</b> should return with the value CKR_CANT_LOCK.
2212 2213	If some, but not all, of the supplied function pointers to <b>C_Initialize</b> are non-NULL_PTR, then <b>C_Initialize</b> should return with the value CKR_ARGUMENTS_BAD.

- A call to **C\_Initialize** with *pInitArgs* set to NULL\_PTR is treated like a call to **C\_Initialize** with *pInitArgs*
- 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
- implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or any other resources it requires.
- If several applications are using Cryptoki, each one should call **C\_Initialize**. Every call to **C\_Initialize** should (eventually) be succeeded by a single call to **C\_Finalize**. See **[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 2229

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Finalize)( CK\_VOID\_PTR pReserved

2230

);

C\_Finalize is called to indicate that an application is finished with the Cryptoki library. It should be the
 last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for
 this version, it should be set to NULL\_PTR (if C\_Finalize is called with a non-NULL\_PTR value for
 *pReserved*, it should return the value CKR\_ARGUMENTS\_BAD.

- If several applications are using Cryptoki, each one should call C\_Finalize. Each application's call to
   C\_Finalize should be preceded by a single call to C\_Initialize; in between the two calls, an application can make calls to other Cryptoki functions. See [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 information. 2253 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);

#### 5.4.4 C\_GetFunctionList 2281

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.
        Whether or not this is the case, no attempt should be made to write to this memory.
2289
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
        to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.
2292
        Return values: CKR ARGUMENTS BAD, CKR FUNCTION FAILED, CKR GENERAL ERROR,
2293
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 2313	<b>C_GetInterfaceList</b> is used to obtain a list of interfaces supported by a Cryptoki library. <i>pulCount</i> points to the location that receives the number of interfaces.
2314	There are two ways for an application to call <b>C_GetInterfaceList</b> :
2315 2316 2317	<ol> <li>If pInterfaceList is NULL_PTR, then all that C_GetInterfaceList does is return (in *pulCount) the number of interfaces, without actually returning a list of interfaces. The contents of *pulCount on entry to C_GetInterfaceList has no meaning in this case, and the call returns the value CKR_OK.</li> </ol>
2318 2319 2320 2321 2322	<ol> <li>If <i>pIntrerfaceList</i> is not NULL_PTR, then *<i>pulCount</i> MUST contain the size (in terms of CK_INTERFACE elements) of the buffer pointed to by <i>pInterfaceList</i>. If that buffer is large enough to hold the list of interfaces, then the list is returned in it, and CKR_OK is returned. If not, then the call to C_GetInterfaceList returns the value CKR_BUFFER_TOO_SMALL. In either case, the value *<i>pulCount</i> is set to hold the number of interfaces.</li> </ol>
2323 2324	Because <b>C_GetInterfaceList</b> does not allocate any space of its own, an application will often call <b>C_GetInterfaceList</b> twice. However, this behavior is by no means required.
2325 2326 2327 2328	<b>C_GetInterfaceList</b> obtains (in * <i>pFunctionList</i> of each interface) a pointer to the Cryptoki library's list of function pointers. <i>The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable</i> . Whether or not this is the case, no attempt should be made to write to this memory. The same caveat applies to the interface names returned.
2329 2330 2321	C_GetFunctionList, C_GetInterfaceList, and C_GetInterface are the only Cryptoki functions which an 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.
2332 2333 2333 2334	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.
2332 2333 2334 2335	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example:
2331 2332 2333 2334 2335 2336	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG_ulCount=0;
2331 2332 2333 2334 2335 2336 2337	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL;
2332 2333 2334 2335 2336 2337 2338	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv;
2331 2332 2333 2334 2335 2336 2337 2338 2339	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I;
2331 2332 2333 2334 2335 2336 2337 2338 2339 2340	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I;
2332 2333 2334 2335 2336 2337 2338 2339 2340 2341	to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */
2332 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342	<pre>to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */ rv = C_GetInterfaceList(NULL, &amp;ulCount);</pre>
2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343	<pre>to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */ rv = C_GetInterfaceList(NULL,&amp;ulCount); if (rv == CKR_OK) {</pre>
2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344	<pre>to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */ rv = C_GetInterfaceList(NULL,&amp;ulCount); if (rv == CKR_OK) {     /* get copy of interfaces */</pre>
2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345	<pre>to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */ rv = C_GetInterfaceList(NULL,&amp;ulCount); if (rv == CKR_OK) { /* get copy of interfaces */ interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));</pre>
2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346	<pre>to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously. Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK. Example: CK_ULONG ulCount=0; CK_INTERFACE_PTR interfaceList=NULL; CK_RV rv; int I; /* get number of interfaces */ rv = C_GetInterfaceList(NULL,&amp;ulCount); if (rv == CKR_OK) { /* get copy of interfaces */ interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE)); rv = C_GetInterfaceList(interfaceList,&amp;ulCount);</pre>

```
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
       }
```

#### 2357 5.4.6 C\_GetInterface

2356

```
2358
       CK DECLARE FUNCTION (CK RV,C GetInterface) (
2359
          CK UTF8CHAR PTR
                                   pInterfaceName,
2360
                                   pVersion,
          CK VERSION PTR
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
```

**C\_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that 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**:

- 23681. If pInterfaceName is not NULL\_PTR, the name of the interface returned must match. If2369pInterfaceName 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 NULL\_PTR, the cryptoki library can return an interface of any version
- If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

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

C\_GetFunctionList, C\_GetInterfaceList, and C\_GetInterface are the only Cryptoki functions which an
 application may call before calling C\_Initialize. It is provided to make it easier and faster for applications
 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

```
2381Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,2382CKR_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 */
       rv = C GetInterface((CK UTF8CHAR PTR)"PKCS 11",NULL,&interface,flags);
2401
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
```

#### 2435 5.5 Slot and token management functions

2436 Cryptoki provides the following functions for slot and token management:

#### 2437 **5.5.1 C\_GetSlotList**

```
2438
        CK DECLARE FUNCTION (CK RV, C GetSlotList) (
2439
                CK BBOOL tokenPresent,
2440
                CK SLOT ID PTR pSlotList,
2441
                CK ULONG PTR pulCount
2442
        );
2443
        C GetSlotList is used to obtain a list of slots in the system. tokenPresent indicates whether the list
2444
        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.
2445
2446
        There are two ways for an application to call C GetSlotList:
2447
        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
2448
2449
            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
2450
            elements) of the buffer pointed to by pSlotList. If that buffer is large enough to hold the list of slots,
2451
            then the list is returned in it, and CKR OK is returned. If not, then the call to C_GetSlotList returns
2452
2453
            the value CKR BUFFER TOO SMALL. In either case, the value *pulCount is set to hold the number
            of slots.
2454
2455
        Because C_GetSlotList does not allocate any space of its own, an application will often call
2456
        C GetSlotList twice (or sometimes even more times—if an application is trying to get a list of all slots
2457
        with a token present, then the number of such slots can (unfortunately) change between when the
2458
        application asks for how many such slots there are and when the application asks for the slots
2459
        themselves). However, multiple calls to C_GetSlotList are by no means required.
2460
        All slots which C GetSlotList reports MUST be able to be queried as valid slots by C GetSlotInfo.
2461
        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
2462
        C_GetSlotList with a non-NULL pSlotList, and then the user adds or removes a hardware device, the
2463
        changed slot list will only be visible and effective if C GetSlotList is called again with NULL. Even if C
2464
        GetSlotList is successfully called this way, it may or may not be the case that the changed slot list will be
2465
        successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11
2466
        compliant libraries, it may be necessary to successfully call C Initialize or to restart the entire system.
2467
2468
2469
        Return values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,
2470
        CKR CRYPTOKI NOT INITIALIZED CKR FUNCTION FAILED CKR GENERAL ERROR.
2471
        CKR HOST MEMORY, CKR OK.
2472
        Example:
2473
        CK ULONG ulSlotCount, ulSlotWithTokenCount;
2474
        CK SLOT ID PTR pSlotList, pSlotWithTokenList;
2475
        CK RV rv;
2476
2477
        /* Get list of all slots */
2478
        rv = C GetSlotList(CK FALSE, NULL PTR, &ulSlotCount);
2479
        if (rv == CKR OK) {
2480
           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; *plnfo* 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.** 

#### 2523 **5.5.3 C\_GetTokenInfo**

```
2524
       CK DECLARE FUNCTION(CK RV, C GetTokenInfo)(
2525
             CK SLOT ID slotID,
2526
             CK TOKEN INFO PTR pInfo
2527
       );
2528
       C GetTokenInfo obtains information about a particular token in the system. slotID is the ID of the
       token's slot; plnfo points to the location that receives the token information.
2529
       Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2530
       CKR DEVICE REMOVED, CKR FUNCTION FAILED, CKR GENERAL ERROR,
2531
2532
       CKR HOST MEMORY, CKR OK, CKR SLOT ID INVALID, CKR TOKEN NOT PRESENT,
2533
       CKR TOKEN NOT RECOGNIZED, CKR ARGUMENTS BAD.
2534
       Example:
2535
       CK ULONG ulCount;
2536
       CK SLOT ID PTR pSlotList;
2537
       CK SLOT INFO slotInfo;
2538
       CK TOKEN INFO tokenInfo;
2539
       CK RV rv;
2540
2541
       rv = C GetSlotList(CK FALSE, NULL PTR, &ulCount);
2542
       if ((rv == CKR OK) && (ulCount > 0)) {
2543
         pSlotList = (CK SLOT ID PTR) malloc(ulCount*sizeof(CK SLOT ID));
2544
         rv = C GetSlotList(CK FALSE, pSlotList, &ulCount);
2545
         assert(rv == CKR OK);
2546
2547
         /* Get slot information for first slot */
2548
         rv = C GetSlotInfo(pSlotList[0], &slotInfo);
2549
         assert(rv == CKR OK);
2550
2551
         /* Get token information for first slot */
2552
         rv = C GetTokenInfo(pSlotList[0], &tokenInfo);
2553
         if (rv == CKR TOKEN NOT PRESENT) {
2554
2555
2556
         }
2557
2558
2559
         free(pSlotList);
2560
```

#### 2561 5.5.4 C\_WaitForSlotEvent

2562 2563 2564 CK\_DECLARE\_FUNCTION(CK\_RV, C\_WaitForSlotEvent)( CK\_FLAGS flags, CK SLOT ID PTR pSlot,

2565	CK_VOID_PTR pReserved
2566	);
2567 2568 2569 2570	<b>C_WaitForSlotEvent</b> waits for a slot event, such as token insertion or token removal, to occur. <i>flags</i> determines whether or not the <b>C_WaitForSlotEvent</b> call blocks ( <i>i.e.</i> , waits for a slot event to occur); <i>pSlot</i> points to a location which will receive the ID of the slot that the event occurred in. <i>pReserved</i> is reserved for future versions; for this version of Cryptoki, it should be NULL_PTR.
2571	At present, the only flag defined for use in the <i>flags</i> argument is <b>CKF_DONT_BLOCK</b> :
2572 2573 2574 2575	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 <b>C_Initialize</b> , 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.
2576 2577 2578 2579	If <b>C_WaitForSlotEvent</b> is called with the <b>CKF_DONT_BLOCK</b> flag set in the <i>flags</i> 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 <i>pSlot</i> . 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.
2580 2581 2582	If <b>C_WaitForSlotEvent</b> is called with the <b>CKF_DONT_BLOCK</b> flag set in the <i>flags</i> 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 <i>pSlot</i> when <b>C_WaitForSlotEvent</b> are undefined.
2583 2584 2585 2586 2587	If <b>C_WaitForSlotEvent</b> is called with the <b>CKF_DONT_BLOCK</b> flag clear in the <i>flags</i> 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, <b>C_WaitForSlotEvent</b> will wait until some slot's event flag becomes set. If a thread of an application has a <b>C_WaitForSlotEvent</b> call blocking when another thread of that application calls <b>C_Finalize</b> , the <b>C_WaitForSlotEvent</b> call returns with the value CKR_CRYPTOKI_NOT_INITIALIZED.
2588 2589 2590	Although the parameters supplied to <b>C_Initialize</b> can in general allow for safe multi-threaded access to a Cryptoki library, <b>C_WaitForSlotEvent</b> is exceptional in that the behavior of Cryptoki is undefined if multiple threads of a single application make simultaneous calls to <b>C_WaitForSlotEvent</b> .
2591 2592 2593	Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_NO_EVENT, CKR_OK.
2594	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	<pre>rv = C_WaitForSlotEvent(flags, &amp;slotID, NULL_PTR);</pre>
2603	<pre>assert(rv == CKR_OK);</pre>
2604	
2605	/* See what's up with that slot */
2606	<pre>rv = C_GetSlotInfo(slotID, &amp;slotInfo);</pre>
2607	assert(rv == CKR_OK);
2608	

# 2609 5.5.5 C\_GetMechanismList

2610 CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetMechanismList)(

2611	CK SIOT ID SIGTID
2612	CK_SIGI_ID_SIGCID,
2012	CK_MECHANISM_IIIE_FIK PMechanismusse,
2013	).
2014	
2615 2616	of the token's slot; <i>pulCount</i> points to the location that receives the number of mechanisms.
2617	There are two ways for an application to call <b>C_GetMechanismList</b> :
2618 2619 2620 2621	<ol> <li>If <i>pMechanismList</i> is NULL_PTR, then all that C_GetMechanismList does is return (in *<i>pulCount</i>) the number of mechanisms, without actually returning a list of mechanisms. The contents of *<i>pulCount</i> on entry to C_GetMechanismList has no meaning in this case, and the call returns the value CKR_OK.</li> </ol>
2622 2623 2624 2625 2626	2. If <i>pMechanismList</i> is not NULL_PTR, then * <i>pulCount</i> MUST contain the size (in terms of CK_MECHANISM_TYPE elements) of the buffer pointed to by <i>pMechanismList</i> . If that buffer is large enough to hold the list of mechanisms, then the list is returned in it, and CKR_OK is returned. If not, then the call to C_GetMechanismList returns the value CKR_BUFFER_TOO_SMALL. In either case, the value * <i>pulCount</i> is set to hold the number of mechanisms.
2627 2628	Because <b>C_GetMechanismList</b> does not allocate any space of its own, an application will often call <b>C_GetMechanismList</b> twice. However, this behavior is by no means required.
2629 2630 2631 2632 2633	Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.
2634	Example:
2634 2635	Example: CK SLOT ID slotID;
2634 2635 2636	Example: CK_SLOT_ID slotID; CK_ULONG ulCount;
2634 2635 2636 2637	Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList;
2634 2635 2636 2637 2638	Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv;
2634 2635 2636 2637 2638 2639	Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv;
2634 2635 2636 2637 2638 2639 2640	Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv;
2634 2635 2636 2637 2638 2639 2640 2641	Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv;
2634 2635 2636 2637 2638 2639 2640 2641 2642	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount);</pre>
2634 2635 2636 2637 2638 2639 2640 2641 2642 2643	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) {</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2643 2644	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList =</pre>
2634 2635 2636 2637 2638 2639 2640 2641 2642 2643 2644 2644 2645	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR)</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2643 2644 2645 2646	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE));</pre>
2634 2635 2636 2637 2638 2639 2640 2641 2642 2643 2644 2645 2646 2647	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount);</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2643 2644 2645 2645 2646 2647 2648	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount); if (rv == CKR_OK) {</pre>
2634 2635 2636 2637 2638 2639 2640 2641 2642 2643 2644 2645 2646 2647 2648 2649	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount); if (rv == CKR_OK) { .</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2644 2645 2644 2645 2646 2647 2648 2649 2650	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount); if (rv == CKR_OK) { .</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2644 2645 2646 2647 2648 2649 2650 2651	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount); if (rv == CKR_OK) { .</pre>
2634 2635 2637 2638 2639 2640 2641 2642 2643 2644 2645 2644 2645 2646 2647 2648 2649 2650 2651 2651	<pre>Example: CK_SLOT_ID slotID; CK_ULONG ulCount; CK_MECHANISM_TYPE_PTR pMechanismList; CK_RV rv; rv = C_GetMechanismList(slotID, NULL_PTR, &amp;ulCount); if ((rv == CKR_OK) &amp;&amp; (ulCount &gt; 0)) { pMechanismList = (CK_MECHANISM_TYPE_PTR) malloc(ulCount*sizeof(CK_MECHANISM_TYPE)); rv = C_GetMechanismList(slotID, pMechanismList, &amp;ulCount); if (rv == CKR_OK) { } free(pMechanismList); </pre>

# 2654 5.5.6 C\_GetMechanismInfo

2655

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetMechanismInfo)(

15 June 2020 Page 85 of 167

2656	CK_SLOT_ID slotID,
2657	CK_MECHANISM_TYPE type,
2658	CK_MECHANISM_INFO_PTR pInfo
2659	);
2660 2661 2662	<b>C_GetMechanismInfo</b> obtains information about a particular mechanism possibly supported by a token. <i>slotID</i> is the ID of the token's slot; <i>type</i> is the type of mechanism; <i>pInfo</i> points to the location that receives the mechanism information.
2663 2664 2665 2666	Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_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	<pre>rv = C_GetMechanismInfo(slotID, CKM_MD2, &amp;info);</pre>
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 );

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

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

- 2704 and all bules of some objects may be set to default values.
- 2705 If the token has a "protected authentication path", as indicated by the

CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its CK\_TOKEN\_INFO being set, then that means
 that there is some way for a user to be authenticated to the token without having the application send a
 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin* parameter to C\_InitToken should be NULL\_PTR. During the execution of C\_InitToken, the SO's PIN will
 be entered through the protected authentication path.

- 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.
- A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
- call to C\_InitToken is made under such circumstances, the call fails with error CKR\_SESSION\_EXISTS.
   Unfortunately, it may happen when C\_InitToken is called that some other application *does* have an open
- 2716 Onortunately, it may happen when **C\_init loken** is called that some other application *does* have an ope 2717 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
- applications using the token. If this is the case, then the consequences of the **C\_InitToken** call are undefined.
- The **C\_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations, an initialization mechanism outside the scope of Cryptoki MUST be employed. The definition of "complex
- token" is product specific.
- 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	<pre>CK_UTF8CHAR pin[] = { "MyPIN" };</pre>
2732	CK_UTF8CHAR label[32];
2733	CK_RV rv;
2734	
2735	
2736	
2737	<pre>memset(label, ` ', sizeof(label));</pre>
2738	<pre>memcpy(label, "My first token", strlen("My first token"));</pre>
2739	<pre>rv = C_InitToken(slotID, pin, strlen(pin), label);</pre>
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 2749	CK_ULONG ulPinLen );
2750 2751 2752	<b>C_InitPIN</b> initializes the normal user's PIN. <i>hSession</i> is the session's handle; <i>pPin</i> points to the normal user's PIN; <i>ulPinLen</i> 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.
2753 2754	<b>C_InitPIN</b> 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.
2755 2756 2757 2758 2759 2760 2760	If the token has a "protected authentication path", as indicated by the CKF_PROTECTED_AUTHENTICATION_PATH flag in its <b>CK_TOKEN_INFO</b> 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 <i>pPin</i> parameter to <b>C_InitPIN</b> should be NULL_PTR. During the execution of <b>C_InitPIN</b> , the SO will enter the new PIN through the protected authentication path.
2762 2763	If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether or not <b>C_InitPIN</b> can be used to initialize the normal user's token access.
2764 2765 2766 2767 2768 2769	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.
2770	Example:
2771	CK_SESSION_HANDLE hSession;
2772	<pre>CK_UTF8CHAR newPin[] = {"NewPIN"};</pre>
2773	CK_RV rv;
2774	
2775	<pre>rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);</pre>
2776	if (rv == CKR_OK) {
2777	
2778	

#### 2780 5.5.9 C SetPIN

}

2779

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.

2793 C\_SetPIN can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User
 2794 Functions" state. An attempt to call it from a session in any other state fails with error

2795 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
- 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 modify the current user's PIN on a token with such a protected authentication path,
- the *pOldPin* and *pNewPin* parameters to **C\_SetPIN** should be NULL\_PTR. During the execution of
- 2802 **C\_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
- path. It is not specified how the 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**

- A typical application might perform the following series of steps to make use of a token (note that there are other reasonable sequences of events that an application might perform):
- 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 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.
- As has been observed, an application may have concurrent sessions with more than one token. It is also possible for a token to have concurrent sessions with more than one application.
- 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

C\_OpenSession opens a session between an application and a token in a particular slot. *slotID* is the
 slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to
 the notification callback; *Notify* is the address of the notification callback function (see Section 5.21);
 *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.
- There may be a limit on the number of concurrent sessions an application may have with the token, which may depend on whether the session is "read-only" or "read/write". An attempt to open a session which does not succeed because there are too many existing sessions of some type should return CKR SESSION COUNT.
- 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
- application does not wish to support callbacks, it should pass a value of NULL\_PTR as the *Notify* 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
- 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, CKR DEVICE REMOVED, CKR FUNCTION FAILED, CKR GENERAL ERROR, 2893 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 2917	CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions)( CK SLOT ID slotID
2918	);
2919	C_CloseAllSessions closes all sessions an application has with a token. <i>slotID</i> specifies the token's slot.
2920	When a session is closed, all session objects created by the session are destroyed automatically.
2921 2922 2923	After successful execution of this function, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.
2924 2925	Depending on the token, when the last open session any application has with the token is closed, the token may be "ejected" from its reader (if this capability exists).
2926 2927 2928	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.
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,
         CK_SESSION INFO PTR pInfo
2939
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,
       CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
2945
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* indicates the operations to cancel.

- To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical 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.
- If a flag is set for an operation that has not been initialized in the session, the operation flag will be 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

If C\_SessionCancel is called from an application callback (see Section 5.16), no action will be taken by
 the library and CKR\_FUNCTION\_FAILED must be returned.
 If C\_SessionCancel is used to cancel one half of a dual-function operation, the remaining operation

should still be left in an active state. However, it is expected that some Cryptoki implementations may not
 support this and return CKR\_OPERATION\_CANCEL\_FAILED unless flags for both operations are
 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
```

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

#### 3020 **5.6.6 C\_GetOperationState**

3017

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

3024 3025	CK_ULONG_PTR pulOperationStateLen );
3026 3027 3028	<b>C_GetOperationState</b> obtains a copy of the cryptographic operations state of a session, encoded as a string of bytes. <i>hSession</i> is the session's handle; <i>pOperationState</i> points to the location that receives the state; <i>pulOperationStateLen</i> points to the location that receives the length in bytes of the state.
3029 3030 3031	Although the saved state output by <b>C_GetOperationState</b> is not really produced by a "cryptographic mechanism", <b>C_GetOperationState</b> nonetheless uses the convention described in Section 5.2 on producing output.
3032 3033 3034	Precisely what the "cryptographic operations state" this function saves is varies from token to token; however, this state is what is provided as input to <b>C_SetOperationState</b> to restore the cryptographic activities of a session.
3035 3036 3037 3038 3039 3040 3041 3042 3043 3043 3044	Consider a session which is performing a message digest operation using SHA-1 ( <i>i.e.</i> , the session is using the <b>CKM_SHA_1</b> mechanism). Suppose that the message digest operation was initialized properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The application now wants to "save the state" of this digest operation, so that it can continue it later. In this particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken together, these three pieces of information suffice to continue the current hashing operation at a later time.
3045 3046 3047 3048 3049 3050 3051 3052 3053 3054	Consider next a session which is performing an encryption operation with DES (a block cipher with a block size of 64 bits) in CBC (cipher-block chaining) mode ( <i>i.e.</i> , the session is using the <b>CKM_DES_CBC</b> mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already been produced and output. In this case, the cryptographic operations state of the session most likely consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipherblock chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some administrative data indicating that this saved state comes from a session which was performing DES encryption in CBC mode; and possibly the DES key being used for encryption (see <b>C_SetOperationState</b> for more information on whether or not the key is present in the saved state).
3055 3056 3057	If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the cryptographic operations state of the session will contain all the necessary information to restore both operations.
3058 3059 3060 3061	An attempt to save the cryptographic operations state of a session which does not currently have some active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message recovery, verification without message recovery, or some legal combination of two of these) should fail with the error CKR_OPERATION_NOT_INITIALIZED.
3062 3063 3064 3065	An attempt to save the cryptographic operations state of a session which is performing an appropriate cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain necessary state information and/or key information can't leave the token, for example) should fail with the error CKR_STATE_UNSAVEABLE.
3066 3067 3068 3069 3070	Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_STATE_UNSAVEABLE, CKR_ARGUMENTS_BAD.
3071	Example: see <b>C_SetOperationState</b> .
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 3080 obtained with C\_GetOperationState. hSession is the session's handle; pOperationState points to the 3081 3082 location holding the saved state; *ulOperationStateLen* holds the length of the saved state; 3083 hEncryptionKey holds a handle to the key which will be used for an ongoing encryption or decryption 3084 operation in the restored session (or 0 if no encryption or decryption key is needed, either because no 3085 such operation is ongoing in the stored session or because all the necessary key information is present in 3086 the saved state); hAuthenticationKey holds a handle to the key which will be used for an ongoing 3087 signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either 3088 because no such operation is ongoing in the stored session or because all the necessary key information 3089 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
- 3103 **C\_SetOperationState** will fail and return the error CKR\_KEY\_NEEDED. If the key in use for the 3104 operation *is* saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not 3105 required.
- 3106 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
- 3108 **C\_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C\_SetOperationState** will 3109 fail with the error CKR\_KEY\_NEEDED. If the key in use for the operation *is* saved in the state, then it *can* 3110 be supplied in the *hAuthenticationKey* argument, but this is not required.
- 3111 If an *irrelevant* key is supplied to **C\_SetOperationState** call (*e.g.*, a nonzero key handle is submitted in the *b*-per matient key argument, but the several emptagraphic expertises extended and the several emptagraphic extended and the several emptagr
- 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
- 3113 ongoing encryption or decryption operation, then C\_SetOperationS
   3114 CKR KEY NOT NEEDED.
- SII4 CKR\_KET\_NOT\_NEEDED.
  - 3115 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,
- 3118 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.
- 3125 **C\_SetOperationState** can successfully restore cryptographic operations state to a session even if that 3126 session has active cryptographic or object search operations when **C\_SetOperationState** is called (the 3127 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,
       CKR OK, CKR SAVED STATE INVALID, CKR SESSION CLOSED,
3131
3132
       CKR SESSION HANDLE INVALID, CKR ARGUMENTS BAD.
3133
       Example:
3134
       CK SESSION HANDLE hSession;
3135
       CK MECHANISM digestMechanism;
3136
       CK BYTE PTR pState;
       CK ULONG ulStateLen;
3137
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 */
       pkcs11-base-v3.0-os
                                                                                15 June 2020
       Standards Track Work Product
```

Copyright © OASIS Open 2020. All Rights Reserved.

Page 96 of 167

```
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);
       if (rv == CKR OK) {
3178
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 );
```

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

When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of C\_Login depends on the context in which it is called. Improper use of this user type will result in a return value

- 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 on the slot device. Or the user might not even use a PIN-authentication could be achieved by some 3202 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.

If there are any active cryptographic or object finding operations in an application's session, and then
 C\_Login is successfully executed by that application, it may or may not be the case that those operations

- 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.

When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of **C\_LoginUser** depends 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 method supported by the token will have been performed; a return value of CKR OK means that the user 3251 3252 was successfully authenticated, and a return value of CKR PIN INCORRECT means that the user was 3253 denied access.

- If there are any active cryptographic or object finding operations in an application's session, and then 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
```

#### 5.6.10 C Logout 3288

```
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 either the "R/W Public Session" state or the "R/O Public Session" state. 3294

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 session objects from sessions belonging to the application are destroyed. 3297

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 operations are still active. Therefore, before logging out, any active operations should be finished.

3300

```
3301
      Return values: CKR CRYPTOKI NOT INITIALIZED, CKR DEVICE ERROR, CKR DEVICE MEMORY,
      CKR DEVICE REMOVED, CKR FUNCTION FAILED, CKR GENERAL ERROR,
3302
```

```
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 );
```

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

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

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.
- Whenever an object is created, a value for CKA\_UNIQUE\_ID is generated and assigned to the new 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	<pre>keyClass = CKO_PUBLIC_KEY;</pre>
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 CKA NEVER EXTRACTABLE attribute will have the value CK FALSE. Similarly, the template may 3430 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 the CKA TOKEN and CKA PRIVATE attributes (e.g., to copy a session object to a token object). If the 3433 3434 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. 3435

- If a call to C\_CopyObject cannot support the precise template supplied to it, it will fail and return without
   creating any object. If the object indicated by hObject has its CKA\_COPYABLE attribute set to
   CK\_FALSE, C\_CopyObject will return CKR\_ACTION\_PROHIBITED.
- Whenever an object is copied, a new value for CKA\_UNIQUE\_ID is generated and assigned to the new object (See Section 4.4.1).
- Only session objects can be created during a read-only session. Only public objects can be createdunless 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
```

# 3482 5.7.3 C\_DestroyObject

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

# 3487 C\_DestroyObject destroys an object. *hSession* is the session's handle; and *hObject* is the object's 3488 handle.

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

- 3491 Certain objects may not be destroyed. Calling C\_DestroyObject on such objects will result in the
- 3492 CKR\_ACTION\_PROHIBITED error code. An application can consult the object's CKA\_DESTROYABLE 3493 attribute to determine if an object may be destroyed or not.
- 3494 Return values: CKR\_ACTION\_PROHIBITED, CKR\_CRYPTOKI\_NOT\_INITIALIZED,
- 3495 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,

- 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.
       Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure
3509
       of how much token memory the object takes up. If an application deletes (say) a private object of size S,
3510
       it might be reasonable to assume that the ulFreePrivateMemory field of the token's CK TOKEN INFO
3511
3512
       structure increases by approximately S.
3513
       Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
       CKR DEVICE ERROR, CKR DEVICE MEMORY, CKR DEVICE REMOVED,
3514
       CKR FUNCTION FAILED, CKR GENERAL ERROR, CKR HOST MEMORY,
3515
       CKR INFORMATION SENSITIVE, CKR OBJECT HANDLE INVALID, CKR OK,
3516
       CKR SESSION CLOSED, CKR SESSION HANDLE INVALID.
3517
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[] = \{\ldots\};
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	<pre>rv = C_DestroyObject(hSession, hObject);</pre>
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,
           CK ULONG ulCount
3553
3554
        );
3555
        C_GetAttributeValue obtains the value of one or more attributes of an object. hSession is the session's
        handle; hObject is the object's handle; pTemplate points to a template that specifies which attribute
3556
        values are to be obtained, and receives the attribute values; ulCount is the number of attributes in the
3557
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
             attribute), then the ulValueLen field in that triple is modified to hold the value
3565
            CK UNAVAILABLE INFORMATION.
3566
3567
        3. Otherwise, if the pValue field has the value NULL_PTR, then the uIValueLen field is modified to hold
            the exact length of the specified attribute for the object.
3568
3569
           Otherwise, if the length specified in ulValueLen is large enough to hold the value of the specified
        4.
3570
             attribute for the object, then that attribute is copied into the buffer located at pValue, and the
            ulValueLen field is modified to hold the exact length of the attribute.
3571
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
        return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes,
3575
        then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these
3576
        error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the
3577
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
        MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is
3581
        NULL PTR then the ulValueLen of elements within the array will be set to the required length. If the
3582
        pValue of elements within the array is not NULL PTR, then the ulValueLen element of attributes within
3583
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
        small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
3590
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
       value can be returned by the call to C GetAttributeValue will be returned by the call to
3596
       C GetAttributeValue.
3597
3598
       Return values: CKR ARGUMENTS BAD, CKR ATTRIBUTE SENSITIVE,
3599
       CKR ATTRIBUTE TYPE INVALID, CKR BUFFER TOO SMALL,
       CKR CRYPTOKI NOT INITIALIZED, CKR DEVICE ERROR, CKR DEVICE MEMORY,
3600
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.
       Example:
3604
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
```

## 3634 5.7.6 C\_SetAttributeValue

3635 3636	CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue)( CK_SESSION_HANDLE_bSession,			
3637	CK OBJECT HANDLE hObject,			
3638	CK_ATTRIBUTE_PTR pTemplate,			
3639	CK_ULONG ulCount			
3040	);			
3641 3642 3643	<b>C_SetAttributeValue</b> modifies the value of one or more attributes of an object. <i>hSession</i> is the session's handle; <i>hObject</i> is the object's handle; <i>pTemplate</i> points to a template that specifies which attribute values are to be modified and their new values; <i>ulCount</i> is the number of attributes in the template.			
3644 3645 3646	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.			
3647	Only session objects can be modified during a read-only session.			
3648 3649 3650	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.			
3651	Not all attributes can be modified; see Section 4.1.2 for more details.			
3652	Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,			
3653	CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,			
3654	CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,			
3656	CKR GENERAL ERROR, CKR HOST MEMORY, CKR OBJECT HANDLE INVALID, CKR OK,			
3657	CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,			
3658	CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,			
2029	CKR_USER_NUT_LUGGED_IN.			
3000				
3661	CK_SESSION_HANDLE hSession;			
3662	CK_OBJECT_HANDLE hObject;			
3663	<pre>CK_UTF8CHAR label[] = {"New label"};</pre>			
3664	CK_ATTRIBUTE template[] = {			
3665	{CKA_LABEL, label, sizeof(label)-1}			
3666	};			
3667	CK_RV rv;			
3668				
3669				
3670				
3671	<pre>rv = C_SetAttributeValue(hSession, hObject, template, 1);</pre>			
3672	if $(rv == CKR_OK)$ {			
3673				
0074				
3674				

# 3676 5.7.7 C\_FindObjectsInit

```
3677 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit)(
3678 CK_SESSION_HANDLE hSession,
3679 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; ulCount is the number of attributes in the search template. The matching criterion is an exact byte-for-3684 3685 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

- After calling C FindObjectsInit, the application may call C FindObjects one or more times to obtain 3686 3687 handles for objects matching the template, and then eventually call C FindObjectsFinal to finish the active search operation. At most one search operation may be active at a given time in a given session. 3688
- 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 active search operation, then those objects may or may not be found by the search operation. Note that 3693 this means that, under these circumstances, the search operation may return invalid object handles. 3694
- Even though C FindObiectsInit can return the values CKR ATTRIBUTE TYPE INVALID and 3695 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 search operation which will match no objects and return CKR OK. 3698
- 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,
- CKR DEVICE MEMORY, CKR DEVICE REMOVED, CKR FUNCTION FAILED, 3703
- 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.

#### 5.7.8 C FindObjects 3707

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 );

**C** FindObjects continues a search for token and session objects that match a template, obtaining 3714 additional object handles. hSession is the session's handle: phObject points to the location that receives 3715 the list (array) of additional object handles: ulMaxObjectCount is the maximum number of object handles 3716 to be returned; pulObjectCount points to the location that receives the actual number of object handles 3717 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,
- CKR FUNCTION FAILED, CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK, 3724
- 3725 CKR OPERATION NOT INITIALIZED, CKR SESSION CLOSED, CKR SESSION HANDLE INVALID.
- 3726 Example: see C FindObiectsFinal.
## 3727 5.7.9 C\_FindObjectsFinal

3728 3729 3730	<pre>CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal)(     CK_SESSION_HANDLE hSession );</pre>		
3731 3732	<b>C_FindObjectsFinal</b> terminates a search for token and session objects. <i>hSession</i> is the session's handle.		
3733 3734 3735 3736	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.		
3737	Example:		
3738	CK_SESSION_HANDLE hSession;		
3739	CK_OBJECT_HANDLE hObject;		
3740	CK_ULONG ulObjectCount;		
3741	CK_RV rv;		
3742			
3743			
3744			
3745	<pre>rv = C_FindObjectsInit(hSession, NULL_PTR, 0);</pre>		
3746	<pre>assert(rv == CKR_OK);</pre>		
3747	while (1) {		
3748	<pre>rv = C_FindObjects(hSession, &amp;hObject, 1, &amp;ulObjectCount);</pre>		
3749	if (rv != CKR_OK    ulObjectCount == 0)		
3750	break;		
3751			
3752			
3753	}		
3754			
3755	<pre>rv = C_FindObjectsFinal(hSession);</pre>		
3756	<pre>assert(rv == CKR_OK);</pre>		

# 3757 **5.8 Encryption functions**

3758 Cryptoki provides the following functions for encrypting data:

### 3759 **5.8.1 C\_EncryptInit**

```
3760 CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit)(
3761 CK_SESSION_HANDLE hSession,
3762 CK_MECHANISM_PTR pMechanism,
3763 CK_OBJECT_HANDLE hKey
3764 );
```

3765 C\_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points
 3766 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 3788 3789 3790 3791 3792 3793	<pre>CK_DECLARE_FUNCTION(CK_RV, C_Encrypt)( CK_SESSION_HANDLE hSession, CK_BYTE_PTR pData, CK_ULONG ulDataLen, CK_BYTE_PTR pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen );</pre>
3794 3795 3796 3797	<b>C_Encrypt</b> encrypts single-part data. <i>hSession</i> is the session's handle; <i>pData</i> points to the data; <i>ulDataLen</i> is the length in bytes of the data; <i>pEncryptedData</i> points to the location that receives the encrypted data; <i>pulEncryptedDataLen</i> points to the location that holds the length in bytes of the encrypted data.
3798	C_Encrypt uses the convention described in Section 5.2 on producing output.
3799 3800 3801 3802	The encryption operation MUST have been initialized with <b>C_EncryptInit</b> . A call to <b>C_Encrypt</b> always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call ( <i>i.e.</i> , one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.
3803 3804	<b>C_Encrypt</b> cannot be used to terminate a multi-part operation, and MUST be called after <b>C_EncryptInit</b> without intervening <b>C_EncryptUpdate</b> calls.
3805 3806 3807 3808	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 <b>C_Encrypt</b> will fail with return code CKR_DATA_LEN_RANGE.
3809 3810	The plaintext and ciphertext can be in the same place, <i>i.e.</i> , it is OK if <i>pData</i> and <i>pEncryptedData</i> point to the same location.
3811 3812	For most mechanisms, <b>C_Encrypt</b> is equivalent to a sequence of <b>C_EncryptUpdate</b> operations followed by <b>C_EncryptFinal</b> .
3813 3814 3815 3816 3817 3818	Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
3819	Example: see <b>C_EncryptFinal</b> for an example of similar functions.
	nkes11 base v2.0 os

#### 5.8.3 C EncryptUpdate 3820

3821 3822 3823 3824 3825 3826 3827	<pre>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 );</pre>
3828 3829 3830 3831	<b>C_EncryptUpdate</b> continues a multiple-part encryption operation, processing another data part. <i>hSession</i> is the session's handle; <i>pPart</i> points to the data part; <i>ulPartLen</i> is the length of the data part; <i>pEncryptedPart</i> points to the location that receives the encrypted data part; <i>pulEncryptedPartLen</i> points to the location that holds the length in bytes of the encrypted data part.
3832	C_EncryptUpdate uses the convention described in Section 5.2 on producing output.
3833 3834 3835	The encryption operation MUST have been initialized with <b>C_EncryptInit</b> . This function may be called any number of times in succession. A call to <b>C_EncryptUpdate</b> which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current encryption operation.
3836 3837	The plaintext and ciphertext can be in the same place, <i>i.e.</i> , it is OK if <i>pPart</i> and <i>pEncryptedPart</i> point to the same location.
3838 3839 3840 3841 3842	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.
3843	Example: see <b>C_EncryptFinal.</b>
3844	5.8.4 C_EncryptFinal
3845 3846 3847 3848 3849	CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal)( CK_SESSION_HANDLE hSession, CK_BYTE_PTR pLastEncryptedPart, CK_ULONG_PTR pulLastEncryptedPartLen );
3850 3851 3852	<b>C_EncryptFinal</b> finishes a multiple-part encryption operation. <i>hSession</i> is the session's handle; <i>pLastEncryptedPart</i> points to the location that receives the last encrypted data part, if any; <i>pulLastEncryptedPartLen</i> points to the location that holds the length of the last encrypted data part.
3853	C_EncryptFinal uses the convention described in Section 5.2 on producing output.
3854 3855 3856 3857	The encryption operation MUST have been initialized with <b>C_EncryptInit</b> . A call to <b>C_EncryptFinal</b> always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call ( <i>i.e.</i> , one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.
3858 3859 3860	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 <b>C_EncryptFinal</b> will fail with return code CKR_DATA_LEN_RANGE.
3861	Return values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,

- CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, 3862
- 3863 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,
- CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, 3864
- 3865 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.
- 3866 Example:

	-				
3867	#define	PLAINTEXT_	BUF	SZ	200

3868 #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 = {
         CKM DES CBC PAD, iv, sizeof(iv)
3875
3876
       };
3877
       CK BYTE data[PLAINTEXT BUF SZ];
3878
       CK BYTE encryptedData[CIPHERTEXT BUF SZ];
3879
       CK ULONG ulEncryptedDatalLen;
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,
           &encryptedData[0], &ulEncryptedDatalLen);
3895
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	<pre>sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;</pre>
3915	rv = C_EncryptFinal(
3916	hSession,
3917	<pre>&amp;encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],</pre>
3918	<pre>&amp;ulEncryptedData3Len);</pre>
3919	if (rv != CKR_OK) {
3920	
3921	
3922	}
3923	}

# 3924 **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.

3928 Cryptoki provides the following functions for message-based encryption:

# 3929 5.9.1 C\_MessageEncryptInit

```
3930
3931
3932
```

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageEncryptInit)( CK\_SESSION\_HANDLE hSession, CK\_MECHANISM\_PTR pMechanism, CK\_OBJECT\_HANDLE hKey );

3933 3934

3935 C\_MessageEncryptInit prepares a session for one or more encryption operations that use the same
 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

3941 message in a single part, or call **C\_EncryptMessageBegin**, followed by **C\_EncryptMessageNext** one or 3942 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

3944 message-based encryption process.

3945 C\_MessageEncryptInit can be called with *pMechanism* set to NULL\_PTR to terminate a message-based
 active operation 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.

- 3949 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, 3950 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,
- 3951 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED,
- 3952 CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT,
- 3953 CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK,
- 3954 CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED,
- 3955 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,
- 3956 CKR\_OPERATION\_CANCEL\_FAILED.

### 3957 **5.9.2 C\_EncryptMessage**

3958	CK DECLARE FUNCTION(CK RV, C EncryptMessage)(
3959	CK SESSION HANDLE hSession,
3960	CK_VOID_PTR pParameter,
3961	CK ULONG ulParameterLen,
3962	CK_BYTE_PTR pAssociatedData,
3963	CK_ULONG ulAssociatedDataLen,
3964	CK_BYTE_PTR pPlaintext,
3965	CK_ULONG ulPlaintextLen,
3966	CK BYTE PTR pCiphertext,
3967	CK ULONG PTR pulCiphertextLen
3968	);
3969	<b>C EncryptMessage</b> encrypts a message in a single part <i>b</i> Session is the session's handle: <i>p</i> Parameter

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.

3975 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.

- 3979 If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and 3980 should be set to (NULL, 0).
- 3981 **C\_EncryptMessage** uses the convention described in Section 5.2 on producing output.
- 3982 The message-based encryption process MUST have been initialized with **C\_MessageEncryptInit**. A call 3983 to **C EncryptMessage** begins and terminates a message encryption operation.
- 3984 **C\_EncryptMessage** cannot be called in the middle of a multi-part message encryption operation.
- 3985 For some encryption mechanisms, the input plaintext data has certain length constraints (either because 3986 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
- 3987 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
- 3988 **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.

3992 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,

- 3993 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,
- 3994 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,
- 3995 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,
- 3996 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

# 3997 5.9.3 C\_EncryptMessageBegin

```
3998 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin)(
3999 CK_SESSION_HANDLE hSession,
4000 CK_VOID_PTR pParameter,
4001 CK_ULONG ulParameterLen,
4002 CK_BYTE_PTR pAssociatedData,
4003 CK_ULONG ulAssociatedDataLen
4004 );
```

4005 **C\_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the 4006 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.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C\_EncryptMessageNext**, *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.

- 4047 **C\_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.
- The message encryption operation MUST have been started with **C\_EncryptMessageBegin**. This function may be called any number of times in succession. A call to **C\_EncryptMessageBegin**. This which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message encryption operation. A call to **C\_EncryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE always terminates the active message 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.

Although the last C\_EncryptMessageNext call ends the encryption of a message, it does not finish the
 message-based encryption process. Additional C\_EncryptMessage or C\_EncryptMessageBegin and
 C EncryptMessageNext calls may be made on the session.

The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart* 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 4071 4072	CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext)( CK_SESSION_HANDLE hSession );
4073 4074	<b>C_MessageEncryptFinal</b> finishes a message-based encryption process. hSession is the session's handle.
4075	The message-based encryption process MUST have been initialized with <b>C_MessageEncryptInit</b> .
4076 4077 4078	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,
4079 4080	CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, 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
         &qcmParams, sizeof(qcmParams),
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),
         &auth[1][0], sizeof(auth[1])
4124
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
         \cap
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);
```

# 4146 **5.10 Decryption functions**

4147 Cryptoki provides the following functions for decrypting data:

### 4148 **5.10.1** C\_DecryptInit

4149 4150 4151 4152 4153	CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit)( CK_SESSION_HANDLE hSession, CK_MECHANISM_PTR pMechanism, CK_OBJECT_HANDLE hKey );
4154 4155	<b>C_DecryptInit</b> initializes a decryption operation. <i>hSession</i> is the session's handle; <i>pMechanism</i> points to the decryption mechanism; <i>hKey</i> is the handle of the decryption key.
4156 4157	The <b>CKA_DECRYPT</b> attribute of the decryption key, which indicates whether the key supports decryption, MUST be CK_TRUE.
4158 4159 4160 4161 4162	After calling <b>C_DecryptInit</b> , the application can either call <b>C_Decrypt</b> to decrypt data in a single part; or call <b>C_DecryptUpdate</b> zero or more times, followed by <b>C_DecryptFinal</b> , to decrypt data in multiple parts. The decryption operation is active until the application uses a call to <b>C_Decrypt</b> or <b>C_DecryptFinal</b> to actually obtain the final piece of plaintext. To process additional data (in single or multiple parts), the application MUST call <b>C_DecryptInit</b> again.
4163 4164 4165	<b>C_DecryptInit</b> can be called with <i>pMechanism</i> set to NULL_PTR to terminate an active decryption operation. If an active operation cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.
4166 4167 4168 4169 4170 4171 4172 4173	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.
4174	Example: see <b>C_DecryptFinal</b> .

### 4175 **5.10.2 C\_Decrypt**

4176	CK DECLARE FUNCTION(CK RV, C Decrypt)(
4177	CK SESSION HANDLE hSession,
4178	CK BYTE PTR pEncryptedData,
4179	CK_ULONG ulEncryptedDataLen,
4180	CK BYTE PTR pData,
4181	CK ULONG PTR pulDataLen
4182	);

- 4183 **C\_Decrypt** decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData* 4184 points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the 4185 location that receives the recovered data; *pulDataLen* points to the location that holds the length of the 4186 recovered data.
- 4187 **C\_Decrypt** uses the convention described in Section 5.2 on producing output.
- 4188 The decryption operation MUST have been initialized with **C\_DecryptInit**. A call to **C\_Decrypt** always
- 4189 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.
- 4192 C\_Decrypt cannot be used to terminate a multi-part operation, and MUST be called after C\_DecryptInit
   4193 without intervening C\_DecryptUpdate calls.

- The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
- the same location.
- 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;
- *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to
   the location that holds the length of the last recovered data part.
- 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,
       CKR CRYPTOKI NOT INITIALIZED, CKR DEVICE ERROR, CKR DEVICE MEMORY,
4249
       CKR DEVICE REMOVED, CKR ENCRYPTED DATA INVALID,
4250
       CKR ENCRYPTED DATA LEN RANGE, CKR FUNCTION CANCELED, CKR FUNCTION FAILED,
4251
       CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK, CKR OPERATION NOT INITIALIZED,
4252
       CKR SESSION CLOSED, CKR SESSION HANDLE INVALID, CKR USER NOT LOGGED IN.
4253
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
```

### 5.11 Message-based decryption functions 4309

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 authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm. 4312

4313 Cryptoki provides the following functions for message-based decryption.

### 5.11.1 C MessageDecryptInit 4314

```
CK DECLARE FUNCTION (CK RV, C MessageDecryptInit) (
4315
4316
         CK SESSION HANDLE hSession,
4317
         CK MECHANISM PTR pMechanism,
         CK OBJECT HANDLE hKey
4318
       );
```

- 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
- C DecryptMessageNext one or more times, to decrypt an encrypted message in multiple parts. This 4328 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.
- The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to 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.

# 4381 5.11.3 C\_DecryptMessageBegin

4382	CK DECLARE FUNCTION(CK RV, C DecryptMessageBegin)(
4383	CK SESSION HANDLE hSession,
4384	CK VOID PTR pParameter,
4385	CK ULONG ulParameterLen,
4386	CK BYTE PTR pAssociatedData,
4387	CK ULONG ulAssociatedDataLen
4388	); —

4389 **C\_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the 4390 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the 4391 message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data 4392 for an AEAD mechanism.

4393 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of 4394 **C\_EncryptMessageBegin**, *pParameter* is always an input parameter.

4395 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and 4396 should be set to (NULL, 0).

4397 After calling **C\_DecryptMessageBegin**, the application should call **C\_DecryptMessageNext** one or

4398 more times to decrypt the encrypted message in multiple parts. The message decryption operation is

4399 active until the application uses a call to C\_DecryptMessageNext with flags=CKF\_END\_OF\_MESSAGE

4400 to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or

4401 multiple parts), the application MUST call **C\_DecryptMessage** or **C\_DecryptMessageBegin** again.

4402 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,

- 4403 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,
- 4404 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,
- 4405 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,
- 4406 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### 4407 5.11.4 C\_DecryptMessageNext

4408	CK DECLARE FUNCTION(CK RV, C DecryptMessageNext)(
4409	CK SESSION HANDLE hSession,
4410	CK VOID PTR pParameter,
4411	CK_ULONG ulParameterLen,
4412	CK BYTE PTR pCiphertextPart,
4413	CK ULONG ulCiphertextPartLen,
4414	CK_BYTE_PTR pPlaintextPart,
4415	CK_ULONG_PTR pulPlaintextPartLen,
4416	CK_FLAGS flags
4417	);

4418 C\_DecryptMessageNext continues a multiple-part message decryption operation, processing another
4419 encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
4420 mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the
4421 encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart*4422 points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location
4423 that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to
4424 follow, or set to CKF\_END\_OF\_MESSAGE if this is the last ciphertext part.

4425 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of 4426 **C EncryptMessageNext**, *pParameter* is always an input parameter.

4427 **C\_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.

4428 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.
- The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart* point to the same location.
- Although the last **C\_DecryptMessageNext** call ends the decryption of a message, it does not finish the message-based decryption process. Additional **C\_DecryptMessage** or **C\_DecryptMessageBegin** and
- 4439 **C\_DecryptMessageNext c**alls 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

CK\_SESSION\_HANDLE hSession
);

- 4457 **C\_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's handle.
- 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	<b>C_DigestInit</b> initializes a message-digesting operation. <i>hSession</i> is the session's handle; <i>pMechanism</i>

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.
- The digest operation MUST have been initialized with **C\_DigestInit**. A call to **C\_Digest** always terminates the active digest operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the message digest.
- 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.

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.

The message-digesting operation MUST have been initialized with **C\_DigestInit**. Calls to this function and **C\_DigestKey** may be interspersed any number of times in any order. A call to **C\_DigestUpdate** which results in an error terminates the current digest operation.

- 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 4540	<b>C_DigestKey</b> continues a multiple-part message-digesting operation by digesting the value of a secret key. <i>hSession</i> is the session's handle; <i>hKey</i> is the handle of the secret key to be digested.
4541 4542	The message-digesting operation MUST have been initialized with <b>C_DigestInit</b> . Calls to this function and <b>C_DigestUpdate</b> may be interspersed any number of times in any order.
4543 4544	If the value of the supplied key cannot be digested purely for some reason related to its length, <b>C_DigestKey</b> 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 4553 4554 4555	CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal)( CK_SESSION_HANDLE hSession, CK_BYTE_PTR pDigest, CK_ULONG_PTR pulDigestLen
4556	);
4557 4558 4559	<b>C_DigestFinal</b> finishes a multiple-part message-digesting operation, returning the message digest. <i>hSession</i> is the session's handle; <i>pDigest</i> points to the location that receives the message digest; <i>pulDigestLen</i> 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 4562 4563 4564	The digest operation MUST have been initialized with <b>C_DigestInit</b> . A call to <b>C_DigestFinal</b> always terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call ( <i>i.e.</i> , one which returns CKR_OK) to determine the length of the buffer needed to hold the message digest.
4565 4566 4567 4568 4569	Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
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;
       CK RV rv;
4579
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
       .
```

# 4605 **5.13 Signing and MACing functions**

4606 Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations 4607 also encompass message authentication codes).

## 4608 **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 );
```

4614 **C\_SignInit** initializes a signature operation, where the signature is an appendix to the data. *hSession* is 4615 the session's handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature 4616 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.

### 4664 **5.13.3 C\_SignUpdate**

```
4665 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate)(
4666 CK_SESSION_HANDLE hSession,
4667 CK_BYTE_PTR pPart,
4668 CK_ULONG ulPartLen
4669 );
```

4670 **C\_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is 4671 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

- 4672 The signature operation MUST have been initialized with **C\_SignInit**. This function may be called any 4673 number of times in succession. A call to **C\_SignUpdate** which results in an error terminates the current 4674 signature operation.
- 4675 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,
- 4676 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,
- 4677 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,
- 4678 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,
- 4679 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,
- 4680 CKR\_TOKEN\_RESOURCE\_EXCEEDED.
- 4681 Example: see **C\_SignFinal**.

### 4682 **5.13.4 C\_SignFinal**

```
CK DECLARE FUNCTION(CK RV, C SignFinal)(
4683
4684
          CK SESSION HANDLE hSession,
4685
          CK BYTE PTR pSignature,
4686
          CK ULONG PTR pulSignatureLen
4687
       );
4688
       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
4689
       the location that holds the length of the signature.
4690
4691
       C SignFinal uses the convention described in Section 5.2 on producing output.
4692
       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
4693
4694
       call (i.e., one which returns CKR OK) to determine the length of the buffer needed to hold the signature.
4695
       Return values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,
       CKR CRYPTOKI NOT INITIALIZED, CKR DATA LEN RANGE, CKR DEVICE ERROR,
4696
4697
       CKR DEVICE MEMORY, CKR DEVICE REMOVED, CKR FUNCTION CANCELED,
4698
       CKR FUNCTION FAILED, CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK,
       CKR OPERATION NOT INITIALIZED, CKR SESSION CLOSED, CKR SESSION HANDLE INVALID,
4699
       CKR USER NOT LOGGED IN, CKR FUNCTION REJECTED,
4700
4701
       CKR TOKEN RESOURCE EXCEEDED.
4702
       Example:
4703
       CK SESSION HANDLE hSession;
4704
       CK OBJECT HANDLE hKey;
4705
       CK MECHANISM mechanism = {
4706
          CKM DES MAC, NULL PTR, 0
4707
       };
4708
       CK BYTE data[] = \{\ldots\};
4709
       CK BYTE mac[4];
4710
        CK ULONG ulMacLen;
4711
       CK RV rv;
       pkcs11-base-v3.0-os
                                                                                          15 June 2020
       Standards Track Work Product
                                   Copyright © OASIS Open 2020. All Rights Reserved.
                                                                                        Page 129 of 167
```

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

# 4725 5.13.5 C\_SignRecoverInit

4726 4727 4728 4729 4730	CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit)( CK_SESSION_HANDLE hSession, CK_MECHANISM_PTR pMechanism, CK_OBJECT_HANDLE hKey );
4731 4732 4733	<b>C_SignRecoverInit</b> initializes a signature operation, where the data can be recovered from the signature. <i>hSession</i> is the session's handle; <i>pMechanism</i> points to the structure that specifies the signature mechanism; $hKey$ is the handle of the signature key.
4734 4735	The <b>CKA_SIGN_RECOVER</b> 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.
4736 4737 4738 4739	After calling <b>C_SignRecoverInit</b> , the application may call <b>C_SignRecover</b> to sign in a single part. The signature operation is active until the application uses a call to <b>C_SignRecover</b> to actually obtain the signature. To process additional data in a single part, the application MUST call <b>C_SignRecoverInit</b> again.
4740 4741 4742	<b>C_SignRecoverInit</b> can be called with <i>pMechanism</i> 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.
4743 4744 4745 4746 4747 4748 4749 4750	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.
4751	Example: see <b>C_SignRecover</b> .

## 4752 **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.
       hSession is the session's handle; pData points to the data; uLDataLen is the length of the data;
4761
       pSignature points to the location that receives the signature: pulSignatureLen points to the location that
4762
       holds the length of the signature.
4763
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
       always terminates the active signing operation unless it returns CKR BUFFER TOO SMALL or is a
4766
       successful call (i.e., one which returns CKR OK) to determine the length of the buffer needed to hold the
4767
4768
       signature.
4769
       Return values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,
       CKR CRYPTOKI NOT INITIALIZED, CKR DATA INVALID, CKR DATA LEN RANGE,
4770
       CKR DEVICE ERROR, CKR DEVICE MEMORY, CKR DEVICE REMOVED,
4771
       CKR FUNCTION CANCELED, CKR FUNCTION FAILED, CKR GENERAL ERROR.
4772
       CKR HOST MEMORY, CKR OK, CKR OPERATION NOT INITIALIZED, CKR SESSION CLOSED,
4773
       CKR SESSION HANDLE INVALID, CKR USER NOT LOGGED IN,
4774
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[] = \{\ldots\};
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
- 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 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*
- is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
   *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
- in multiple parts. The verification operation is active until the application calls C\_Verify or C\_VerifyFinal.
   To process additional data (in single or multiple parts), the application MUST call C VerifyInit again.
- 4950 **C VerifyInit** can be called with *pMechanism* set to NULL PTR to terminate an active verification
- 4950 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

CK DECLARE FUNCTION(CK RV, C Verify)(
CK SESSION HANDLE hSession,
CK BYTE PTR pData,
CK ULONG ulDataLen,
CK BYTE PTR pSignature,
CK ULONG ulSignatureLen
);

- 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,
          CK BYTE PTR pSignature,
5012
          CK ULONG ulSignatureLen
5013
5014
        );
        C VerifyFinal finishes a multiple-part verification operation, checking the signature, hSession is the
5015
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
        signature is valid) or CKR SIGNATURE INVALID (indicating that the supplied signature is invalid). If the
5020
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,
        CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK, CKR OPERATION NOT INITIALIZED,
5027
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[] = \{\ldots\};
5037
        CK BYTE mac[4];
5038
        CK RV rv;
        pkcs11-base-v3.0-os
                                                                                           15 June 2020
        Standards Track Work Product
                                   Copyright © OASIS Open 2020. All Rights Reserved.
                                                                                         Page 136 of 167
```

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.

After calling C\_VerifyRecoverInit, the application may call C\_VerifyRecover to verify a signature on
 data in a single part. The verification operation is active until the application uses a call to
 C VerifyRecover to actually obtain the recovered message.

- 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 5086 5087 5088	<b>C_VerifyRecover</b> verifies a signature in a single-part operation, where the data is recovered from the signature. <i>hSession</i> is the session's handle; <i>pSignature</i> points to the signature; <i>ulSignatureLen</i> is the length of the signature; <i>pData</i> points to the location that receives the recovered data; and <i>pulDataLen</i> 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 5091 5092 5093	The verification operation MUST have been initialized with <b>C_VerifyRecoverInit</b> . A call to <b>C_VerifyRecover</b> always terminates the active verification operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call ( <i>i.e.</i> , one which returns CKR_OK) to determine the length of the buffer needed to hold the recovered data.
5094 5095 5096 5097 5098 5099 5100	A successful call to <b>C_VerifyRecover</b> should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code CKR_BUFFER_TOO_SMALL, <i>i.e.</i> , if <b>C_VerifyRecover</b> is supplied with an invalid signature, it will never return CKR_BUFFER_TOO_SMALL.
5101 5102 5103 5104 5105 5106 5107	Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID, CKR_TOKEN_RESOURCE_EXCEEDED.
5108	Example:
5109	CK_SESSION_HANDLE hSession;
5110	CK_OBJECT_HANDLE hKey;
5110 5111	CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = {
5110 5111 5112	CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0
5110 5111 5112 5113	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 };</pre>
5110 5111 5112 5113 5114	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {};</pre>
5110 5111 5112 5113 5114 5115	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen;</pre>
5110 5111 5112 5113 5114 5115 5116	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128];</pre>
5110 5111 5112 5113 5114 5115 5116 5117	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv;</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv;</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; .</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; .</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey);</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey); if (rv == CKR_OK) {</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey); if (rv == CKR_OK) { ulDataLen = sizeof(data);</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123 5124	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey); if (rv == CKR_OK) { ulDataLen = sizeof(data); rv = C_VerifyRecover(</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123 5124 5125	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey); if (rv == CKR_OK) { ulDataLen = sizeof(data); rv = C_VerifyRecover( hSession, signature, sizeof(signature), data, &amp;ulDataLen);</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123 5124 5125 5126	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv;</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123 5124 5125 5126 5127	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv;</pre>
5110 5111 5112 5113 5114 5115 5116 5117 5118 5119 5120 5121 5122 5123 5124 5125 5126 5127 5128	<pre>CK_OBJECT_HANDLE hKey; CK_MECHANISM mechanism = { CKM_RSA_9796, NULL_PTR, 0 }; CK_BYTE data[] = {}; CK_ULONG ulDataLen; CK_ULONG ulDataLen; CK_BYTE signature[128]; CK_RV rv; rv = C_VerifyRecoverInit(hSession, &amp;mechanism, hKey); if (rv == CKR_OK) { ulDataLen = sizeof(data); rv = C_VerifyRecover( hSession, signature, sizeof(signature), data, &amp;ulDataLen); .</pre>

# 5129 5.16 Message-based functions for verifying signatures and MACs

- 5130 Message-based verification refers to the process of verifying signatures on multiple messages using the 5131 same verification mechanism and verification key.
- 5132 Cryptoki provides the following functions for verifying signatures on messages (for the purposes of
- 5133 Cryptoki, these operations also encompass message authentication codes).

# 5134 5.16.1 C\_MessageVerifyInit

5135 5136 CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageVerifyInit)(

- 5136 CK\_SESSION\_HANDLE hSession,
- 5137 CK\_MECHANISM\_PTR pMechanism,
- 5138 CK\_OBJECT\_HANDLE hKey
- 5139

):

5140 **C\_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or
 5141 more verification operations (where the signature is an appendix to the data) that use the same
 5142 verification mechanism and verification key. *hSession* is the session's handle; *pMechanism* points to the
 5143 structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

- 5144 The **CKA\_VERIFY** attribute of the verification key, which indicates whether the key supports verification 5145 where the signature is an appendix to the data, MUST be CK\_TRUE.
- 5146 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.
- 5149 The message-based verification process is active until the application calls **C\_MessageVerifyFinal** to 5150 finish the message-based verification process.
- 5151 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,
- 5152 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,
- 5153 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,
- 5154 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,
- 5155 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,
- 5156 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,
- 5157 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

# 5158 **5.16.2** C\_VerifyMessage

- 5159 CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyMessage)(
  5160 CK\_SESSION\_HANDLE hSession,
  5161 CK\_VOID\_PTR pParameter,
  5162 CK\_ULONG\_ulParameterLen,
- 5163 CK\_BYTE\_PTR pData,
- 5164 CK\_ULONG ulDataLen,
- 5165 CK\_BYTE\_PTR pSignature,
- 5166 CK\_ULONG ulSignatureLen
- 5167

);

5168 **C\_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an 5169 appendix to the data. **C\_MessageVerifyInit** must previously been called on the session. *hSession* is the 5170 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the 5171 message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* 5172 points to the signature; *ulSignatureLen* is the length of the signature.

5173 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
- supplied signature is valid) or CKR SIGNATURE INVALID (indicating that the supplied signature is 5177
- 5178 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
- CKR SIGNATURE LEN RANGE should be returned. 5179
- C VerifyMessage does not finish the message-based verification process. Additional C VerifyMessage 5180 or C VerifyMessageBegin and C\_VerifyMessageNext calls may be made on the session. 5181
- For most mechanisms, C VerifyMessage is equivalent to C VerifyMessageBegin followed by a 5182 5183 sequence of C VerifyMessageNext operations.
- Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, 5184
- CKR DATA LEN RANGE, CKR DEVICE ERROR, CKR DEVICE MEMORY, 5185
- CKR DEVICE REMOVED, CKR FUNCTION CANCELED, CKR FUNCTION FAILED, 5186
- CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, 5187
- 5188 CKR SESSION CLOSED, CKR SESSION HANDLE INVALID, CKR SIGNATURE INVALID,
- CKR SIGNATURE LEN RANGE, CKR TOKEN RESOURCE EXCEEDED. 5189

#### 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

- **C** VerifyMessageBegin begins a multiple-part message verification operation, where the signature is an 5196 appendix to the message. C\_MessageVerifyInit must previously been called on the session. hSession is 5197 the session's handle; pParameter and ulParameterLen specify any mechanism-specific parameters for 5198 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,
- CKR FUNCTION CANCELED, CKR FUNCTION FAILED, CKR GENERAL ERROR. 5208
- CKR HOST MEMORY, CKR OK, CKR OPERATION ACTIVE, CKR PIN EXPIRED, 5209
- 5210 CKR SESSION CLOSED, CKR SESSION HANDLE INVALID, CKR USER NOT LOGGED IN.

### 5.16.4 C VerifyMessageNext 5211

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
- *pSignature* which results in an error terminates the current message verification operation. A call to
   **C\_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification 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
- b235 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its
   b236 length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active
   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
       );
       C DigestEncryptUpdate continues multiple-part digest and encryption operations, processing another
5271
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
       pEncryptedPart has the value NULL PTR, or because pulEncryptedPartLen is too small to hold the entire
5277
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
       C DigestInit and C EncryptInit, respectively). This function may be called any number of times in
5280
       succession, and may be interspersed with C_DigestUpdate, C_DigestKey, and C_EncryptUpdate calls
5281
5282
       (it would be somewhat unusual to intersperse calls to C DigestEncryptUpdate with calls to
5283
       C DigestKey, however).
       Return values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,
5284
5285
       CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
       CKR DEVICE MEMORY, CKR DEVICE REMOVED, CKR FUNCTION CANCELED,
5286
       CKR FUNCTION FAILED, CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK,
5287
       CKR OPERATION NOT INITIALIZED, CKR SESSION CLOSED, CKR SESSION HANDLE INVALID.
5288
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;
       int i;
5307
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++)</pre>
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.

Decryption and digesting operations MUST both be active (they MUST have been initialized with
 C\_DecryptInit and C\_DigestInit, respectively). This function may be called any number of times in
 succession, and may be interspersed with C\_DecryptUpdate, C\_DigestUpdate, and C\_DigestKey calls
 (it would be somewhat unusual to intersperse calls to C\_DigestEncryptUpdate with calls to
 C\_DigestKey, however).

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

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.
        Since there is no more ciphertext, the application calls C_DecryptFinal. This tells Cryptoki that there's
5407
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
        bytes of plaintext are not passed on to be digested.
5410
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,
        the last 2 bytes of plaintext get passed into the active digesting operation via a C_DigestUpdate call.
5413
5414
        Because of this, it is critical that when an application uses a padded decryption mechanism with
        C DecryptDigestUpdate, it knows exactly how much plaintext has been passed into the active digesting
5415
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,
        CKR DEVICE REMOVED. CKR ENCRYPTED DATA INVALID.
5420
        CKR ENCRYPTED DATA LEN RANGE, CKR FUNCTION CANCELED, CKR FUNCTION FAILED,
5421
        CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK, CKR OPERATION NOT INITIALIZED,
5422
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

```
CK DECLARE FUNCTION (CK RV, C SignEncryptUpdate) (
5507
5508
          CK SESSION HANDLE hSession,
5509
          CK BYTE PTR pPart,
          CK ULONG ulPartLen,
5510
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.
       C SignEncryptUpdate uses the convention described in Section 5.2 on producing output. If a
5518
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.
       Signature and encryption operations MUST both be active (they MUST have been initialized with
5522
5523
       C SignInit and C EncryptInit, respectively). This function may be called any number of times in
       succession, and may be interspersed with C SignUpdate and C EncryptUpdate calls.
5524
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,
       CKR FUNCTION FAILED. CKR GENERAL ERROR. CKR HOST MEMORY. CKR OK.
5528
       CKR OPERATION NOT INITIALIZED, CKR SESSION CLOSED, CKR SESSION HANDLE INVALID,
5529
       CKR USER NOT LOGGED IN.
5530
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
      ulEncryptedDataLen = sizeof(encryptedData);
5566
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++)</pre>
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
- C SignEncryptUpdate, the "inverse function" of C DecryptVerifyUpdate. This is because when 5633
- **C** SignEncryptUpdate is called, precisely the same input is passed to both the active signing operation 5634 and the active encryption operation; however, when C DecryptVerifyUpdate is called, the input passed 5635
- to the active verifying operation is the *output of* the active decryption operation. This issue comes up only 5636 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 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this 5639 ciphertext and verify a signature on the original plaintext thereby obtained. 5640
- 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 ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation. 5644
- 5645 Since there is no more ciphertext, the application calls C DecryptFinal. This tells Cryptoki that there's no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active 5646
- 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
- C DecryptVerifyUpdate, it knows exactly how much plaintext has been passed into the active 5654
- 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,
- CKR CRYPTOKI NOT INITIALIZED, CKR DATA LEN RANGE, CKR DEVICE ERROR, 5658
- CKR DEVICE MEMORY, CKR DEVICE REMOVED, CKR ENCRYPTED DATA INVALID. 5659
- CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, 5660
- CKR GENERAL ERROR, CKR HOST MEMORY, CKR OK, CKR OPERATION NOT INITIALIZED, 5661
- CKR\_SESSION\_CLOSED, CKR\_SESSION HANDLE INVALID. 5662
- 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
      ulDataLen = sizeof(data);
5697
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
       }
       rv = C VerifyFinal(hSession, MAC, ulMacLen);
5739
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 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	<pre>CK_BYTE publicExponent[] = { 3 };</pre>
5844	<pre>CK_BYTE subject[] = {};</pre>
5845	CK_BYTE id[] = {123};
5846	CK_BBOOL true = CK_TRUE;
5847	<pre>CK_ATTRIBUTE publicKeyTemplate[] = {</pre>
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	<pre>CK_ATTRIBUTE privateKeyTemplate[] = {</pre>
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
```

#### 5875 **5.18.3 C\_WrapKey**

CK DECLARE FUNCTION(CK RV, C WrapKey)(
CK_SESSION_HANDLE hSession,
CK_MECHANISM_PTR pMechanism,
CK OBJECT HANDLE hWrappingKey,
CK OBJECT HANDLE hKey,
CK BYTE PTR pWrappedKey,
CK_ULONG PTR pulWrappedKeyLen
);

5884 **C\_WrapKey** wraps (*i.e.*, encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism* 5885 points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle 5886 of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and 5887 *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

5888 **C\_WrapKey** uses the convention described in Section 5.2 on producing output.

5889 The **CKA\_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping, 5890 MUST be CK\_TRUE. The **CKA\_EXTRACTABLE** attribute of the key to be wrapped MUST also be 5891 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.

- 5896 **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.
- 5901 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
       attribute template and the size is the number of items in the template times the size of CK ATTRIBUTE. If
5906
       this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
5907
       checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
5908
5909
       CKR KEY HANDLE INVALID.
5910
       Return Values: CKR ARGUMENTS BAD, CKR BUFFER TOO SMALL,
       CKR CRYPTOKI NOT INITIALIZED, CKR DEVICE ERROR, CKR DEVICE MEMORY,
5911
5912
       CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
       CKR GENERAL ERROR, CKR HOST MEMORY, CKR KEY HANDLE INVALID,
5913
       CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
5914
       CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5915
       CKR OPERATION ACTIVE, CKR PIN EXPIRED, CKR SESSION CLOSED,
5916
       CKR SESSION HANDLE INVALID, CKR USER NOT LOGGED IN,
5917
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 5950 5951	CK_ULONG ulAttributeCount, CK_OBJECT_HANDLE_PTR phKey );
5952 5953 5954 5955 5956 5957	<b>C_UnwrapKey</b> unwraps ( <i>i.e.</i> decrypts) a wrapped key, creating a new private key or secret key object. <i>hSession</i> is the session's handle; <i>pMechanism</i> points to the unwrapping mechanism; <i>hUnwrappingKey</i> is the handle of the unwrapping key; <i>pWrappedKey</i> points to the wrapped key; <i>ulWrappedKeyLen</i> is the length of the wrapped key; <i>pTemplate</i> points to the template for the new key; <i>ulAttributeCount</i> is the number of attributes in the template; <i>phKey</i> points to the location that receives the handle of the recovered key.
5958 5959	The <b>CKA_UNWRAP</b> attribute of the unwrapping key, which indicates whether the key supports unwrapping, MUST be CK_TRUE.
5960 5961 5962	The new key will have the <b>CKA_ALWAYS_SENSITIVE</b> attribute set to CK_FALSE, and the <b>CKA_NEVER_EXTRACTABLE</b> attribute set to CK_FALSE. The <b>CKA_EXTRACTABLE</b> attribute is by default set to CK_TRUE.
5963 5964	Some mechanisms may modify, or attempt to modify. the contents of the pMechanism structure at the same time that the key is unwrapped.
5965 5966	If a call to <b>C_UnwrapKey</b> cannot support the precise template supplied to it, it will fail and return without creating any key object.
5967 5968 5969	The key object created by a successful call to <b>C_UnwrapKey</b> will have its <b>CKA_LOCAL</b> attribute set to CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).
5970 5971 5972 5973 5974 5975 5976 5977	To partition the unwrapping keys so they can only unwrap a subset of keys the attribute CKA_UNWRAP_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute conflict occurs on an attempt to unwrap a key then the function SHALL return CKR_TEMPLATE_INCONSISTENT.
5978 5979 5980 5981 5982 5983 5984 5985 5986 5987 5988 5989 5990	Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.
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_BITE 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,
       CKR TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
6049
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, O
6056
       };
6057
       CK BYTE prime[] = \{\ldots\};
6058
       CK BYTE base[] = \{\ldots\};
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
```

#### 6107 5.19 Random number generation functions

6108 Cryptoki provides the following functions for generating random numbers:

#### 6109 **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 );
```

#### 6115 **C\_SeedRandom** mixes additional seed material into the token's random number generator. *hSession* is 6116 the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed

6117 material.

```
6118 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
```

- 6119 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,
- 6120 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,
- 6121 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE,
- 6122 CKR\_RANDOM\_SEED\_NOT\_SUPPORTED, CKR\_RANDOM\_NO\_RNG, CKR\_SESSION\_CLOSED,
- 6123 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.
- 6124 Example: see **C\_GenerateRandom**.

#### 6125 **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,
       CKR FUNCTION CANCELED, CKR FUNCTION FAILED, CKR GENERAL ERROR.
6136
       CKR HOST MEMORY, CKR OK, CKR OPERATION ACTIVE, CKR RANDOM NO RNG,
6137
       CKR SESSION CLOSED, CKR SESSION HANDLE INVALID, CKR USER NOT LOGGED IN.
6138
6139
       Example:
6140
       CK SESSION HANDLE hSession;
6141
       CK BYTE seed[] = \{\ldots\};
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 );
```

```
6164In previous versions of Cryptoki, C_GetFunctionStatus obtained the status of a function running in6165parallel with an application. Now, however, C_GetFunctionStatus is a legacy function which should6166simply 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	);
	<u></u>

- 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 signatures and MACs; dual-purpose cryptographic functions; key management functions; random number 6186 generation functions) executing in Cryptoki sessions can periodically surrender control to the application 6187 who called them if the session they are executing in had a notification callback function associated with it 6188 when it was opened. They do this by calling the session's callback with the arguments (hSession, 6189 CKN\_SURRENDER, pApplication), where hSession is the session's handle and pApplication was 6190 supplied to C OpenSession when the session was opened. Surrender callbacks should return either the 6191 6192 value CKR OK (to indicate that Cryptoki should continue executing the function) or the value CKR CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before 6193 6194 returning one of these values, the callback function can perform some computation, if desired.

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

application-supplied notification callback routines should examine each callback they receive, and if they
 are unfamiliar with the type of that callback, they should immediately give control back to the library by
 returning with the value CKR OK.

### 6205 6 PKCS #11 Implementation Conformance

An implementation is a conforming implementation if it meets the conditions specified in one or more 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 .

### 6211 Appendix A. Acknowledgments

- 6212 The following individuals have participated in the creation of this specification and are gratefully 6213 acknowledged:
- 6214 Participants:
- 6215 Benton Stark Cisco Systems
- 6216 Anthony Berglas Cryptsoft Pty Ltd.
- 6217 Justin Corlett Cryptsoft Pty Ltd.
- 6218 Tony Cox Cryptsoft Pty Ltd.
- 6219 Tim Hudson Cryptsoft Pty Ltd.
- 6220 Bruce Rich Cryptsoft Pty Ltd.
- 6221 Greg Scott Cryptsoft Pty Ltd.
- 6222 Jason Thatcher Cryptsoft Pty Ltd.
- 6223 Magda Zdunkiewicz Cryptsoft Pty Ltd.
- 6224 Andrew Byrne Dell
- 6225 David Horton Dell
- 6226 Kevin Mooney Fornetix
- 6227 Gerald Stueve Fornetix
- 6228 Charles White Fornetix
- 6229 Matt Bauer Galois, Inc.
- 6230 Wan-Teh Chang Google Inc.
- 6231 Patrick Steuer IBM
- 6232 Michele Drgon Individual
- 6233 Gershon Janssen Individual
- 6234 Oscar So Individual
- 6235 Michelle Brochmann Information Security Corporation
- 6236 Michael Mrkowitz Information Security Corporation
- 6237 Jonathan Schulze-Hewett Information Security Corporation
- 6238 Philip Lafrance ISARA Corporation
- 6239 Thomas Hardjono M.I.T.
- 6240 Hamish Cameron nCipher
- 6241 Paul King nCipher
- 6242 Sander Temme nCipher
- 6243 Chet Ensign OASIS
- 6244 Jane Harnad OASIS
- 6245 Web Master OASIS
- 6246 Dee Schur OASIS
- 6247 Xuelei Fan Oracle
- 6248 Jan Friedel Oracle
- 6249 Susan Gleeson Oracle
- 6250 Dina Kurktchi-Nimeh Oracle
- 6251 John Leser Oracle

- 6252 Darren Moffat - Oracle 6253 Mark Joseph - P6R, Inc 6254 Jim Susoy - P6R, Inc 6255 Roland Bramm - PrimeKey Solutions AB 6256 Warren Armstrong - QuintessenceLabs Pty Ltd. 6257 Kenli Chong - QuintessenceLabs Pty Ltd. 6258 John Leiseboer - QuintessenceLabs Pty Ltd. 6259 Florian Poppa - QuintessenceLabs Pty Ltd. 6260 Martin Shannon - QuintessenceLabs Pty Ltd. 6261 Jakub Jelen - Red Hat 6262 Chris Malafis - Red Hat 6263 Robert Relyea - Red Hat 6264 Christian Bollich - Utimaco IS GmbH
- 6265 Dieter Bong Utimaco IS GmbH
- 6266 Chris Meyer Utimaco IS GmbH
- 6267 Daniel Minder Utimaco IS GmbH
- 6268 Roland Reichenberg Utimaco IS GmbH
- 6269 Manish Upasani Utimaco IS GmbH
- 6270 Steven Wierenga Utimaco IS GmbH

## 6271 Appendix B. Manifest constants

6272 The definitions for manifest constants specified in this document can be found in the following normative 6273 computer language definition files:

- include/pkcs11-v3.00/pkcs11.h
- 6275 include/pkcs11-v3.00/pkcs11t.h
- 6276 include/pkcs11-v3.00/pkcs11f.h

# 6277 Appendix C. Revision History

6278

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 mechnisms review" document
csprd 02 wd03	Dec 3 2019	Dieter Bong	Changes as per "PKCS11 base spec review" document

6279