



---

# PKCS #11 Specification Version 3.1

## Committee Specification Draft 01

16 February 2022

**This stage:**

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.pdf> (Authoritative)  
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.html>  
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.docx>

**Previous stage:**

N/A

**Latest stage:**

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.pdf> (Authoritative)  
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>  
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.docx>

**Technical Committee:**

OASIS PKCS 11 TC

**Chair:**

Robert Relyea ([rrelyea@redhat.com](mailto:rrelyea@redhat.com)), Red Hat

**Editors:**

Dieter Bong ([dieter.bong@utimaco.com](mailto:dieter.bong@utimaco.com)), Utimaco IS GmbH  
Tony Cox ([tony.cox@cryptsoft.com](mailto:tony.cox@cryptsoft.com)), Cryptsoft Pty Ltd

**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-spec/v3.1/csd01/include/pkcs11-v3.1/>

**Related work:**

This specification replaces or supersedes:

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

This specification is related to:

- *PKCS #11 Profiles Version 3.1*. Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html>.

**Abstract:**

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

**Status:**

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical

Committee (TC) are listed at [https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=pkcs11#technical](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.

#### **Key words:**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [[RFC2119](#)] and [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

#### **Citation format:**

When referencing this document, the following citation format should be used:

#### **[PKCS11-Spec-v3.1]**

*PKCS #11 Specification Version 3.1.* Edited by Dieter Bong and Tony Cox. 16 February 2022. OASIS Committee Specification Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>.

#### **Notices:**

Copyright © OASIS Open 2022. All Rights Reserved.

Distributed under the terms of the OASIS IPR Policy, [<https://www.oasis-open.org/policies-guidelines/ipr/>]. For complete copyright information please see the full Notices section in an Appendix below.

---

# Table of Contents

1	Introduction .....	18
1.1	Definitions .....	18
1.2	Symbols and abbreviations.....	20
1.3	Normative References .....	22
1.4	Non-Normative References .....	25
2	Platform- and compiler-dependent directives for C or C++.....	28
2.1	Structure packing .....	28
2.2	Pointer-related macros .....	28
3	General data types .....	30
3.1	General information .....	30
3.2	Slot and token types .....	31
3.3	Session types .....	36
3.4	Object types .....	37
3.5	Data types for mechanisms .....	41
3.6	Function types .....	44
3.7	Locking-related types.....	49
4	Objects .....	52
4.1	Creating, modifying, and copying objects.....	53
4.1.1	Creating objects .....	53
4.1.2	Modifying objects.....	54
4.1.3	Copying objects .....	54
4.2	Common attributes .....	54
4.3	Hardware Feature Objects.....	55
4.3.1	Definitions .....	55
4.3.2	Overview.....	55
4.3.3	Clock.....	56
4.3.3.1	Definition .....	56
4.3.3.2	Description .....	56
4.3.4	Monotonic Counter Objects.....	56
4.3.4.1	Definition .....	56
4.3.4.2	Description .....	56
4.3.5	User Interface Objects.....	56
4.3.5.1	Definition .....	56
4.3.5.2	Description .....	57
4.4	Storage Objects .....	57
4.4.1	The CKA_UNIQUE_ID attribute .....	58
4.5	Data objects .....	59
4.5.1	Definitions .....	59
4.5.2	Overview.....	59
4.6	Certificate objects .....	59
4.6.1	Definitions .....	59
4.6.2	Overview.....	59
4.6.3	X.509 public key certificate objects .....	60
4.6.4	WTLS public key certificate objects.....	62

4.6.5 X.509 attribute certificate objects .....	63
4.7 Key objects .....	64
4.7.1 Definitions.....	64
4.7.2 Overview.....	64
4.8 Public key objects .....	65
4.9 Private key objects.....	67
4.10 Secret key objects .....	69
4.11 Domain parameter objects.....	71
4.11.1 Definitions.....	71
4.11.2 Overview.....	71
4.12 Mechanism objects .....	72
4.12.1 Definitions.....	72
4.12.2 Overview.....	72
4.13 Profile objects .....	72
4.13.1 Definitions.....	72
4.13.2 Overview.....	72
5 Functions .....	74
5.1 Function return values .....	77
5.1.1 Universal Cryptoki function return values.....	78
5.1.2 Cryptoki function return values for functions that use a session handle .....	78
5.1.3 Cryptoki function return values for functions that use a token .....	78
5.1.4 Special return value for application-supplied callbacks .....	79
5.1.5 Special return values for mutex-handling functions .....	79
5.1.6 All other Cryptoki function return values .....	79
5.1.7 More on relative priorities of Cryptoki errors .....	84
5.1.8 Error code “gotchas”.....	85
5.2 Conventions for functions returning output in a variable-length buffer .....	85
5.3 Disclaimer concerning sample code.....	86
5.4 General-purpose functions .....	86
5.4.1 C_Initialize.....	86
5.4.2 C_Finalize.....	87
5.4.3 C_GetInfo .....	87
5.4.4 C_GetFunctionList.....	88
5.4.5 C_GetInterfaceList .....	89
5.4.6 C_GetInterface .....	90
5.5 Slot and token management functions .....	92
5.5.1 C_GetSlotList .....	92
5.5.2 C_GetSlotInfo .....	93
5.5.3 C_GetTokenInfo .....	94
5.5.4 C_WaitForSlotEvent .....	95
5.5.5 C_GetMechanismList .....	96
5.5.6 C_GetMechanismInfo.....	97
5.5.7 C_InitToken .....	97
5.5.8 C_InitPIN .....	99
5.5.9 C_SetPIN.....	99
5.6 Session management functions.....	100

5.6.1 C_OpenSession .....	101
5.6.2 C_CloseSession .....	101
5.6.3 C_CloseAllSessions .....	102
5.6.4 C_GetSessionInfo .....	103
5.6.5 C_SessionCancel .....	103
5.6.6 C_GetOperationState .....	105
5.6.7 C_SetOperationState .....	106
5.6.8 C_Login .....	108
5.6.9 C_LoginUser .....	109
5.6.10 C_Logout .....	110
5.7 Object management functions .....	111
5.7.1 C_CreateObject .....	111
5.7.2 C_CopyObject .....	113
5.7.3 C_DestroyObject .....	114
5.7.4 C_GetObjectSize .....	115
5.7.5 C_GetAttributeValue .....	116
5.7.6 C_SetAttributeValue .....	118
5.7.7 C_FindObjectsInit .....	119
5.7.8 C_FindObjects .....	119
5.7.9 C_FindObjectsFinal .....	120
5.8 Encryption functions .....	120
5.8.1 C_EncryptInit .....	120
5.8.2 C_Encrypt .....	121
5.8.3 C_EncryptUpdate .....	122
5.8.4 C_EncryptFinal .....	122
5.9 Message-based encryption functions .....	124
5.9.1 C_MessageEncryptInit .....	124
5.9.2 C_EncryptMessage .....	125
5.9.3 C_EncryptMessageBegin .....	125
5.9.4 C_EncryptMessageNext .....	126
5.9.5 C_MessageEncryptFinal .....	127
5.10 Decryption functions .....	129
5.10.1 C_DecryptInit .....	129
5.10.2 C_Decrypt .....	129
5.10.3 C_DecryptUpdate .....	130
5.10.4 C_DecryptFinal .....	131
5.11 Message-based decryption functions .....	132
5.11.1 C_MessageDecryptInit .....	132
5.11.2 C_DecryptMessage .....	133
5.11.3 C_DecryptMessageBegin .....	134
5.11.4 C_DecryptMessageNext .....	134
5.11.5 C_MessageDecryptFinal .....	135
5.12 Message digesting functions .....	135
5.12.1 C_DigestInit .....	136
5.12.2 C_Digest .....	136
5.12.3 C_DigestUpdate .....	137

5.12.4 C_DigestKey.....	137
5.12.5 C_DigestFinal .....	137
5.13 Signing and MACing functions.....	139
5.13.1 C_SignInit .....	139
5.13.2 C_Sign.....	139
5.13.3 C_SignUpdate .....	140
5.13.4 C_SignFinal .....	140
5.13.5 C_SignRecoverInit .....	141
5.13.6 C_SignRecover .....	142
5.14 Message-based signing and MACing functions .....	143
5.14.1 C_MessageSignInit .....	143
5.14.2 C_SignMessage .....	143
5.14.3 C_SignMessageBegin.....	144
5.14.4 C_SignMessageNext.....	145
5.14.5 C_MessageSignFinal .....	145
5.15 Functions for verifying signatures and MACs .....	146
5.15.1 C_VerifyInit.....	146
5.15.2 C_Verify.....	146
5.15.3 C_VerifyUpdate .....	147
5.15.4 C_VerifyFinal.....	147
5.15.5 C_VerifyRecoverInit .....	148
5.15.6 C_VerifyRecover .....	149
5.16 Message-based functions for verifying signatures and MACs .....	150
5.16.1 C_MessageVerifyInit .....	150
5.16.2 C_VerifyMessage .....	150
5.16.3 C_VerifyMessageBegin.....	151
5.16.4 C_VerifyMessageNext.....	152
5.16.5 C_MessageVerifyFinal .....	152
5.17 Dual-function cryptographic functions .....	153
5.17.1 C_DigestEncryptUpdate.....	153
5.17.2 C_DecryptDigestUpdate.....	155
5.17.3 C_SignEncryptUpdate .....	158
5.17.4 C_DecryptVerifyUpdate.....	161
5.18 Key management functions .....	163
5.18.1 C_GenerateKey.....	163
5.18.2 C_GenerateKeyPair .....	165
5.18.3 C_WrapKey .....	166
5.18.4 C_UnwrapKey .....	168
5.18.5 C_DeriveKey .....	169
5.19 Random number generation functions .....	171
5.19.1 C_SeedRandom.....	171
5.19.2 C_GenerateRandom .....	172
5.20 Parallel function management functions.....	173
5.20.1 C_GetFunctionStatus .....	173
5.20.2 C_CancelFunction.....	173
5.21 Callback functions.....	173

5.21.1 Surrender callbacks.....	173
5.21.2 Vendor-defined callbacks .....	174
6    Mechanisms .....	175
6.1 RSA.....	175
6.1.1 Definitions.....	175
6.1.2 RSA public key objects.....	176
6.1.3 RSA private key objects .....	177
6.1.4 PKCS #1 RSA key pair generation .....	179
6.1.5 X9.31 RSA key pair generation.....	179
6.1.6 PKCS #1 v1.5 RSA .....	179
6.1.7 PKCS #1 RSA OAEP mechanism parameters .....	180
6.1.8 PKCS #1 RSA OAEP .....	182
6.1.9 PKCS #1 RSA PSS mechanism parameters .....	182
6.1.10 PKCS #1 RSA PSS .....	183
6.1.11 ISO/IEC 9796 RSA.....	183
6.1.12 X.509 (raw) RSA .....	184
6.1.13 ANSI X9.31 RSA .....	185
6.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160 .....	186
6.1.15 PKCS #1 v1.5 RSA signature with SHA-224 .....	186
6.1.16 PKCS #1 RSA PSS signature with SHA-224 .....	186
6.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512.....	186
6.1.18 PKCS #1 v1.5 RSA signature with SHA3.....	187
6.1.19 PKCS #1 RSA PSS signature with SHA3 .....	187
6.1.20 ANSI X9.31 RSA signature with SHA-1 .....	187
6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA.....	188
6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP.....	188
6.1.23 RSA AES KEY WRAP .....	189
6.1.24 RSA AES KEY WRAP mechanism parameters .....	190
6.1.25 FIPS 186-4 .....	190
6.2 DSA.....	190
6.2.1 Definitions.....	191
6.2.2 DSA public key objects.....	192
6.2.3 DSA Key Restrictions .....	193
6.2.4 DSA private key objects .....	193
6.2.5 DSA domain parameter objects .....	194
6.2.6 DSA key pair generation .....	195
6.2.7 DSA domain parameter generation.....	195
6.2.8 DSA probabilistic domain parameter generation.....	195
6.2.9 DSA Shawe-Taylor domain parameter generation .....	196
6.2.10 DSA base domain parameter generation.....	196
6.2.11 DSA without hashing .....	196
6.2.12 DSA with SHA-1 .....	197
6.2.13 FIPS 186-4 .....	197
6.2.14 DSA with SHA-224 .....	197
6.2.15 DSA with SHA-256 .....	198

6.2.16 DSA with SHA-384 .....	198
6.2.17 DSA with SHA-512 .....	199
6.2.18 DSA with SHA3-224 .....	199
6.2.19 DSA with SHA3-256 .....	200
6.2.20 DSA with SHA3-384 .....	200
6.2.21 DSA with SHA3-512 .....	200
6.3 Elliptic Curve .....	201
6.3.1 EC Signatures .....	203
6.3.2 Definitions .....	203
6.3.3 Short Weierstrass Elliptic Curve public key objects .....	204
6.3.4 Short Weierstrass Elliptic Curve private key objects .....	205
6.3.5 Edwards Elliptic Curve public key objects .....	206
6.3.6 Edwards Elliptic Curve private key objects .....	207
6.3.7 Montgomery Elliptic Curve public key objects .....	208
6.3.8 Montgomery Elliptic Curve private key objects .....	209
6.3.9 Elliptic Curve key pair generation .....	210
6.3.10 Edwards Elliptic Curve key pair generation .....	211
6.3.11 Montgomery Elliptic Curve key pair generation .....	211
6.3.12 ECDSA without hashing .....	212
6.3.13 ECDSA with hashing .....	212
6.3.14 EdDSA .....	213
6.3.15 XEdDSA .....	213
6.3.16 EC mechanism parameters .....	214
6.3.17 Elliptic Curve Diffie-Hellman key derivation .....	219
6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation .....	219
6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation .....	220
6.3.20 ECDH AES KEY WRAP .....	221
6.3.21 ECDH AES KEY WRAP mechanism parameters .....	222
6.3.22 FIPS 186-4 .....	223
6.4 Diffie-Hellman .....	223
6.4.1 Definitions .....	223
6.4.2 Diffie-Hellman public key objects .....	224
6.4.3 X9.42 Diffie-Hellman public key objects .....	224
6.4.4 Diffie-Hellman private key objects .....	225
6.4.5 X9.42 Diffie-Hellman private key objects .....	226
6.4.6 Diffie-Hellman domain parameter objects .....	227
6.4.7 X9.42 Diffie-Hellman domain parameters objects .....	227
6.4.8 PKCS #3 Diffie-Hellman key pair generation .....	228
6.4.9 PKCS #3 Diffie-Hellman domain parameter generation .....	229
6.4.10 PKCS #3 Diffie-Hellman key derivation .....	229
6.4.11 X9.42 Diffie-Hellman mechanism parameters .....	230
6.4.12 X9.42 Diffie-Hellman key pair generation .....	232
6.4.13 X9.42 Diffie-Hellman domain parameter generation .....	233
6.4.14 X9.42 Diffie-Hellman key derivation .....	233
6.4.15 X9.42 Diffie-Hellman hybrid key derivation .....	234
6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation .....	234

6.5 Extended Triple Diffie-Hellman (x3dh) .....	235
6.5.1 Definitions .....	235
6.5.2 Extended Triple Diffie-Hellman key objects .....	235
6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange.....	235
6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange.....	236
6.5.5 Extended Triple Diffie-Hellman parameters .....	237
6.6 Double Ratchet .....	238
6.6.1 Definitions .....	238
6.6.2 Double Ratchet secret key objects .....	238
6.6.3 Double Ratchet key derivation .....	239
6.6.4 Double Ratchet Encryption mechanism .....	240
6.6.5 Double Ratchet parameters .....	241
6.7 Wrapping/unwrapping private keys .....	241
6.8 Generic secret key .....	243
6.8.1 Definitions .....	244
6.8.2 Generic secret key objects .....	244
6.8.3 Generic secret key generation .....	245
6.9 HMAC mechanisms .....	245
6.9.1 General block cipher mechanism parameters.....	245
6.10 AES .....	245
6.10.1 Definitions .....	246
6.10.2 AES secret key objects .....	246
6.10.3 AES key generation .....	247
6.10.4 AES-ECB.....	247
6.10.5 AES-CBC.....	248
6.10.6 AES-CBC with PKCS padding .....	248
6.10.7 AES-OFB.....	249
6.10.8 AES-CFB .....	249
6.10.9 General-length AES-MAC .....	250
6.10.10 AES-MAC .....	250
6.10.11 AES-XCBC-MAC .....	250
6.10.12 AES-XCBC-MAC-96.....	251
6.11 AES with Counter .....	251
6.11.1 Definitions .....	251
6.11.2 AES with Counter mechanism parameters .....	251
6.11.3 AES with Counter Encryption / Decryption.....	252
6.12 AES CBC with Cipher Text Stealing CTS.....	252
6.12.1 Definitions .....	252
6.12.2 AES CTS mechanism parameters .....	253
6.13 Additional AES Mechanisms .....	253
6.13.1 Definitions .....	253
6.13.2 AES-GCM Authenticated Encryption / Decryption .....	253
6.13.3 AES-CCM authenticated Encryption / Decryption.....	255
6.13.4 AES-GMAC .....	257
6.13.5 AES GCM and CCM Mechanism parameters.....	257
6.14 AES CMAC .....	260

6.14.1 Definitions .....	260
6.14.2 Mechanism parameters .....	260
6.14.3 General-length AES-CMAC .....	261
6.14.4 AES-CMAC .....	261
6.15 AES XTS .....	261
6.15.1 Definitions .....	262
6.15.2 AES-XTS secret key objects .....	262
6.15.3 AES-XTS key generation .....	262
6.15.4 AES-XTS .....	262
6.16 AES Key Wrap .....	262
6.16.1 Definitions .....	263
6.16.2 AES Key Wrap Mechanism parameters .....	263
6.16.3 AES Key Wrap .....	263
6.17 Key derivation by data encryption – DES & AES .....	264
6.17.1 Definitions .....	264
6.17.2 Mechanism Parameters .....	265
6.17.3 Mechanism Description .....	265
6.18 Double and Triple-length DES .....	265
6.18.1 Definitions .....	266
6.18.2 DES2 secret key objects .....	266
6.18.3 DES3 secret key objects .....	267
6.18.4 Double-length DES key generation .....	267
6.18.5 Triple-length DES Order of Operations .....	268
6.18.6 Triple-length DES in CBC Mode .....	268
6.18.7 DES and Triple length DES in OFB Mode .....	268
6.18.8 DES and Triple length DES in CFB Mode .....	269
6.19 Double and Triple-length DES CMAC .....	269
6.19.1 Definitions .....	269
6.19.2 Mechanism parameters .....	269
6.19.3 General-length DES3-MAC .....	270
6.19.4 DES3-CMAC .....	270
6.20 SHA-1 .....	270
6.20.1 Definitions .....	271
6.20.2 SHA-1 digest .....	271
6.20.3 General-length SHA-1-HMAC .....	271
6.20.4 SHA-1-HMAC .....	272
6.20.5 SHA-1 key derivation .....	272
6.20.6 SHA-1 HMAC key generation .....	272
6.21 SHA-224 .....	273
6.21.1 Definitions .....	273
6.21.2 SHA-224 digest .....	273
6.21.3 General-length SHA-224-HMAC .....	273
6.21.4 SHA-224-HMAC .....	274
6.21.5 SHA-224 key derivation .....	274
6.21.6 SHA-224 HMAC key generation .....	274
6.22 SHA-256 .....	274

6.22.1 Definitions .....	275
6.22.2 SHA-256 digest .....	275
6.22.3 General-length SHA-256-HMAC .....	275
6.22.4 SHA-256-HMAC .....	276
6.22.5 SHA-256 key derivation.....	276
6.22.6 SHA-256 HMAC key generation.....	276
6.23 SHA-384 .....	276
6.23.1 Definitions .....	276
6.23.2 SHA-384 digest .....	277
6.23.3 General-length SHA-384-HMAC .....	277
6.23.4 SHA-384-HMAC .....	277
6.23.5 SHA-384 key derivation.....	277
6.23.6 SHA-384 HMAC key generation.....	278
6.24 SHA-512 .....	278
6.24.1 Definitions .....	278
6.24.2 SHA-512 digest .....	278
6.24.3 General-length SHA-512-HMAC .....	279
6.24.4 SHA-512-HMAC .....	279
6.24.5 SHA-512 key derivation.....	279
6.24.6 SHA-512 HMAC key generation.....	279
6.25 SHA-512/224 .....	280
6.25.1 Definitions .....	280
6.25.2 SHA-512/224 digest .....	280
6.25.3 General-length SHA-512/224-HMAC .....	280
6.25.4 SHA-512/224-HMAC .....	281
6.25.5 SHA-512/224 key derivation.....	281
6.25.6 SHA-512/224 HMAC key generation .....	281
6.26 SHA-512/256 .....	281
6.26.1 Definitions .....	282
6.26.2 SHA-512/256 digest .....	282
6.26.3 General-length SHA-512/256-HMAC .....	282
6.26.4 SHA-512/256-HMAC .....	283
6.26.5 SHA-512/256 key derivation.....	283
6.26.6 SHA-512/256 HMAC key generation .....	283
6.27 SHA-512/t .....	283
6.27.1 Definitions .....	284
6.27.2 SHA-512/t digest .....	284
6.27.3 General-length SHA-512/t-HMAC .....	284
6.27.4 SHA-512/t-HMAC .....	284
6.27.5 SHA-512/t key derivation.....	285
6.27.6 SHA-512/t HMAC key generation .....	285
6.28 SHA3-224 .....	285
6.28.1 Definitions .....	285
6.28.2 SHA3-224 digest .....	286
6.28.3 General-length SHA3-224-HMAC .....	286
6.28.4 SHA3-224-HMAC .....	286

6.28.5 SHA3-224 key derivation.....	286
6.28.6 SHA3-224 HMAC key generation .....	286
6.29 SHA3-256 .....	287
6.29.1 Definitions .....	287
6.29.2 SHA3-256 digest .....	287
6.29.3 General-length SHA3-256-HMAC .....	287
6.29.4 SHA3-256-HMAC .....	288
6.29.5 SHA3-256 key derivation.....	288
6.29.6 SHA3-256 HMAC key generation .....	288
6.30 SHA3-384 .....	288
6.30.1 Definitions .....	289
6.30.2 SHA3-384 digest .....	289
6.30.3 General-length SHA3-384-HMAC .....	289
6.30.4 SHA3-384-HMAC .....	290
6.30.5 SHA3-384 key derivation.....	290
6.30.6 SHA3-384 HMAC key generation .....	290
6.31 SHA3-512 .....	290
6.31.1 Definitions .....	290
6.31.2 SHA3-512 digest .....	291
6.31.3 General-length SHA3-512-HMAC .....	291
6.31.4 SHA3-512-HMAC .....	291
6.31.5 SHA3-512 key derivation.....	291
6.31.6 SHA3-512 HMAC key generation .....	291
6.32 SHAKE.....	292
6.32.1 Definitions .....	292
6.32.2 SHAKE Key Derivation.....	292
6.33 BLAKE2B-160.....	293
6.33.1 Definitions .....	293
6.33.2 BLAKE2B-160 digest.....	293
6.33.3 General-length BLAKE2B-160-HMAC .....	293
6.33.4 BLAKE2B-160-HMAC .....	294
6.33.5 BLAKE2B-160 key derivation.....	294
6.33.6 BLAKE2B-160 HMAC key generation .....	294
6.34 BLAKE2B-256.....	294
6.34.1 Definitions .....	295
6.34.2 BLAKE2B-256 digest.....	295
6.34.3 General-length BLAKE2B-256-HMAC .....	295
6.34.4 BLAKE2B-256-HMAC .....	295
6.34.5 BLAKE2B-256 key derivation.....	295
6.34.6 BLAKE2B-256 HMAC key generation .....	296
6.35 BLAKE2B-384.....	296
6.35.1 Definitions .....	296
6.35.2 BLAKE2B-384 digest.....	296
6.35.3 General-length BLAKE2B-384-HMAC .....	297
6.35.4 BLAKE2B-384-HMAC .....	297
6.35.5 BLAKE2B-384 key derivation .....	297

6.35.6 BLAKE2B-384 HMAC key generation.....	297
6.36 BLAKE2B-512.....	297
6.36.1 Definitions .....	298
6.36.2 BLAKE2B-512 digest.....	298
6.36.3 General-length BLAKE2B-512-HMAC .....	298
6.36.4 BLAKE2B-512-HMAC .....	299
6.36.5 BLAKE2B-512 key derivation.....	299
6.36.6 BLAKE2B-512 HMAC key generation.....	299
6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE).....	299
6.37.1 Definitions .....	300
6.37.2 Password-based encryption/authentication mechanism parameters.....	300
6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters .....	301
6.37.4 PKCS #5 PBKD2 key generation .....	303
6.38 PKCS #12 password-based encryption/authentication mechanisms .....	303
6.38.1 SHA-1-PBE for 3-key triple-DES-CBC .....	304
6.38.2 SHA-1-PBE for 2-key triple-DES-CBC .....	304
6.38.3 SHA-1-PBA for SHA-1-HMAC.....	304
6.39 SSL .....	304
6.39.1 Definitions .....	305
6.39.2 SSL mechanism parameters .....	305
6.39.3 Pre-master key generation .....	307
6.39.4 Master key derivation .....	307
6.39.5 Master key derivation for Diffie-Hellman .....	308
6.39.6 Key and MAC derivation.....	309
6.39.7 MD5 MACing in SSL 3.0 .....	310
6.39.8 SHA-1 MACing in SSL 3.0 .....	310
6.40 TLS 1.2 Mechanisms .....	310
6.40.1 Definitions .....	311
6.40.2 TLS 1.2 mechanism parameters .....	311
6.40.3 TLS MAC .....	314
6.40.4 Master key derivation .....	314
6.40.5 Master key derivation for Diffie-Hellman .....	315
6.40.6 Key and MAC derivation.....	316
6.40.7 CKM_TLS12_KEY_SAFE_DERIVE.....	316
6.40.8 Generic Key Derivation using the TLS PRF .....	317
6.40.9 Generic Key Derivation using the TLS12 PRF .....	317
6.41 WTLS .....	318
6.41.1 Definitions .....	318
6.41.2 WTLS mechanism parameters .....	319
6.41.3 Pre master secret key generation for RSA key exchange suite.....	321
6.41.4 Master secret key derivation .....	322
6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography .....	322
6.41.6 WTLS PRF (pseudorandom function) .....	323
6.41.7 Server Key and MAC derivation .....	323
6.41.8 Client key and MAC derivation .....	324
6.42 SP 800-108 Key Derivation .....	325

6.42.1 Definitions .....	325
6.42.2 Mechanism Parameters .....	326
6.42.3 Counter Mode KDF .....	331
6.42.4 Feedback Mode KDF .....	331
6.42.5 Double Pipeline Mode KDF .....	332
6.42.6 Deriving Additional Keys .....	333
6.42.7 Key Derivation Attribute Rules .....	334
6.42.8 Constructing PRF Input Data .....	334
6.42.8.1 Sample Counter Mode KDF .....	334
6.42.8.2 Sample SCP03 Counter Mode KDF .....	335
6.42.8.3 Sample Feedback Mode KDF .....	336
6.42.8.4 Sample Double-Pipeline Mode KDF .....	338
6.43 Miscellaneous simple key derivation mechanisms .....	339
6.43.1 Definitions .....	339
6.43.2 Parameters for miscellaneous simple key derivation mechanisms .....	339
6.43.3 Concatenation of a base key and another key .....	340
6.43.4 Concatenation of a base key and data .....	340
6.43.5 Concatenation of data and a base key .....	341
6.43.6 XORing of a key and data .....	342
6.43.7 Extraction of one key from another key .....	343
6.44 CMS .....	343
6.44.1 Definitions .....	344
6.44.2 CMS Signature Mechanism Objects .....	344
6.44.3 CMS mechanism parameters .....	344
6.44.4 CMS signatures .....	345
6.45 Blowfish .....	346
6.45.1 Definitions .....	347
6.45.2 BLOWFISH secret key objects .....	347
6.45.3 Blowfish key generation .....	348
6.45.4 Blowfish-CBC .....	348
6.45.5 Blowfish-CBC with PKCS padding .....	348
6.46 Twofish .....	349
6.46.1 Definitions .....	349
6.46.2 Twofish secret key objects .....	349
6.46.3 Twofish key generation .....	350
6.46.4 Twofish -CBC .....	350
6.46.5 Twofish-CBC with PKCS padding .....	350
6.47 CAMELLIA .....	350
6.47.1 Definitions .....	351
6.47.2 Camellia secret key objects .....	351
6.47.3 Camellia key generation .....	352
6.47.4 Camellia-ECB .....	352
6.47.5 Camellia-CBC .....	353
6.47.6 Camellia-CBC with PKCS padding .....	353
6.47.7 CAMELLIA with Counter mechanism parameters .....	354
6.47.8 General-length Camellia-MAC .....	355
6.47.9 Camellia-MAC .....	355

6.48 Key derivation by data encryption - Camellia .....	356
6.48.1 Definitions .....	356
6.48.2 Mechanism Parameters .....	356
6.49 ARIA.....	356
6.49.1 Definitions .....	357
6.49.2 Aria secret key objects .....	357
6.49.3 ARIA key generation .....	358
6.49.4 ARIA-ECB.....	358
6.49.5 ARIA-CBC .....	358
6.49.6 ARIA-CBC with PKCS padding .....	359
6.49.7 General-length ARIA-MAC .....	360
6.49.8 ARIA-MAC .....	360
6.50 Key derivation by data encryption - ARIA.....	360
6.50.1 Definitions .....	361
6.50.2 Mechanism Parameters .....	361
6.51 SEED .....	361
6.51.1 Definitions .....	362
6.51.2 SEED secret key objects.....	362
6.51.3 SEED key generation .....	363
6.51.4 SEED-ECB .....	363
6.51.5 SEED-CBC .....	363
6.51.6 SEED-CBC with PKCS padding.....	363
6.51.7 General-length SEED-MAC.....	364
6.51.8 SEED-MAC.....	364
6.52 Key derivation by data encryption - SEED .....	364
6.52.1 Definitions .....	364
6.52.2 Mechanism Parameters .....	364
6.53 OTP.....	365
6.53.1 Usage overview.....	365
6.53.2 Case 1: Generation of OTP values .....	365
6.53.3 Case 2: Verification of provided OTP values .....	366
6.53.4 Case 3: Generation of OTP keys .....	366
6.53.5 OTP objects.....	367
6.53.5.1 Key objects .....	367
6.53.6 OTP-related notifications .....	370
6.53.7 OTP mechanisms .....	370
6.53.7.1 OTP mechanism parameters .....	370
6.53.8 RSA SecurID .....	374
6.53.8.1 RSA SecurID secret key objects .....	374
6.53.8.2 RSA SecurID key generation .....	375
6.53.8.3 SecurID OTP generation and validation.....	375
6.53.8.4 Return values.....	376
6.53.9 OATH HOTP.....	376
6.53.9.1 OATH HOTP secret key objects .....	376
6.53.9.2 HOTP key generation .....	377
6.53.9.3 HOTP OTP generation and validation.....	377
6.53.10 ActivIdentity ACTI .....	377

6.53.10.1 ACTI secret key objects .....	377
6.53.10.2 ACTI key generation .....	378
6.53.10.3 ACTI OTP generation and validation .....	378
6.54 CT-KIP .....	379
6.54.1 Principles of Operation .....	379
6.54.2 Mechanisms .....	379
6.54.3 Definitions .....	380
6.54.4 CT-KIP Mechanism parameters .....	380
6.54.5 CT-KIP key derivation .....	380
6.54.6 CT-KIP key wrap and key unwrap .....	381
6.54.7 CT-KIP signature generation .....	381
6.55 GOST 28147-89 .....	381
6.55.1 Definitions .....	381
6.55.2 GOST 28147-89 secret key objects .....	382
6.55.3 GOST 28147-89 domain parameter objects .....	382
6.55.4 GOST 28147-89 key generation .....	383
6.55.5 GOST 28147-89-ECB .....	384
6.55.6 GOST 28147-89 encryption mode except ECB .....	384
6.55.7 GOST 28147-89-MAC .....	385
6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 .....	385
6.56 GOST R 34.11-94 .....	386
6.56.1 Definitions .....	386
6.56.2 GOST R 34.11-94 domain parameter objects .....	386
6.56.3 GOST R 34.11-94 digest .....	387
6.56.4 GOST R 34.11-94 HMAC .....	388
6.57 GOST R 34.10-2001 .....	388
6.57.1 Definitions .....	388
6.57.2 GOST R 34.10-2001 public key objects .....	389
6.57.3 GOST R 34.10-2001 private key objects .....	390
6.57.4 GOST R 34.10-2001 domain parameter objects .....	392
6.57.5 GOST R 34.10-2001 mechanism parameters .....	393
6.57.6 GOST R 34.10-2001 key pair generation .....	394
6.57.7 GOST R 34.10-2001 without hashing .....	394
6.57.8 GOST R 34.10-2001 with GOST R 34.11-94 .....	395
6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001 .....	395
6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys .....	395
6.58 ChaCha20 .....	396
6.58.1 Definitions .....	396
6.58.2 ChaCha20 secret key objects .....	396
6.58.3 ChaCha20 mechanism parameters .....	397
6.58.4 ChaCha20 key generation .....	397
6.58.5 ChaCha20 mechanism .....	397
6.59 Salsa20 .....	398
6.59.1 Definitions .....	399
6.59.2 Salsa20 secret key objects .....	399
6.59.3 Salsa20 mechanism parameters .....	400

6.59.4 Salsa20 key generation .....	400
6.59.5 Salsa20 mechanism .....	400
6.60 Poly1305 .....	401
6.60.1 Definitions .....	401
6.60.2 Poly1305 secret key objects .....	401
6.60.3 Poly1305 mechanism .....	402
6.61 ChaCha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption .....	402
6.61.1 Definitions .....	403
6.61.2 Usage .....	403
6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters .....	404
6.62 HKDF Mechanisms .....	405
6.62.1 Definitions .....	406
6.62.2 HKDF mechanism parameters .....	406
6.62.3 HKDF derive .....	407
6.62.4 HKDF Data .....	407
6.62.5 HKDF Key gen .....	407
6.63 NULL Mechanism .....	407
6.63.1 Definitions .....	408
6.63.2 CKM_NULL mechanism parameters .....	408
6.64 IKE Mechanisms .....	408
6.64.1 Definitions .....	408
6.64.2 IKE mechanism parameters .....	409
6.64.3 IKE PRF DERIVE .....	411
6.64.4 IKEv1 PRF DERIVE .....	412
6.64.5 IKEv2 PRF PLUS DERIVE .....	412
6.64.6 IKEv1 Extended Derive .....	413
6.65 HSS .....	413
6.65.1 Definitions .....	414
6.65.2 HSS public key objects .....	414
6.65.3 HSS private key objects .....	415
6.65.4 HSS key pair generation .....	416
6.65.5 HSS without hashing .....	416
7 PKCS #11 Implementation Conformance .....	418
7.1 PKCS#11 Consumer Implementation Conformance .....	418
7.2 PKCS#11 Provider Implementation Conformance .....	418
Appendix A. Acknowledgments .....	419
Appendix B. Manifest constants .....	421
Appendix C. Revision History .....	422
Appendix D. Notices .....	424

---

# 1 Introduction

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

The PKCS#11 standard specifies an application programming interface (API), called "Cryptoki," for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a "cryptographic token".

This document specifies the data types and functions available to an application requiring cryptographic services using the ANSI C programming language. The supplier of a Cryptoki library implementation typically provides these data types and functions via ANSI C header files. Generic ANSI C header files for Cryptoki are available from the PKCS#11 web page. This document and up-to-date errata for Cryptoki will also be available from the same place.

Additional documents may provide a generic, language-independent Cryptoki interface and/or bindings between Cryptoki and other programming languages.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable. How Cryptoki provides this isolation is beyond the scope of this document, although some conventions for the support of multiple types of device will be addressed here and possibly in a separate document.

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

## 1.1 Definitions

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

24	<b>AES</b>	Advanced Encryption Standard, as defined in FIPS PUB 197.
25	<b>API</b>	Application programming interface.
26	<b>Application</b>	Any computer program that calls the Cryptoki interface.
27	<b>ASN.1</b>	Abstract Syntax Notation One, as defined in X.680.
28	<b>Attribute</b>	A characteristic of an object.
29	<b>BER</b>	Basic Encoding Rules, as defined in X.690.
30	<b>BLOWFISH</b>	The Blowfish Encryption Algorithm of Bruce Schneier, <a href="http://www.schneier.com">www.schneier.com</a> .
32	<b>CAMELLIA</b>	The Camellia encryption algorithm, as defined in RFC 3713.
33	<b>CBC</b>	Cipher-Block Chaining mode, as defined in FIPS PUB 81.
34	<b>Certificate</b>	A signed message binding a subject name and a public key, or a subject name and a set of attributes.
36	<b>CDMF</b>	Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
39	<b>CMAC</b>	Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].
41	<b>CMS</b>	Cryptographic Message Syntax (see RFC 5652)

	<b>Cryptographic Device</b>	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.
42	<b>Cryptoki</b>	The Cryptographic Token Interface defined in this standard.
43	<b>Cryptoki library</b>	A library that implements the functions specified in this standard.
44	<b>CT-KIP</b>	Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])
45		
46	<b>DER</b>	Distinguished Encoding Rules, as defined in X.690.
47	<b>DES</b>	Data Encryption Standard, as defined in FIPS PUB 46-3.
48	<b>DSA</b>	Digital Signature Algorithm, as defined in FIPS PUB 186-4.
49		
50	<b>EC</b>	Elliptic Curve
51	<b>ECB</b>	Electronic Codebook mode, as defined in FIPS PUB 81.
52	<b>ECDH</b>	Elliptic Curve Diffie-Hellman.
53		
54	<b>ECDSA</b>	Elliptic Curve DSA, as in ANSI X9.62.
55	<b>ECMQV</b>	Elliptic Curve Menezes-Qu-Vanstone
56	<b>GOST 28147-89</b>	The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].
57	<b>GOST R 34.11-94</b>	Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].
58		
59	<b>GOST R 34.10-2001</b>	The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].
60		
61	<b>IV</b>	Initialization Vector.
62		
63	<b>MAC</b>	Message Authentication Code.
64		
65	<b>Mechanism</b>	A process for implementing a cryptographic operation.
66		
67	<b>MQV</b>	Menezes-Qu-Vanstone
68		
69	<b>OAEP</b>	Optimal Asymmetric Encryption Padding for RSA.
70		
71	<b>Object</b>	An item that is stored on a token. May be data, a certificate, or a key.
72		
73	<b>PIN</b>	Personal Identification Number.
74		
75	<b>PKCS</b>	Public-Key Cryptography Standards.
76		
77	<b>PRF</b>	Pseudo random function.
78		
79	<b>PTD</b>	Personal Trusted Device, as defined in MeT-PTD
80		
81	<b>RSA</b>	The RSA public-key cryptosystem.
82		
82	<b>Reader</b>	The means by which information is exchanged with a device.
82		
82	<b>Session</b>	A logical connection between an application and a token.
82		
82	<b>SHA-1</b>	The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.
82		
82	<b>SHA-224</b>	The Secure Hash Algorithm with a 224-bit message digest, as defined in RFC 3874. Also defined in FIPS PUB 180-2 with Change Notice 1.

83	<b>SHA-256</b>	The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.
84		
85	<b>SHA-384</b>	The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.
86		
87	<b>SHA-512</b>	The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.
88		
89	<b>Slot</b>	A logical reader that potentially contains a token.
90	<b>SSL</b>	The Secure Sockets Layer 3.0 protocol.
91	<b>Subject Name</b>	The X.500 distinguished name of the entity to which a key is assigned.
92		
93	<b>SO</b>	A Security Officer user.
94	<b>TLS</b>	Transport Layer Security.
95	<b>Token</b>	The logical view of a cryptographic device defined by Cryptoki.
96	<b>User</b>	The person using an application that interfaces to Cryptoki.
97	<b>UTF-8</b>	Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.
98		
99		
100	<b>WTLS</b>	Wireless Transport Layer Security.

## 1.2 Symbols and abbreviations

102 The following symbols are used in this standard:

103 *Table 1, Symbols*

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

104 The following prefixes are used in this standard:

105 *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

<b>Prefix</b>	<b>Description</b>
CKP_	Pseudo-random function
CKS_	Session state
CKR_	Return value
CKU_	User type
CKZ_	Salt/Encoding parameter source
h	a handle
ul	a CK ULONG
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pul	a pointer to a CK ULONG

106

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

108

```

109  /* an unsigned 8-bit value */
110  typedef unsigned char CK_BYTE;
111
112  /* an unsigned 8-bit character */
113  typedef CK_BYTE CK_CHAR;
114
115  /* an 8-bit UTF-8 character */
116  typedef CK_BYTE CK_UTF8CHAR;
117
118  /* a BYTE-sized Boolean flag */
119  typedef CK_BYTE CK_BBOOL;
120
121  /* an unsigned value, at least 32 bits long */
122  typedef unsigned long int CK ULONG;
123
124  /* a signed value, the same size as a CK ULONG */
125  typedef long int CK_LONG;
126
127  /* at least 32 bits; each bit is a Boolean flag */
128  typedef CK ULONG CK_FLAGS;
129

```

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

```

132  CK_BYTE_PTR      /* Pointer to a CK_BYTE */
133  CK_CHAR_PTR      /* Pointer to a CK_CHAR */
134  CK_UTF8CHAR_PTR  /* Pointer to a CK_UTF8CHAR */
135  CK ULONG_PTR     /* Pointer to a CK ULONG */
136  CK VOID_PTR      /* Pointer to a void */
137

```

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

```

139  CK VOID_PTR_PTR  /* Pointer to a CK VOID_PTR */
140

```

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

```

142  NULL_PTR         /* A NULL pointer */
143

```

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

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

150 The **CK\_CHAR** data type holds characters from the following table, taken from ANSI C:

151 *Table 3, Character Set*

Category	Characters
Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Numbers	0 1 2 3 4 5 6 7 8 9
Graphic characters	! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ {   } ~
Blank character	" "

152 The **CK\_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-  
153 8 allows internationalization while maintaining backward compatibility with the Local String definition of  
154 PKCS #11 version 2.01.

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

```
159 #define CK_FALSE 0
160 #define CK_TRUE 1
```

162 For backwards compatibility, header files for this version of Cryptoki also define TRUE and FALSE as  
163 (**CK\_DISABLE\_TRUE\_FALSE** may be set by the application vendor):

```
164 #ifndef CK_DISABLE_TRUE_FALSE
165 #ifndef FALSE
166 #define FALSE CK_FALSE
167 #endif
168
169 #ifndef TRUE
170 #define TRUE CK_TRUE
171 #endif
172 #endif
```

## 174 1.3 Normative References

175 **[ARIA]** National Security Research Institute, Korea, "Block Cipher Algorithm ARIA",  
176 URL: <https://www.ietf.org/rfc/rfc5794.txt>

177 **[BLOWFISH]** B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish),  
178 December 1993.  
179 URL: <https://www.schneier.com/paper-blowfish-fse.html>

180 **[CAMELLIA]** M. Matsui, J. Nakajima, S. Moriai. A Description of the Camellia Encryption Algorithm,  
181 April 2004.  
182 URL: <http://www.ietf.org/rfc/rfc3713.txt>

183 **[CDMF]** Johnson, D.B. The Commercial Data Masking Facility (CDMF) data privacy algorithm, March  
184 1994.  
185 URL: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5389557>

- 186    [CHACHA]    D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.  
 187    URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
- 188    [DH]    W. Diffie, M. Hellman. New Directions in Cryptography. Nov, 1976.  
 189    URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>
- 190    [FIPS PUB 46-3]    NIST. *FIPS 46-3: Data Encryption Standard*. October 1999.  
 191    URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- 192    [FIPS PUB 81]    NIST. *FIPS 81: DES Modes of Operation*. December 1980.  
 193    URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>
- 194    [FIPS PUB 186-4]    NIST. FIPS 186-4: Digital Signature Standard. July, 2013.  
 195    URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>
- 196    [FIPS SP 800-56A]    NIST. Special Publication 800-56A Revision 2: *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, May 2013.  
 197    URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>
- 199    [FIPS SP 800-108]    NIST. Special Publication 800-108 (Revised): *Recommendation for Key Derivation Using Pseudorandom Functions*, October 2009.  
 200    URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>
- 202    [GOST] V. Dolmatov, A. Degtyarev. GOST R. 34.11-2012: Hash Function. August 2013.  
 203    URL: <https://tools.ietf.org/html/rfc6986>
- 204    [MD2] B. Kaliski. RSA Laboratories. The MD2 Message-Digest Algorithm. April, 1992.  
 205    URL: <https://www.ietf.org/rfc/rfc1319.txt>
- 206    [MD5] RSA Data Security. R. Rivest. The MD5 Message-Digest Algorithm. April, 1992.  
 207    URL: <https://www.ietf.org/rfc/rfc1321.txt>
- 208    [NIST 802-208] NIST Special Publication 800-208: *Recommendation for Stateful Hash-Based Signature Schemes*, October 2020.  
 209    URL: <https://csrc.nist.gov/publications/detail/sp/800-208/final>
- 211    [OAEP] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with RSA. Nov 19, 1995.
- 213    [PKCS11-Hist] PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.1. Work in progress. Latest stage: <URL>
- 215    [PKCS11-Prof] PKCS #11 Profiles Version 3.1. Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html>.
- 217    [PKCS #1] RSA Laboratories. RSA Cryptography Standard. v2.1, June 14, 2002.  
 218    URL: <https://tools.ietf.org/html/rfc8017>
- 219    [PKCS #3] RSA Laboratories. Diffie-Hellman Key-Agreement Standard. v1.4, November 1993.  
 220    URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-3.doc>
- 221    [PKCS #5] RSA Laboratories. Password-Based Encryption Standard. v2.0, March 25, 1999  
 222    URL: <https://tools.ietf.org/html/rfc8018>
- 223    [PKCS #7] RSA Laboratories. Cryptographic Message Syntax Standard. v1.5, November 1993  
 224    URL: <https://tools.ietf.org/html/rfc2315>
- 225    [PKCS #8] RSA Laboratories. Private-Key Information Syntax Standard. v1.2, November 1993.  
 226    URL: <https://tools.ietf.org/html/rfc5958>
- 227    [PKCS #12] RSA Laboratories. Personal Information Exchange Syntax Standard. v1.0, June 1999.  
 228    URL: <https://tools.ietf.org/html/rfc7292>
- 229    [POLY1305]    D.J. Bernstein. The Poly1305-AES message-authentication code. Jan 2005.  
 230    URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>

- 231 [RFC 2409] D. Harkins, D.Carrel. RFC 2409: *The Internet Key Exchange (IKE)*, November 1998.  
232 URL: <https://tools.ietf.org/html/rfc2409>
- 233 [RFC 2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC  
234 2119, March 1997.  
235 URL: <http://www.ietf.org/rfc/rfc2119.txt>.
- 236 [RFC 2279] F. Yergeau. RFC 2279: UTF-8, a transformation format of ISO 10646 Alis Technologies,  
237 January 1998.  
238 URL: <http://www.ietf.org/rfc/rfc2279.txt>
- 239 [RFC 2534] Masinter, L., Wing, D., Mutz, A., and K. Holtman. RFC 2534: *Media Features for Display, Print, and Fax*. March 1999.  
240 URL: <http://www.ietf.org/rfc/rfc2534.txt>
- 242 [RFC 5652] R. Housley. RFC 5652: *Cryptographic Message Syntax*. Septmber 2009. URL:  
243 <http://www.ietf.org/rfc/rfc5652.txt>
- 244 [RFC 5707] Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC  
245 5705, March 2010.  
246 URL: <http://www.ietf.org/rfc/rfc5705.txt>
- 247 [RFC 5996] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen. RFC 5996: *Internet Key Exchange Protocol Version 2 (IKEv2)*, September 2010.  
248 URL: <https://tools.ietf.org/html/rfc5996>
- 250 [RFC 8554] D. McGrew, m. Curcio, S. Fluhrer. RFC 8554: *Leighton-Micali Hash-Based Signatures*,  
251 April 2019.  
252 URL: <https://tools.ietf.org/html/rfc8554>
- 253 [RIPEMD] H. Dobbertin, A. Bosselaers, B. Preneel. The hash function RIPEMD-160, Feb 13, 2012.  
254 URL: <http://homes.esat.kuleuven.be/~bosselaer/ripemd160.html>
- 255 [SALSA] D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.  
256 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
- 257 [SEED] KISA. SEED 128 Algorithm Specification. Sep 2003.  
258 URL: [http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D\\_SEED+128\\_Specification\\_english\\_M.pdf](http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D_SEED+128_Specification_english_M.pdf)
- 260 [SHA-1] NIST. FIPS 180-4: Secure Hash Standard. March 2012.  
261 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>
- 262 [SHA-2] NIST. FIPS 180-4: Secure Hash Standard. March 2012.  
263 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>
- 264 [TLS] [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.  
265 URL: <http://www.ietf.org/rfc/rfc2246.txt>, superseded by [RFC4346] Dierks, T. and E. Rescorla, "The  
266 Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006. URL:  
267 <http://www.ietf.org/rfc/rfc4346.txt>, which was superseded by [TLS12].
- 268 [TLS12] [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol  
269 Version 1.2", RFC 5246, August 2008.  
270 URL: <http://www.ietf.org/rfc/rfc5246.txt>
- 271 [TWOFISH] B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. Twofish: A 128-Bit Block  
272 Cipher. June 15, 1998.  
273 URL: <https://www.schneier.com/academic/twofish/>
- 274 [X.500] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of  
275 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
- 276 [X.509] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key  
277 and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8

278 [X.680] ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic  
279 Notation. July 2002. Identical to ISO/IEC 8824-1

280 [X.690] ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules  
281 (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical  
282 to ISO/IEC 8825-1

283

## 284 **1.4 Non-Normative References**

285 [CAP-1.2] Common Alerting Protocol Version 1.2. 01 July 2010. OASIS Standard.  
286 URL: <http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html>

287 [AES KEYWRAP] National Institute of Standards and Technology, NIST Special Publication 800-  
288 38F, Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, December  
289 2012, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf>

290 [ANSI C] ANSI/ISO. American National Standard for Programming Languages – C. 1990.

291 [ANSI X9.31] Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key  
292 Cryptography for the Financial Services Industry (rDSA). 1998.

293 [ANSI X9.42] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services  
294 Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.

295 [ANSI X9.62] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services  
296 Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.

297 [ANSI X9.63] Accredited Standards Committee X9. Public Key Cryptography for the Financial Services  
298 Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001.  
299 URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011>

300 [BRAINPOOL] ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005  
301 URL: <http://www.ecc-brainpool.org>

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

305 [CDPD] Ameritech Mobile Communications et al. Cellular Digital Packet Data System Specifications: Part  
306 406: Airlink Security. 1993.

307 [CT-KIP] RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0,  
308 December 2005.

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

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

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

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

317 [ISO/IEC 8825-1] ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic  
318 Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).  
319 2002.

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

- 322 [ISO/IEC 9594-8] ISO. Information Technology — Open Systems Interconnection — The Directory:  
323 Public-key and Attribute Certificate Frameworks. 2001
- 324 [ISO/IEC 9796-2] ISO. Information Technology — Security Techniques — Digital Signature  
325 Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.
- 326 [Java MIDP] Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition.  
327 November 2002.  
328 URL: <http://jcp.org/jsr/detail/118.jsp>
- 329 [LEGIFRANCE] Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of  
330 Elliptic Curve parameters by the French state)  
331 URL: <https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816>
- 332 [MeT-PTD] MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February  
333 2003.  
334 URL: <http://www.mobiletransaction.org>
- 335 [NIST AES CTS] National Institute of Standards and Technology, Addendum to NIST Special  
336 Publication 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of  
337 Ciphertext Stealing for CBC Mode”  
338 URL: [http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist\\_sp800-38A.pdf](http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf)
- 339 [PCMCIA] Personal Computer Memory Card International Association. *PC Card Standard*,  
340 Release 2.1., July 1993.
- 341 [RFC 2865] Rigney et al, “Remote Authentication Dial In User Service (RADIUS)”, IETF RFC2865,  
342 June 2000.  
343 URL: <http://www.ietf.org/rfc/rfc2865.txt>.
- 344 [RFC 3686] Housley, “Using Advanced Encryption Standard (AES) Counter Mode With IPsec  
345 Encapsulating Security Payload (ESP),” IETF RFC 3686, January 2004.  
346 URL: <http://www.ietf.org/rfc/rfc3686.txt>.
- 347 [RFC 3717] Matsui, et al, "A Description of the Camellia Encryption Algorithm," IETF RFC 3717, April  
348 2004.  
349 URL: <http://www.ietf.org/rfc/rfc3713.txt>.
- 350 [RFC 3610] Whiting, D., Housley, R., and N. Ferguson, “Counter with CBC-MAC (CCM)”, IETF RFC  
351 3610, September 2003.  
352 URL: <http://www.ietf.org/rfc/rfc3610.txt>
- 353 [RFC 3874] Smit et al, “A 224-bit One-way Hash Function: SHA-224,” IETF RFC 3874, June 2004.  
354 URL: <http://www.ietf.org/rfc/rfc3874.txt>.
- 355 [RFC 3748] Aboba et al, “Extensible Authentication Protocol (EAP)”, IETF RFC 3748, June 2004.  
356 URL: <http://www.ietf.org/rfc/rfc3748.txt>.
- 357 [RFC 4269] South Korean Information Security Agency (KISA) “The SEED Encryption Algorithm”,  
358 December 2005.  
359 URL: <https://ftp.rfc-editor.org/in-notes/rfc4269.txt>
- 360 [RFC 4309] Housley, R., “Using Advanced Encryption Standard (AES) CCM Mode with IPsec  
361 Encapsulating Security Payload (ESP),” IETF RFC 4309, December 2005.  
362 URL: <http://www.ietf.org/rfc/rfc4309.txt>
- 363 [RFC 4357] V. Popov, I. Kurepkin, S. Leontiev “Additional Cryptographic Algorithms for Use with  
364 GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms”, January  
365 2006.  
366 URL: <http://www.ietf.org/rfc/rfc4357.txt>
- 367 [RFC 4490] S. Leontiev, Ed. G. Chudov, Ed. “Using the GOST 28147-89, GOST R 34.11-94, GOST  
368 R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)”, May 2006.  
369 URL: <http://www.ietf.org/rfc/rfc4490.txt>

- 370    [RFC 4491]    S. Leontiev, Ed., D. Shefanovski, Ed., "Using the GOST R 34.10-94, GOST R 34.10-  
371    2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and  
372    CRL Profile", May 2006.  
373    URL: <http://www.ietf.org/rfc/rfc4491.txt>
- 374    [RFC 4493]    J. Song et al. *RFC 4493: The AES-CMAC Algorithm*. June 2006.  
375    URL: <http://www.ietf.org/rfc/rfc4493.txt>
- 376    [RFC 5705]    Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC  
377    5705, March 2010.  
378    URL: <http://www.ietf.org/rfc/rfc5705.txt>
- 379    [RFC 5869]    H. Krawczyk, P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function  
380    (HKDF)", May 2010  
381    URL: <http://www.ietf.org/rfc/rfc5869.txt>
- 382    [RFC 7539]    Y Nir, A. Langley. *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols*, May 2015  
383    URL: <https://tools.ietf.org/rfc/rfc7539.txt>
- 384    [RFC 7748]    Aboba et al, "Elliptic Curves for Security", IETF RFC 7748, January 2016  
385    URL: <https://tools.ietf.org/html/rfc7748>
- 386    [RFC 8032]    Aboba et al, "Edwards-Curve Digital Signature Algorithm (EdDSA)", IETF RFC 8032,  
387    January 2017  
388    URL: <https://tools.ietf.org/html/rfc8032>
- 389    [SEC 1] Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC)*  
390    1: Elliptic Curve Cryptography. Version 1.0, September 20, 2000.
- 391    [SEC 2] Standards for Efficient Cryptography Group (SECG). Standards for Efficient Cryptography (SEC)  
392    2: Recommended Elliptic Curve Domain Parameters. Version 1.0, September 20, 2000.
- 393    [WTLS] WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.  
394    URL: <http://openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf>
- 395    [XEDDSA]    The XEdDSA and VXEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin  
396    (editor)  
397    URL: <https://signal.org/docs/specifications/xeddsa/>
- 398    [X.500] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of  
399    Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
- 400    [X.509] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key  
401    and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
- 402    [X.680] ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic  
403    Notation. July 2002. Identical to ISO/IEC 8824-1
- 404    [X.690] ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules  
405    (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical  
406    to ISO/IEC 8825-1

---

## 407    2 Platform- and compiler-dependent directives for C 408       or C++

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

413 This means that when writing C or C++ code, certain preprocessor directives MUST be issued before  
414 including a Cryptoki header file. These directives are described in the remainder of this section.

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

### 416    2.1 Structure packing

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

### 419    2.2 Pointer-related macros

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

#### 422    ◆ CK\_PTR

423 CK\_PTR is the “indirection string” a given platform and compiler uses to make a pointer to an object. It is  
424 used in the following fashion:

```
425    typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

#### 426    ◆ CK\_DECLARE\_FUNCTION

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

```
430    CK_DECLARE_FUNCTION(CK_RV, C_Initialize) (  
431       CK_VOID_PTR pReserved  
432    );
```

#### 433    ◆ CK\_DECLARE\_FUNCTION\_POINTER

434 CK\_DECLARE\_FUNCTION\_POINTER(returnType, name), when followed by a  
435 parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to  
436 a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its  
437 name. It SHALL be used in either of the following fashions to define a function pointer variable,  
438 myC\_Initialize, which can point to a C\_Initialize function in a Cryptoki library (note that neither of the  
439 following code snippets actually assigns a value to myC\_Initialize):

```
440    CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_Initialize) (  
441       CK_VOID_PTR pReserved  
442    );
```

444 or:

```
445    typedef CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_InitializeType) (
```

```
446     CK_VOID_PTR pReserved  
447 ) ;  
448 myC_InitializeType myC_Initialize;
```

#### 449 ◆ **CK\_CALLBACK\_FUNCTION**

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

```
457     CK_CALLBACK_FUNCTION(CK_RV, myCallback) (args) ;  
458
```

459 or:

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

#### 462 ◆ **NULL\_PTR**

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

---

## 465    3 General data types

466    The general Cryptoki data types are described in the following subsections. The data types for holding  
467    parameters for various mechanisms, and the pointers to those parameters, are not described here; these  
468    types are described with the information on the mechanisms themselves, in Section 6.

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

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

### 476    3.1 General information

477    Cryptoki represents general information with the following types:

#### 478    ♦ CK\_VERSION; CK\_VERSION\_PTR

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

```
482        typedef struct CK_VERSION {  
483            CK_BYTE major;  
484            CK_BYTE minor;  
485        } CK_VERSION;
```

487    The fields of the structure have the following meanings:

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

489        *minor* minor version number (the hundredths portion of the version)

490    Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10. Table 4  
491    below lists the major and minor version values for the officially published Cryptoki specifications.

492    *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

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

495    **CK\_VERSION\_PTR** is a pointer to a **CK\_VERSION**.

#### 496    ♦ CK\_INFO; CK\_INFO\_PTR

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

```
498     typedef struct CK_INFO {
499         CK_VERSION cryptokiVersion;
500         CK_UTF8CHAR manufacturerID[32];
501         CK_FLAGS flags;
502         CK_UTF8CHAR libraryDescription[32];
503         CK_VERSION libraryVersion;
504     } CK_INFO;
```

506 The fields of the structure have the following meanings:

507	<i>cryptokiVersion</i>	Cryptoki interface version number, for compatibility with future revisions of this interface
509	<i>manufacturerID</i>	ID of the Cryptoki library manufacturer. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
511	<i>flags</i>	bit flags reserved for future versions. MUST be zero for this version
512	<i>libraryDescription</i>	character-string description of the library. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
514	<i>libraryVersion</i>	Cryptoki library version number

515 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.

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

## 518 ♦ CK\_NOTIFICATION

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

```
521     typedef CK ULONG CK_NOTIFICATION;
```

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

```
524     CKN_SURRENDER
```

526 The notifications have the following meanings:

527	<i>CKN_SURRENDER</i>	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 5.21.1).
-----	----------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 532 3.2 Slot and token types

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

## 534 ♦ CK\_SLOT\_ID; CK\_SLOT\_ID\_PTR

535 **CK\_SLOT\_ID** is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

```
536     typedef CK ULONG CK_SLOT_ID;
```

538 A list of **CK\_SLOT\_ID**s is returned by **C\_GetSlotList**. A priori, any value of **CK\_SLOT\_ID** can be a valid slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a slot, however.

541 **CK\_SLOT\_ID\_PTR** is a pointer to a **CK\_SLOT\_ID**.

542 ◆ **CK\_SLOT\_INFO; CK\_SLOT\_INFO\_PTR**

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

```
544     typedef struct CK_SLOT_INFO {  
545         CK_UTF8CHAR slotDescription[64];  
546         CK_UTF8CHAR manufacturerID[32];  
547         CK_FLAGS flags;  
548         CK_VERSION hardwareVersion;  
549         CK_VERSION firmwareVersion;  
550     } CK_SLOT_INFO;
```

552 The fields of the structure have the following meanings:

553           *slotDescription* character-string description of the slot. MUST be padded with the  
554            blank character (' '). MUST NOT be null-terminated.

555           *manufacturerID* ID of the slot manufacturer. MUST be padded with the blank  
556            character (' '). MUST NOT be null-terminated.

557           *flags* bits flags that provide capabilities of the slot. The flags are defined  
558            below

559           *hardwareVersion* version number of the slot's hardware

560           *firmwareVersion* version number of the slot's firmware

561 The following table defines the *flags* field:

562 *Table 5, Slot Information Flags*

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x00000001	True if a token is present in the slot (e.g., a device is in the reader)
CKF_REMOVABLE_DEVICE	0x00000002	True if the reader supports removable devices
CKF_HW_SLOT	0x00000004	True if the slot is a hardware slot, as opposed to a software slot implementing a "soft token"

563 For a given slot, the value of the **CKF\_REMOVABLE\_DEVICE** flag *never changes*. In addition, if this flag  
564 is not set for a given slot, then the **CKF\_TOKEN\_PRESENT** flag for that slot is *always* set. That is, if a  
565 slot does not support a removable device, then that slot always has a token in it.

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

567 ◆ **CK\_TOKEN\_INFO; CK\_TOKEN\_INFO\_PTR**

568 **CK\_TOKEN\_INFO** provides information about a token. It is defined as follows:

```
569     typedef struct CK_TOKEN_INFO {  
570         CK_UTF8CHAR label[32];  
571         CK_UTF8CHAR manufacturerID[32];  
572         CK_UTF8CHAR model[16];  
573         CK_CHAR serialNumber[16];  
574         CK_FLAGS flags;  
575         CK ULONG ulMaxSessionCount;  
576         CK ULONG ulSessionCount;
```

```

577     CK ULONG ulMaxRwSessionCount;
578     CK ULONG ulRwSessionCount;
579     CK ULONG ulMaxPinLen;
580     CK ULONG ulMinPinLen;
581     CK ULONG ulTotalPublicMemory;
582     CK ULONG ulFreePublicMemory;
583     CK ULONG ulTotalPrivateMemory;
584     CK ULONG ulFreePrivateMemory;
585     CK VERSION hardwareVersion;
586     CK VERSION firmwareVersion;
587     CK CHAR utcTime[16];
588 } CK_TOKEN_INFO;
589

```

590 The fields of the structure have the following meanings:

591	<i>label</i>	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
592		
593		
594	<i>manufacturerID</i>	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
595		
596	<i>model</i>	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
597		
598	<i>serialNumber</i>	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
599		
600	<i>flags</i>	bit flags indicating capabilities and status of the device as defined below
601		
602	<i>ulMaxSessionCount</i>	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)
603		
604		
605	<i>ulSessionCount</i>	number of sessions that this application currently has open with the token (see <b>CK_TOKEN_INFO Note</b> below)
606		
607	<i>ulMaxRwSessionCount</i>	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)
608		
609		
610	<i>ulRwSessionCount</i>	number of read/write sessions that this application currently has open with the token (see <b>CK_TOKEN_INFO Note</b> below)
611		
612	<i>ulMaxPinLen</i>	maximum length in bytes of the PIN
613	<i>ulMinPinLen</i>	minimum length in bytes of the PIN
614	<i>ulTotalPublicMemory</i>	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)
615		
616	<i>ulFreePublicMemory</i>	the amount of free (unused) memory on the token in bytes for public objects (see <b>CK_TOKEN_INFO Note</b> below)
617		
618	<i>ulTotalPrivateMemory</i>	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)
619		
620	<i>ulFreePrivateMemory</i>	the amount of free (unused) memory on the token in bytes for private objects (see <b>CK_TOKEN_INFO Note</b> below)
621		
622	<i>hardwareVersion</i>	version number of hardware
623	<i>firmwareVersion</i>	version number of firmware

624                    *utcTime*      current time as a character-string of length 16, represented in the  
 625                    format YYYYMMDDhhmmssxx (4 characters for the year; 2  
 626                    characters each for the month, the day, the hour, the minute, and  
 627                    the second; and 2 additional reserved '0' characters). The value of  
 628                    this field only makes sense for tokens equipped with a clock, as  
 629                    indicated in the token information flags (see below)

630         The following table defines the *flags* field:

631         *Table 6, Token Information Flags*

<b>Bit Flag</b>	<b>Mask</b>	<b>Meaning</b>
CKF_RNG	0x00000001	True if the token has its own random number generator
CKF_WRITE_PROTECTED	0x00000002	True if the token is write-protected (see below)
CKF_LOGIN_REQUIRED	0x00000004	True if there are some cryptographic functions that a user MUST be logged in to perform
CKF_USER_PIN_INITIALIZED	0x00000008	True if the normal user's PIN has been initialized
CKF_RESTORE_KEY_NOT_NEEDED	0x00000020	True if a successful save of a session's cryptographic operations state <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_PATH	0x00000100	True if token has a "protected authentication path", whereby a user can log into the token without passing a PIN through the Cryptoki library
CKF_DUAL_CRYPTO_OPERATIONS	0x00000200	True if a single session with the token can perform dual cryptographic operations (see Section 5.14)
CKF_TOKEN_INITIALIZED	0x00000400	True if the token has been initialized using C_InitToken or an equivalent mechanism outside the scope of this standard. Calling C_InitToken when this flag is set will cause the token to be reinitialized.
CKF_SECONDARY_AUTHENTICATION	0x00000800	True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag)
CKF_USER_PIN_COUNT_LOW	0x00010000	True if an incorrect user login PIN has been entered at least once since the last successful authentication.

Bit Flag	Mask	Meaning
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.

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

- 635 • Creating/modifying/deleting any object on the token.  
 636 • Creating/modifying/deleting a token object on the token.  
 637 • Changing the SO's PIN.  
 638 • Changing the normal user's PIN.

639 The token may change the value of the **CKF\_WRITE\_PROTECTED** flag depending on the session state  
 640 to implement its object management policy. For instance, the token may set the  
 641 **CKF\_WRITE\_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy  
 642 that does not allow any objects, public or private, to be created, modified, or deleted unless the user has  
 643 successfully called C\_Login.

644 The **CKF\_USER\_PIN\_COUNT\_LOW**, **CKF\_USER\_PIN\_COUNT\_LOW**, **CKF\_USER\_PIN\_FINAL\_TRY**,  
 645 and **CKF\_SO\_PIN\_FINAL\_TRY** flags may always be set to false if the token does not support the  
 646 functionality or will not reveal the information because of its security policy.

647 The **CKF\_USER\_PIN\_TO\_BE\_CHANGED** and **CKF\_SO\_PIN\_TO\_BE\_CHANGED** flags may always be  
 648 set to false if the token does not support the functionality. If a PIN is set to the default value, or has  
 649 expired, the appropriate **CKF\_USER\_PIN\_TO\_BE\_CHANGED** or **CKF\_SO\_PIN\_TO\_BE\_CHANGED**  
 650 flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed,  
 651 but only the C\_SetPIN function can be called. Calling any other function that required the user to be  
 652 logged in will cause **CKR\_PIN\_EXPIRED** to be returned until C\_SetPIN is called successfully.

653 **CK\_TOKEN\_INFO Note:** The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount,  
654 ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and  
655 ulFreePrivateMemory can have the special value CK\_UNAVAILABLE\_INFORMATION, which means that  
656 the token and/or library is unable or unwilling to provide that information. In addition, the fields  
657 ulMaxSessionCount and ulMaxRwSessionCount can have the special value  
658 CK\_EFFECTIVELY\_INFINITE, which means that there is no practical limit on the number of sessions  
659 (resp. R/W sessions) an application can have open with the token.

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

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

```
666 CK_TOKEN_INFO info;  
667 .  
668 .  
669 if ((CK_LONG) info.ulMaxSessionCount  
670     == CK_UNAVAILABLE_INFORMATION) {  
671     /* Token refuses to give value of ulMaxSessionCount */  
672     .  
673     .  
674 } else if (info.ulMaxSessionCount == CK_EFFECTIVELY_INFINITE) {  
675     /* Application can open as many sessions as it wants */  
676     .  
677     .  
678 } else {  
679     /* ulMaxSessionCount really does contain what it should */  
680     .  
681     .  
682 }
```

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

### 685 3.3 Session types

686 Cryptoki represents session information with the following types:

#### 687 ♦ CK\_SESSION\_HANDLE; CK\_SESSION\_HANDLE\_PTR

688 **CK\_SESSION\_HANDLE** is a Cryptoki-assigned value that identifies a session. It is defined as follows:

```
689 typedef CK ULONG CK_SESSION_HANDLE;
```

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

```
693 CK_INVALID_HANDLE
```

695 CK\_SESSION\_HANDLE\_PTR is a pointer to a CK\_SESSION\_HANDLE.

#### 696 ♦ CK\_USER\_TYPE

697 **CK\_USER\_TYPE** holds the types of Cryptoki users described in [PKCS11-UG] and, in addition, a  
698 context-specific type described in Section 4.9. It is defined as follows:

```
699 typedef CK ULONG CK_USER_TYPE;
```

700

---

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

702 CKU\_SO  
703 CKU\_USER  
704 CKU\_CONTEXT\_SPECIFIC

---

705 ◆ **CK\_STATE**

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

707 `typedef CK ULONG CK_STATE;`

---

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

710 CKS\_RO\_PUBLIC\_SESSION  
711 CKS\_RO\_USER\_FUNCTIONS  
712 CKS\_RW\_PUBLIC\_SESSION  
713 CKS\_RW\_USER\_FUNCTIONS  
714 CKS\_RW\_SO\_FUNCTIONS

---

715 ◆ **CK\_SESSION\_INFO; CK\_SESSION\_INFO\_PTR**

716 **CK\_SESSION\_INFO** provides information about a session. It is defined as follows:

717 `typedef struct CK_SESSION_INFO {`  
718     `CK_SLOT_ID slotID;`  
719     `CK_STATE state;`  
720     `CK_FLAGS flags;`  
721     `CK ULONG ulDeviceError;`  
722 } CK\_SESSION\_INFO;

---

724

725 The fields of the structure have the following meanings:

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

731 The following table defines the `flags` field:

732 *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

733 **CK\_SESSION\_INFO\_PTR** is a pointer to a **CK\_SESSION\_INFO**.

734 **3.4 Object types**

735 Cryptoki represents object information with the following types:

736 ◆ **CK\_OBJECT\_HANDLE; CK\_OBJECT\_HANDLE\_PTR**

737 **CK\_OBJECT\_HANDLE** is a token-specific identifier for an object. It is defined as follows:

```
738     typedef CK ULONG CK_OBJECT_HANDLE;
```

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

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

```
748     CK_INVALID_HANDLE
```

750 **CK\_OBJECT\_HANDLE\_PTR** is a pointer to a **CK\_OBJECT\_HANDLE**.

751 ◆ **CK\_OBJECT\_CLASS; CK\_OBJECT\_CLASS\_PTR**

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

```
754     typedef CK ULONG CK_OBJECT_CLASS;
```

756 Object classes are defined with the objects that use them. The type is specified on an object through the  
757 CKA\_CLASS attribute of the object.

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

```
759     CKO_VENDOR_DEFINED
```

761 Object classes **CKO\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
762 interoperability, vendors should register their object classes through the PKCS process.

763 **CK\_OBJECT\_CLASS\_PTR** is a pointer to a **CK\_OBJECT\_CLASS**.

764 ◆ **CK\_HW\_FEATURE\_TYPE**

765 **CK\_HW\_FEATURE\_TYPE** is a value that identifies a hardware feature type of a device. It is defined as  
766 follows:

```
767     typedef CK ULONG CK_HW_FEATURE_TYPE;
```

769 Hardware feature types are defined with the objects that use them. The type is specified on an object  
770 through the CKA\_HW\_FEATURE\_TYPE attribute of the object.

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

```
772     CKH_VENDOR_DEFINED
```

774 Feature types **CKH\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
775 interoperability, vendors should register their feature types through the PKCS process.

776 ◆ **CK\_KEY\_TYPE**

777 **CK\_KEY\_TYPE** is a value that identifies a key type. It is defined as follows:

778     **typedef CK ULONG CK\_KEY\_TYPE;**

780 Key types are defined with the objects and mechanisms that use them. The key type is specified on an  
781 object through the CKA\_KEY\_TYPE attribute of the object.

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

783     **CKK\_VENDOR\_DEFINED**

785 Key types **CKK\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
786 interoperability, vendors should register their key types through the PKCS process.

787 ◆ **CK\_CERTIFICATE\_TYPE**

788 **CK\_CERTIFICATE\_TYPE** is a value that identifies a certificate type. It is defined as follows:

789     **typedef CK ULONG CK\_CERTIFICATE\_TYPE;**

791 Certificate types are defined with the objects and mechanisms that use them. The certificate type is  
792 specified on an object through the CKA\_CERTIFICATE\_TYPE attribute of the object.

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

794     **CKC\_VENDOR\_DEFINED**

796 Certificate types **CKC\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
797 interoperability, vendors should register their certificate types through the PKCS process.

798 ◆ **CK\_CERTIFICATE\_CATEGORY**

799 **CK\_CERTIFICATE\_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

800     **typedef CK ULONG CK\_CERTIFICATE\_CATEGORY;**

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

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

803 ◆ **CK\_ATTRIBUTE\_TYPE**

804 **CK\_ATTRIBUTE\_TYPE** is a value that identifies an attribute type. It is defined as follows:

805     **typedef CK ULONG CK\_ATTRIBUTE\_TYPE;**

807 Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an  
808 object as a list of type, length value items. These are often specified as an attribute template.  
809 Vendor defined values for this type may also be specified.

```
810     CKA_VENDOR_DEFINED
```

811

812 Attribute types **CKA\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
813 interoperability, vendors should register their attribute types through the PKCS process.

#### 814 ◆ **CK\_ATTRIBUTE; CK\_ATTRIBUTE\_PTR**

815 **CK\_ATTRIBUTE** is a structure that includes the type, value, and length of an attribute. It is defined as  
816 follows:

```
817     typedef struct CK_ATTRIBUTE {  
818         CK_ATTRIBUTE_TYPE type;  
819         CK_VOID_PTR pValue;  
820         CK ULONG ulValueLen;  
821     } CK_ATTRIBUTE;
```

822

823 The fields of the structure have the following meanings:

824 *type* the attribute type

825 *pValue* pointer to the value of the attribute

826 *ulValueLen* length in bytes of the value

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

833

834 The constant **CK\_UNAVAILABLE\_INFORMATION** is used in the *ulValueLen* field to denote an invalid or  
835 unavailable value. See **C\_GetAttributeValue** for further details.

836

837 **CK\_ATTRIBUTE\_PTR** is a pointer to a **CK\_ATTRIBUTE**.

#### 838 ◆ **CK\_DATE**

839 **CK\_DATE** is a structure that defines a date. It is defined as follows:

```
840     typedef struct CK_DATE {  
841         CK_CHAR year[4];  
842         CK_CHAR month[2];  
843         CK_CHAR day[2];  
844     } CK_DATE;
```

845

846 The fields of the structure have the following meanings:

847 *year* the year (“1900” - “9999”)

848 *month* the month (“01” - “12”)

849 *day* the day (“01” - “31”)

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

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

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

856 ◆ **CK\_PROFILE\_ID; CK\_PROFILE\_ID\_PTR**

857 **CK\_PROFILE\_ID** is an unsigend ulong value represting a specific token profile. It is defined as follows:

```
858     typedef CK ULONG CK_PROFILE_ID;
```

860 Profiles are defines in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater  
861 than 0xffffffff may cause compatibility issues on platforms that have CK ULONG values of 32 bits, and  
862 should be avoided.

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

```
864     CKP_VENDOR_DEFINED
```

866 Profile IDs **CKP\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For  
867 interoperability, vendors should register their object classes through the PKCS process.

868

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

```
871     CKP_INVALID_ID
```

872 CK\_PROFILE\_ID\_PTR is a pointer to a CK\_PROFILE\_ID.

873 ◆ **CK\_JAVA\_MIDP\_SECURITY\_DOMAIN**

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

```
876     typedef CK ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

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

Constant	Value	Meaning
CK_SECURITY_DOMAIN_UNSPECIFIED	0x000000000UL	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

879

880 **3.5 Data types for mechanisms**

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

882 ◆ **CK\_MECHANISM\_TYPE; CK\_MECHANISM\_TYPE\_PTR**

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

```
884     typedef CK ULONG CK_MECHANISM_TYPE;
```

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

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

```
888     CKM_VENDOR_DEFINED
```

890 Mechanism types **CKM\_VENDOR\_DEFINED** and above are permanently reserved for token vendors.

891 For interoperability, vendors should register their mechanism types through the PKCS process.

892 **CK\_MECHANISM\_TYPE\_PTR** is a pointer to a **CK\_MECHANISM\_TYPE**.

893 ◆ **CK\_MECHANISM; CK\_MECHANISM\_PTR**

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

```
896     typedef struct CK_MECHANISM {  
897         CK_MECHANISM_TYPE mechanism;  
898         CK_VOID_PTR pParameter;  
899         CK ULONG ulParameterLen;  
900     } CK_MECHANISM;
```

902 The fields of the structure have the following meanings:

903       *mechanism*     the type of mechanism

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

905       *ulParameterLen*   length in bytes of the parameter

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

909 **CK\_MECHANISM\_PTR** is a pointer to a **CK\_MECHANISM**.

910 ◆ **CK\_MECHANISM\_INFO; CK\_MECHANISM\_INFO\_PTR**

911 **CK\_MECHANISM\_INFO** is a structure that provides information about a particular mechanism. It is  
912 defined as follows:

```
913     typedef struct CK_MECHANISM_INFO {  
914         CK ULONG ulMinKeySize;  
915         CK ULONG ulMaxKeySize;  
916         CK_FLAGS flags;  
917     } CK_MECHANISM_INFO;
```

919 The fields of the structure have the following meanings:

920       *ulMinKeySize*    the minimum size of the key for the mechanism (whether this is  
921                          measured in bits or in bytes is mechanism-dependent)

922       *ulMaxKeySize*    the maximum size of the key for the mechanism (whether this is  
923                          measured in bits or in bytes is mechanism-dependent)

924       *flags*           bit flags specifying mechanism capabilities

- 925 For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.  
 926 The following table defines the *flags* field:  
 927 *Table 8, Mechanism Information Flags*

<b>Bit Flag</b>	<b>Mask</b>	<b>Meaning</b>
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 <b>C_MessageEncryptInit</b>
CKF_MESSAGE_DECRYPT	0x00000004	True if the mechanism can be used with <b>C_MessageDecryptInit</b>
CKF_MESSAGE_SIGN	0x00000008	True if the mechanism can be used with <b>C_MessageSignInit</b>
CKF_MESSAGE_VERIFY	0x00000010	True if the mechanism can be used with <b>C_MessageVerifyInit</b>
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.
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	0x00000800	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>

Bit Flag	Mask	Meaning
CKF_EXTENSION	0x80000000	True if there is an extension to the flags; false if no extensions. MUST be false for this version.

928 CK\_MECHANISM\_INFO\_PTR is a pointer to a CK\_MECHANISM\_INFO.

## 929 3.6 Function types

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

### 931 ◆ CK\_RV

932 **CK\_RV** is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
933     typedef CK ULONG CK_RV;
934 
```

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

```
936     CKR_VENDOR_DEFINED
937 
```

938 Section 5.1 defines the meaning of each **CK\_RV** value. Return values **CKR\_VENDOR\_DEFINED** and  
939 above are permanently reserved for token vendors. For interoperability, vendors should register their  
940 return values through the PKCS process.

### 941 ◆ CK\_NOTIFY

942 **CK\_NOTIFY** is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is  
943 defined as follows:

```
944     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (
945         CK_SESSION_HANDLE hSession,
946         CK_NOTIFICATION event,
947         CK_VOID_PTR pApplication
948     );
949 
```

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

951 <i>hSession</i>	The handle of the session performing the callback
952 <i>event</i>	The type of notification callback
953 <i>pApplication</i>	An application-defined value. This is the same value as was passed 954 to <b>C_OpenSession</b> to open the session performing the callback

### 955 ◆ CK\_C\_XXX

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

961 ◆ **CK\_FUNCTION\_LIST;** **CK\_FUNCTION\_LIST\_PTR;**  
962 **CK\_FUNCTION\_LIST\_PTR\_PTR**

963 **CK\_FUNCTION\_LIST** is a structure which contains a Cryptoki version and a function pointer to each  
964 function in the Cryptoki API. It is defined as follows:

```
965     typedef struct CK_FUNCTION_LIST {
966         CK_VERSION version;
967         CK_C_Initialize C_Initialize;
968         CK_C_Finalize C_Finalize;
969         CK_C_GetInfo C_GetInfo;
970         CK_C_GetFunctionList C_GetFunctionList;
971         CK_C_GetSlotList C_GetSlotList;
972         CK_C_GetSlotInfo C_GetSlotInfo;
973         CK_C_GetTokenInfo C_GetTokenInfo;
974         CK_C_GetMechanismList C_GetMechanismList;
975         CK_C_GetMechanismInfo C_GetMechanismInfo;
976         CK_C_InitToken C_InitToken;
977         CK_C_InitPIN C_InitPIN;
978         CK_C_SetPIN C_SetPIN;
979         CK_C_OpenSession C_OpenSession;
980         CK_C_CloseSession C_CloseSession;
981         CK_C_CloseAllSessions C_CloseAllSessions;
982         CK_C_GetSessionInfo C_GetSessionInfo;
983
984         CK_C_GetOperationState C_GetOperationState;
985         CK_C_SetOperationState C_SetOperationState;
986         CK_C_Login C_Login;
987         CK_C_Logout C_Logout;
988         CK_C_CreateObject C_CreateObject;
989         CK_C_CopyObject C_CopyObject;
990         CK_C_DestroyObject C_DestroyObject;
991         CK_C_GetObjectSize C_GetObjectSize;
992         CK_C_GetAttributeValue C_GetAttributeValue;
993         CK_C_SetAttributeValue C_SetAttributeValue;
994         CK_C_FindObjectsInit C_FindObjectsInit;
995         CK_C_FindObjects C_FindObjects;
996         CK_C_FindObjectsFinal C_FindObjectsFinal;
997         CK_C_EncryptInit C_EncryptInit;
998         CK_C_Encrypt C_Encrypt;
999         CK_C_EncryptUpdate C_EncryptUpdate;
1000        CK_C_EncryptFinal C_EncryptFinal;
1001        CK_C_DecryptInit C_DecryptInit;
1002        CK_C_Decrypt C_Decrypt;
1003        CK_C_DecryptUpdate C_DecryptUpdate;
1004        CK_C_DecryptFinal C_DecryptFinal;
1005        CK_C_DigestInit C_DigestInit;
1006        CK_C_Digest C_Digest;
1007        CK_C_DigestUpdate C_DigestUpdate;
1008        CK_C_DigestKey C_DigestKey;
1009        CK_C_DigestFinal C_DigestFinal;
1010        CK_C_SignInit C_SignInit;
1011        CK_C_Sign C_Sign;
1012        CK_C_SignUpdate C_SignUpdate;
1013        CK_C_SignFinal C_SignFinal;
1014        CK_C_SignRecoverInit C_SignRecoverInit;
1015        CK_C_SignRecover C_SignRecover;
1016        CK_C_VerifyInit C_VerifyInit;
1017        CK_C_Verify C_Verify;
1018        CK_C_VerifyUpdate C_VerifyUpdate;
1019        CK_C_VerifyFinal C_VerifyFinal;
1020        CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1021        CK_C_VerifyRecover C_VerifyRecover;
```

```

1022     CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1023     CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1024     CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1025     CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1026     CK_C_GenerateKey C_GenerateKey;
1027     CK_C_GenerateKeyPair C_GenerateKeyPair;
1028     CK_C_WrapKey C_WrapKey;
1029     CK_C_UnwrapKey C_UnwrapKey;
1030     CK_C_DeriveKey C_DeriveKey;
1031     CK_C_SeedRandom C_SeedRandom;
1032     CK_C_GenerateRandom C_GenerateRandom;
1033     CK_C_GetFunctionStatus C_GetFunctionStatus;
1034     CK_C_CancelFunction C_CancelFunction;
1035     CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1036 } CK_FUNCTION_LIST;
1037

```

1038 Each Cryptoki library has a static **CK\_FUNCTION\_LIST** structure, and a pointer to it (or to a copy of it  
1039 which is also owned by the library) may be obtained by the **C\_GetFunctionList** function (see Section  
1040 5.2). The value that this pointer points to can be used by an application to quickly find out where the  
1041 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API  
1042 MUST have an entry point defined in the Cryptoki library's **CK\_FUNCTION\_LIST** structure. If a particular  
1043 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the  
1044 library's **CK\_FUNCTION\_LIST** structure should point to a function stub which simply returns  
1045 **CKR\_FUNCTION\_NOT\_SUPPORTED**.

1046 In this structure 'version' is the cryptoki specification version number. The major and minor versions must  
1047 be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for  
1048 this version of the specification may be returned via **C\_GetInterfaceList** or **C\_GetInterface**.

1049

1050 An application may or may not be able to modify a Cryptoki library's static **CK\_FUNCTION\_LIST**  
1051 structure. Whether or not it can, it should never attempt to do so.

1052 PKCS #11 modules must not add new functions at the end of the **CK\_FUNCTION\_LIST** that are not  
1053 contained within the defined structure. If a PKCS#11 module needs to define additional functions, they  
1054 should be placed within a vendor defined interface returned via **C\_GetInterfaceList** or **C\_GetInterface**.

1055 **CK\_FUNCTION\_LIST\_PTR** is a pointer to a **CK\_FUNCTION\_LIST**.

1056 **CK\_FUNCTION\_LIST\_PTR\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_PTR**.

1057

1058 ◆ **CK\_FUNCTION\_LIST\_3\_0; CK\_FUNCTION\_LIST\_3\_0\_PTR;**  
1059 **CK\_FUNCTION\_LIST\_3\_0\_PTR\_PTR**

1060 **CK\_FUNCTION\_LIST\_3\_0** is a structure which contains the same function pointers as in  
1061 **CK\_FUNCTION\_LIST** and additional functions added to the end of the structure that were defined in  
1062 Cryptoki version 3.0. It is defined as follows:

```

1063     typedef struct CK_FUNCTION_LIST_3_0 {
1064         CK_VERSION version;
1065         CK_C_Initialize C_Initialize;
1066         CK_C_Finalize C_Finalize;
1067         CK_C_GetInfo C_GetInfo;
1068         CK_C_GetFunctionList C_GetFunctionList;
1069         CK_C_GetSlotList C_GetSlotList;
1070         CK_C_GetSlotInfo C_GetSlotInfo;
1071         CK_C_GetTokenInfo C_GetTokenInfo;
1072         CK_C_GetMechanismList C_GetMechanismList;
1073         CK_C_GetMechanismInfo C_GetMechanismInfo;
1074         CK_C_InitToken C_InitToken;
1075         CK_C_InitPIN C_InitPIN;

```

```
1076     CK_C_SetPIN C_SetPIN;
1077     CK_C_OpenSession C_OpenSession;
1078     CK_C_CloseSession C_CloseSession;
1079     CK_C_CloseAllSessions C_CloseAllSessions;
1080     CK_C_GetSessionInfo C_GetSessionInfo;
1081     CK_C_GetOperationState C_GetOperationState;
1082     CK_C_SetOperationState C_SetOperationState;
1083     CK_C_Login C_Login;
1084     CK_C_Logout C_Logout;
1085     CK_C_CreateObject C_CreateObject;
1086     CK_C_CopyObject C_CopyObject;
1087     CK_C_DestroyObject C_DestroyObject;
1088     CK_C_GetObjectSize C_GetObjectSize;
1089     CK_C_GetAttributeValue C_GetAttributeValue;
1090     CK_C_SetAttributeValue C_SetAttributeValue;
1091     CK_C_FindObjectsInit C_FindObjectsInit;
1092     CK_C_FindObjects C_FindObjects;
1093     CK_C_FindObjectsFinal C_FindObjectsFinal;
1094     CK_C_EncryptInit C_EncryptInit;
1095     CK_C_Encrypt C_Encrypt;
1096     CK_C_EncryptUpdate C_EncryptUpdate;
1097     CK_C_EncryptFinal C_EncryptFinal;
1098     CK_C_DecryptInit C_DecryptInit;
1099     CK_C_Decrypt C_Decrypt;
1100     CK_C_DecryptUpdate C_DecryptUpdate;
1101     CK_C_DecryptFinal C_DecryptFinal;
1102     CK_C_DigestInit C_DigestInit;
1103     CK_C_Digest C_Digest;
1104     CK_C_DigestUpdate C_DigestUpdate;
1105     CK_C_DigestKey C_DigestKey;
1106     CK_C_DigestFinal C_DigestFinal;
1107     CK_C_SignInit C_SignInit;
1108     CK_C_Sign C_Sign;
1109     CK_C_SignUpdate C_SignUpdate;
1110     CK_C_SignFinal C_SignFinal;
1111     CK_C_SignRecoverInit C_SignRecoverInit;
1112     CK_C_SignRecover C_SignRecover;
1113     CK_C_VerifyInit C_VerifyInit;
1114     CK_C_Verify C_Verify;
1115     CK_C_VerifyUpdate C_VerifyUpdate;
1116     CK_C_VerifyFinal C_VerifyFinal;
1117     CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1118     CK_C_VerifyRecover C_VerifyRecover;
1119     CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1120     CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1121     CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1122     CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1123     CK_C_GenerateKey C_GenerateKey;
1124     CK_C_GenerateKeyPair C_GenerateKeyPair;
1125     CK_C_WrapKey C_WrapKey;
1126     CK_C_UnwrapKey C_UnwrapKey;
1127     CK_C_DeriveKey C_DeriveKey;
1128     CK_C_SeedRandom C_SeedRandom;
1129     CK_C_GenerateRandom C_GenerateRandom;
1130     CK_C_GetFunctionStatus C_GetFunctionStatus;
1131     CK_C_CancelFunction C_CancelFunction;
1132     CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1133     CK_C_GetInterfaceList C_GetInterfaceList;
1134     CK_C_GetInterface C_GetInterface;
1135     CK_C_LoginUser C_LoginUser;
1136     CK_C_SessionCancel C_SessionCancel;
1137     CK_C_MessageEncryptInit C_MessageEncryptInit;
1138     CK_C_EncryptMessage C_EncryptMessage;
1139     CK_C_EncryptMessageBegin C_EncryptMessageBegin;
```

```

1140     CK_C_EncryptMessageNext C_EncryptMessageNext;
1141     CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1142     CK_C_MessageDecryptInit C_MessageDecryptInit;
1143     CK_C_DecryptMessage C_DecryptMessage;
1144     CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1145     CK_C_DecryptMessageNext C_DecryptMessageNext;
1146     CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1147     CK_C_MessageSignInit C_MessageSignInit;
1148     CK_C_SignMessage C_SignMessage;
1149     CK_C_SignMessageBegin C_SignMessageBegin;
1150     CK_C_SignMessageNext C_SignMessageNext;
1151     CK_C_MessageSignFinal C_MessageSignFinal;
1152     CK_C_MessageVerifyInit C_MessageVerifyInit;
1153     CK_C_VerifyMessage C_VerifyMessage;
1154     CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1155     CK_C_VerifyMessageNext C_VerifyMessageNext;
1156     CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1157 } CK_FUNCTION_LIST_3_0;
1158

```

1159 For a general description of **CK\_FUNCTION\_LIST\_3\_0** see **CK\_FUNCTION\_LIST**.

1160 In this structure, *version* is the cryptoki specification version number. It should match the value of  
1161 *cryptokiVersion* returned in the **CK\_INFO** structure, but must be 3.0 at minimum.

1162 This function list may be returned via **C\_GetInterfaceList** or **C\_GetInterface**

1163 **CK\_FUNCTION\_LIST\_3\_0\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_3\_0**.

1164 **CK\_FUNCTION\_LIST\_3\_0\_PTR\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_3\_0\_PTR**.

#### 1165 ♦ **CK\_INTERFACE; CK\_INTERFACE\_PTR; CK\_INTERFACE\_PTR\_PTR**

1166 **CK\_INTERFACE** is a structure which contains an interface name with a function list and flag.

1167 It is defined as follows:

```

1168     typedef struct CK_INTERFACE {
1169         CK_UTF8CHAR_PTR pInterfaceName;
1170         CK_VOID_PTR     pFunctionList;
1171         CK_FLAGS        flags;
1172     } CK_INTERFACE;

```

1173

1174 The fields of the structure have the following meanings:

1175       *pInterfaceName*     the name of the interface

1176       *pFunctionList*    the interface function list which must always begin with a  
1177                          **CK\_VERSION** structure as the first field

1178       *flags*           bit flags specifying interface capabilities

1179 The interface name "PKCS 11" is reserved for use by interfaces defined within the cryptoki specification.

1180 Interfaces starting with the string: "Vendor " are reserved for vendor use and will not otherwise be  
1181 defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface  
1182 names of "Vendor {vendor name}". For example "Vendor ACME Inc".

1183

1184 The following table defines the flags field:

1185 *Table 9, CK\_INTERFACE Flags*

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

1186

1187 CK\_INTERFACE\_PTR is a pointer to a CK\_INTERFACE.

1188 CK\_INTERFACE\_PTR\_PTR is a pointer to a CK\_INTERFACE\_PTR.

1189 

### 3.7 Locking-related types

1190 The types in this section are provided solely for applications which need to access Cryptoki from multiple  
1191 threads simultaneously. *Applications which will not do this need not use any of these types.*1192 ◆ **CK\_CREATEMUTEX**1193 CK\_CREATEMUTEX is the type of a pointer to an application-supplied function which creates a new  
1194 mutex object and returns a pointer to it. It is defined as follows:

```
1195     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX) (
1196         CK_VOID_PTR_PTR ppMutex
1197     );

```

1199 Calling a CK\_CREATEMUTEX function returns the pointer to the new mutex object in the location pointed  
1200 to by ppMutex. Such a function should return one of the following values:

```
1201     CKR_OK, CKR_GENERAL_ERROR
1202     CKR_HOST_MEMORY
```

1203 ◆ **CK\_DESTROYMUTEX**1204 CK\_DESTROYMUTEX is the type of a pointer to an application-supplied function which destroys an  
1205 existing mutex object. It is defined as follows:

```
1206     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX) (
1207         CK_VOID_PTR pMutex
1208     );

```

1210 The argument to a CK\_DESTROYMUTEX function is a pointer to the mutex object to be destroyed. Such  
1211 a function should return one of the following values:

```
1212     CKR_OK, CKR_GENERAL_ERROR
1213     CKR_HOST_MEMORY
1214     CKR_MUTEX_BAD
```

1215 ◆ **CK\_LOCKMUTEX** and **CK\_UNLOCKMUTEX**

1216 **CK\_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing  
1217 mutex object. **CK\_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which  
1218 unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- 1219 • If a CK\_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a  
1220 lock on that mutex and returns.
- 1221 • If a CK\_LOCKMUTEX function is called on a mutex which is locked by some thread other than the  
1222 calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- 1223 • If a CK\_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the  
1224 behavior of the function call is undefined.
- 1225 • If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that  
1226 mutex is unlocked and the function call returns. Furthermore:
- 1227     ◦ If exactly one thread was blocking on that particular mutex, then that thread stops blocking,  
1228         obtains a lock on that mutex, and its CK\_LOCKMUTEX call returns.
- 1229     ◦ If more than one thread was blocking on that particular mutex, then exactly one of the  
1230         blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on  
1231         the mutex, and its CK\_LOCKMUTEX call returns. All other threads blocking on that particular  
1232         mutex continue to block.
- 1233 • If a CK\_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call  
1234         returns the error code CKR\_MUTEX\_NOT\_LOCKED.
- 1235 • If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the  
1236         calling thread, the behavior of the function call is undefined.

1237 **CK\_LOCKMUTEX** is defined as follows:

```
1238     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX) (
1239         CK_VOID_PTR pMutex
1240     );
```

1242 The argument to a CK\_LOCKMUTEX function is a pointer to the mutex object to be locked. Such a  
1243 function should return one of the following values:

```
1244     CKR_OK, CKR_GENERAL_ERROR
1245     CKR_HOST_MEMORY,
1246     CKR_MUTEX_BAD
1247
```

1248 **CK\_UNLOCKMUTEX** is defined as follows:

```
1249     typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX) (
1250         CK_VOID_PTR pMutex
1251     );
```

1253 The argument to a CK\_UNLOCKMUTEX function is a pointer to the mutex object to be unlocked. Such a  
1254 function should return one of the following values:

```
1255     CKR_OK, CKR_GENERAL_ERROR
1256     CKR_HOST_MEMORY
1257     CKR_MUTEX_BAD
1258     CKR_MUTEX_NOT_LOCKED
```

1259 ◆ **CK\_C\_INITIALIZE\_ARGS; CK\_C\_INITIALIZE\_ARGS\_PTR**

1260 **CK\_C\_INITIALIZE\_ARGS** is a structure containing the optional arguments for the **C\_Initialize** function.  
1261 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals  
1262 with threads. **CK\_C\_INITIALIZE\_ARGS** is defined as follows:

```
1263     typedef struct CK_C_INITIALIZE_ARGS {  
1264         CK_CREATEMUTEX CreateMutex;  
1265         CK_DESTROYMUTEX DestroyMutex;  
1266         CK_LOCKMUTEX LockMutex;  
1267         CK_UNLOCKMUTEX UnlockMutex;  
1268         CK_FLAGS flags;  
1269         CK_VOID_PTR pReserved;  
1270     } CK_C_INITIALIZE_ARGS;  
1271
```

1272 The fields of the structure have the following meanings:

1273	<i>CreateMutex</i>	pointer to a function to use for creating mutex objects
1274	<i>DestroyMutex</i>	pointer to a function to use for destroying mutex objects
1275	<i>LockMutex</i>	pointer to a function to use for locking mutex objects
1276	<i>UnlockMutex</i>	pointer to a function to use for unlocking mutex objects
1277	<i>flags</i>	bit flags specifying options for <b>C_Initialize</b> ; the flags are defined below
1278	<i>pReserved</i>	reserved for future use. Should be NULL_PTR for this version of Cryptoki

1281 The following table defines the flags field:

1282 *Table 10, C\_Initialize Parameter Flags*

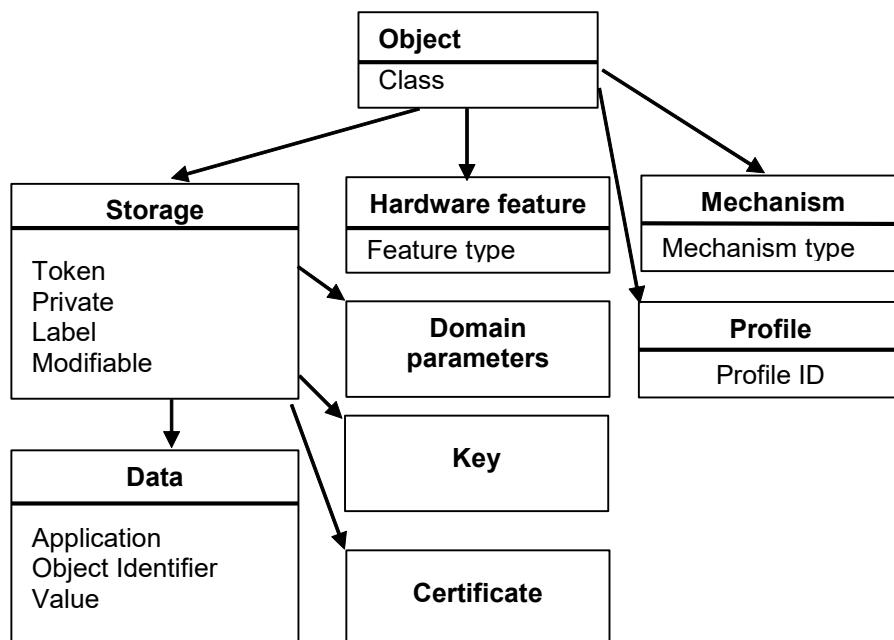
Bit Flag	Mask	Meaning
CKF_LIBRARY_CANT_CREATE_OS_THREADS	0x00000001	True if application threads which are executing calls to the library may <i>not</i> use native operating system calls to spawn new threads; false if they may
CKF_OS_LOCKING_OK	0x00000002	True if the library can use the native operation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

1283 **CK\_C\_INITIALIZE\_ARGS\_PTR** is a pointer to a **CK\_C\_INITIALIZE\_ARGS**.

---

## 1284 4 Objects

1285 Cryptoki recognizes a number of classes of objects, as defined in the **CK\_OBJECT\_CLASS** data type.  
1286 An object consists of a set of attributes, each of which has a given value. Each attribute that an object  
1287 possesses has precisely one value. The following figure illustrates the high-level hierarchy of the  
1288 Cryptoki objects and some of the attributes they support:



1289  
1290 *Figure 1, Object Attribute Hierarchy*

1291 Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and  
1292 modifying the values of their attributes. Some of the cryptographic functions (e.g., **C\_GenerateKey**) also  
1293 create key objects to hold their results.

1294 Objects are always “well-formed” in Cryptoki—that is, an object always contains all required attributes,  
1295 and the attributes are always consistent with one another from the time the object is created. This  
1296 contrasts with some object-based paradigms where an object has no attributes other than perhaps a  
1297 class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

1298 Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the  
1299 attribute value and the meaning of the attribute, which may include a default initial value. Some of the  
1300 data types are defined explicitly by Cryptoki (e.g., **CK\_OBJECT\_CLASS**). Attribute values may also take  
1301 the following types:

1302 Byte array an arbitrary string (array) of **CK\_BYTEs**

1303 Big integer a string of **CK\_BYTEs** representing an unsigned integer of arbitrary  
1304 size, most-significant byte first (e.g., the integer 32768 is  
1305 represented as the 2-byte string 0x80 0x00)

1306 Local string an unpadded string of **CK\_CHARs** (see Table 3) with no null-  
1307 termination

1308 RFC2279 string an unpadded string of **CK\_UTF8CHARs** with no null-termination

1309 A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the  
1310 same values for all their attributes.

1311 In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of  
1312 Cryptoki attributes. Some of these attributes possess default values, and need not be specified when  
1313 creating an object; some of these default values may even be the empty string (""). Nonetheless, the  
1314 object possesses these attributes. A given object has a single value for each attribute it possesses, even  
1315 if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

1316 In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes  
1317 whose meanings and values are not specified by Cryptoki.

## 1318 **4.1 Creating, modifying, and copying objects**

1319 All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments,  
1320 where the template specifies attribute values. Cryptographic functions that create objects (see Section  
1321 5.18) may also contribute some additional attribute values themselves; which attributes have values  
1322 contributed by a cryptographic function call depends on which cryptographic mechanism is being  
1323 performed (see [PKCS11-Curr] and [PKCS11-Hist] for specification of mechanisms for PKCS #11). In  
1324 any case, all the required attributes supported by an object class that do not have default values MUST  
1325 be specified when an object is created, either in the template or by the function itself.

### 1326 **4.1.1 Creating objects**

1327 Objects may be created with the Cryptoki functions **C\_CreateObject** (see Section 5.7), **C\_GenerateKey**,  
1328 **C\_GenerateKeyPair**, **C\_UnwrapKey**, and **C\_DeriveKey** (see Section 5.18). In addition, copying an  
1329 existing object (with the function **C\_CopyObject**) also creates a new object, but we consider this type of  
1330 object creation separately in Section 4.1.3.

1331 Attempting to create an object with any of these functions requires an appropriate template to be  
1332 supplied.

1333 1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the  
1334 error code **CKR\_ATTRIBUTE\_TYPE\_INVALID**. An attribute is valid if it is either one of the attributes  
1335 described in the Cryptoki specification or an additional vendor-specific attribute supported by the library  
1336 and token.

1337 2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with  
1338 the error code **CKR\_ATTRIBUTE\_VALUE\_INVALID**. The valid values for Cryptoki attributes are  
1339 described in the Cryptoki specification.

1340 3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the  
1341 error code **CKR\_ATTRIBUTE\_READ\_ONLY**. Whether or not a given Cryptoki attribute is read-only is  
1342 explicitly stated in the Cryptoki specification; however, a particular library and token may be even more  
1343 restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may  
1344 nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of  
1345 other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-  
1346 only is obviously outside the scope of Cryptoki.

1347 4. If the attribute values in the supplied template, together with any default attribute values and any  
1348 attribute values contributed to the object by the object-creation function itself, are insufficient to fully  
1349 specify the object to create, then the attempt should fail with the error code  
1350 **CKR\_TEMPLATE\_INCOMPLETE**.

1351 5. If the attribute values in the supplied template, together with any default attribute values and any  
1352 attribute values contributed to the object by the object-creation function itself, are inconsistent, then the  
1353 attempt should fail with the error code **CKR\_TEMPLATE\_INCONSISTENT**. A set of attribute values is  
1354 inconsistent if not all of its members can be satisfied simultaneously *by the token*, although each value  
1355 individually is valid in Cryptoki. One example of an inconsistent template would be using a template  
1356 which specifies two different values for the same attribute. Another example would be trying to create  
1357 a secret key object with an attribute which is appropriate for various types of public keys or private keys,  
1358 but not for secret keys. A final example would be a template with an attribute that violates some token

1359 specific requirement. Note that this final example of an inconsistent template is token-dependent—on  
1360 a different token, such a template might *not* be inconsistent.

- 1361 6. If the supplied template specifies the same value for a particular attribute more than once (or the  
1362 template specifies the same value for a particular attribute that the object-creation function itself  
1363 contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to  
1364 create an object can either succeed—thereby creating the same object that would have been created  
1365 if the multiply-specified attribute had only appeared once—or it can fail with error code  
1366 CKR\_TEMPLATE\_INCONSISTENT. Library developers are encouraged to make their libraries behave  
1367 as though the attribute had only appeared once in the template; application developers are strongly  
1368 encouraged never to put a particular attribute into a particular template more than once.

1369 If more than one of the situations listed above applies to an attempt to create an object, then the error  
1370 code returned from the attempt can be any of the error codes from above that applies.

### 1371 4.1.2 Modifying objects

1372 Objects may be modified with the Cryptoki function **C\_SetAttributeValue** (see Section 5.7). The  
1373 template supplied to **C\_SetAttributeValue** can contain new values for attributes which the object already  
1374 possesses; values for attributes which the object does not yet possess; or both.

1375 Some attributes of an object may be modified after the object has been created, and some may not. In  
1376 addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some  
1377 tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is  
1378 modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually  
1379 support modification of some such attributes. Furthermore, whether or not a particular attribute of an  
1380 object on a particular token is modifiable might depend on the values of certain attributes of the object.  
1381 For example, a secret key object's **CKA\_SENSITIVE** attribute can be changed from CK\_FALSE to  
1382 CK\_TRUE, but not the other way around.

1383 All the scenarios in Section 4.1.1—and the error codes they return—apply to modifying objects with  
1384 **C\_SetAttributeValue**, except for the possibility of a template being incomplete.

### 1385 4.1.3 Copying objects

1386 Unless an object's **CKA\_COPYABLE** (see Table 17) attribute is set to CK\_FALSE, it may be copied with  
1387 the Cryptoki function **C\_CopyObject** (see Section 5.7). In the process of copying an object,  
1388 **C\_CopyObject** also modifies the attributes of the newly-created copy according to an application-  
1389 supplied template.

1390 The Cryptoki attributes which can be modified during the course of a **C\_CopyObject** operation are the  
1391 same as the Cryptoki attributes which are described as being modifiable, plus the four special attributes  
1392 **CKA\_TOKEN**, **CKA\_PRIVATE**, **CKA\_MODIFIABLE** and **CKA\_DESTROYABLE**. To be more precise,  
1393 these attributes are modifiable during the course of a **C\_CopyObject** operation *insofar as the Cryptoki*  
1394 *specification is concerned*. A particular token might not actually support modification of some such  
1395 attributes during the course of a **C\_CopyObject** operation. Furthermore, whether or not a particular  
1396 attribute of an object on a particular token is modifiable during the course of a **C\_CopyObject** operation  
1397 might depend on the values of certain attributes of the object. For example, a secret key object's  
1398 **CKA\_SENSITIVE** attribute can be changed from CK\_FALSE to CK\_TRUE during the course of a  
1399 **C\_CopyObject** operation, but not the other way around.

1400 If the **CKA\_COPYABLE** attribute of the object to be copied is set to CK\_FALSE, **C\_CopyObject** returns  
1401 CKR\_ACTION\_PROHIBITED. Otherwise, the scenarios described in 10.1.1 - and the error codes they  
1402 return - apply to copying objects with **C\_CopyObject**, except for the possibility of a template being  
1403 incomplete.

## 1404 4.2 Common attributes

1405 *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**.
- <sup>4</sup> MUST *not* be specified when object is generated with **C\_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.

1406

1407 *Table 12, Common Object Attributes*

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

1408 Refer to Table 11 for footnotes

1409 The above table defines the attributes common to all objects.

## 1410 **4.3 Hardware Feature Objects**

### 1411 **4.3.1 Definitions**

1412 This section defines the object class CKO\_HW\_FEATURE for type CK\_OBJECT\_CLASS as used in the  
1413 CKA\_CLASS attribute of objects.

### 1414 **4.3.2 Overview**

1415 Hardware feature objects (**CKO\_HW\_FEATURE**) represent features of the device. They provide an easily  
1416 expandable method for introducing new value-based features to the Cryptoki interface.

1417 When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, hardware feature objects are  
1418 not returned unless the **CKA\_CLASS** attribute in the template has the value **CKO\_HW\_FEATURE**. This  
1419 protects applications written to previous versions of Cryptoki from finding objects that they do not  
1420 understand.

1421 *Table 13, Hardware Feature Common Attributes*

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

1422 - Refer to Table 11 for footnotes

1423 **4.3.3 Clock**

1424 **4.3.3.1 Definition**

1425 The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_CLOCK of type  
1426 CK\_HW\_FEATURE\_TYPE.

1427 **4.3.3.2 Description**

1428 Clock objects represent real-time clocks that exist on the device. This represents the same clock source  
1429 as the **utcTime** field in the **CK\_TOKEN\_INFO** structure.

1430 *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).

1431 The **CKA\_VALUE** attribute may be set using the **C\_SetAttributeValue** function if permitted by the  
1432 device. The session used to set the time MUST be logged in. The device may require the SO to be the  
1433 user logged in to modify the time value. **C\_SetAttributeValue** will return the error  
1434 CKR\_USER\_NOT\_LOGGED\_IN to indicate that a different user type is required to set the value.

1435 **4.3.4 Monotonic Counter Objects**

1436 **4.3.4.1 Definition**

1437 The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_MONOTONIC\_COUNTER of type  
1438 CK\_HW\_FEATURE\_TYPE.

1439 **4.3.4.2 Description**

1440 Monotonic counter objects represent hardware counters that exist on the device. The counter is  
1441 guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an  
1442 application for generating serial numbers to get some assurance of uniqueness per token.

1443 *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.

1444 <sup>1</sup>Read Only

1445 The **CKA\_VALUE** attribute may not be set by the client.

1446 **4.3.5 User Interface Objects**

1447 **4.3.5.1 Definition**

1448 The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_USER\_INTERFACE of type  
1449 CK\_HW\_FEATURE\_TYPE.

1450 **4.3.5.2 Description**

1451 User interface objects represent the presentation capabilities of the device.

1452 *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 ( <a href="http://www.iana.org">www.iana.org</a> ). 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 ( <a href="http://www.iana.org">www.iana.org</a> ). 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 ( <a href="http://www.iana.org">www.iana.org</a> ). 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".

1453 The selection of attributes, and associated data types, has been done in an attempt to stay as aligned  
1454 with RFC 2534 and CC/PP Struct as possible. The special value CK\_UNAVAILABLE\_INFORMATION  
1455 may be used for CK ULONG-based attributes when information is not available or applicable.

1456 None of the attribute values may be set by an application.

1457 The value of the **CKA\_ENCODING\_METHODS** attribute may be used when the application needs to  
1458 send MIME objects with encoded content to the token.

1459 **4.4 Storage Objects**

1460 This is not an object class; hence no CKO\_ definition is required. It is a category of object classes with  
1461 common attributes for the object classes that follow.

<b>Attribute</b>	<b>Data Type</b>	<b>Meaning</b>
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.

- 1463 Only the **CKA\_LABEL** attribute can be modified after the object is created. (The **CKA\_TOKEN**,  
 1464 **CKA\_PRIVATE**, and **CKA\_MODIFIABLE** attributes can be changed in the process of copying an object,  
 1465 however.)
- 1466 The **CKA\_TOKEN** attribute identifies whether the object is a token object or a session object.
- 1467 When the **CKA\_PRIVATE** attribute is CK\_TRUE, a user may not access the object until the user has  
 1468 been authenticated to the token.
- 1469 The value of the **CKA\_MODIFIABLE** attribute determines whether or not an object is read-only.
- 1470 The **CKA\_LABEL** attribute is intended to assist users in browsing.
- 1471 The value of the **CKA\_COPYABLE** attribute determines whether or not an object can be copied. This  
 1472 attribute can be used in conjunction with **CKA\_MODIFIABLE** to prevent changes to the permitted usages  
 1473 of keys and other objects.
- 1474 The value of the **CKA\_DESTROYABLE** attribute determines whether the object can be destroyed using  
 1475 C\_DestroyObject.

#### 1476 **4.4.1 The CKA\_UNIQUE\_ID attribute**

- 1477 Any time a new object is created, a value for **CKA\_UNIQUE\_ID** MUST be generated by the token and  
 1478 stored with the object. The specific algorithm used to generate unique ID values for objects is token-  
 1479 specific, but values generated MUST be unique across all objects visible to any particular session, and  
 1480 SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling  
 1481 C\_InitToken, MAY cause reuse of **CKA\_UNIQUE\_ID** values.
- 1482 Any attempt to modify the **CKA\_UNIQUE\_ID** attribute of an existing object or to specify the value of the  
 1483 **CKA\_UNIQUE\_ID** attribute in the template for an operation that creates one or more objects MUST fail.  
 1484 Operations failing for this reason return the error code **CKR\_ATTRIBUTE\_READ\_ONLY**.
- 1485

1486 **4.5 Data objects**

1487 **4.5.1 Definitions**

1488 This section defines the object class CKO\_DATA for type CK\_OBJECT\_CLASS as used in the  
1489 CKA\_CLASS attribute of objects.

1490 **4.5.2 Overview**

1491 Data objects (object class **CKO\_DATA**) hold information defined by an application. Other than providing  
1492 access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the  
1493 attributes supported by data objects, in addition to the common attributes defined for this object class:

1494 *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)

1495 The **CKA\_APPLICATION** attribute provides a means for applications to indicate ownership of the data  
1496 objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has  
1497 access to a data object, however.

1498 The **CKA\_OBJECT\_ID** attribute provides an application independent and expandable way to indicate the  
1499 type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier  
1500 matches the data value.

1501 The following is a sample template containing attributes for creating a data object:

```
1502 CK_OBJECT_CLASS class = CKO_DATA;
1503 CK_UTF8CHAR label[] = "A data object";
1504 CK_UTF8CHAR application[] = "An application";
1505 CK_BYTE data[] = "Sample data";
1506 CK_BBOOL true = CK_TRUE;
1507 CK_ATTRIBUTE template[] = {
1508     {CKA_CLASS, &class, sizeof(class)},
1509     {CKA_TOKEN, &true, sizeof(true)},
1510     {CKA_LABEL, label, sizeof(label)-1},
1511     {CKA_APPLICATION, application, sizeof(application)-1},
1512     {CKA_VALUE, data, sizeof(data)}}
1513 };
```

1514

1515 **4.6 Certificate objects**

1516 **4.6.1 Definitions**

1517 This section defines the object class CKO\_CERTIFICATE for type CK\_OBJECT\_CLASS as used in the  
1518 CKA\_CLASS attribute of objects.

1519 **4.6.2 Overview**

1520 Certificate objects (object class **CKO\_CERTIFICATE**) hold public-key or attribute certificates. Other than  
1521 providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The  
1522 following table defines the common certificate object attributes, in addition to the common attributes  
1523 defined for this object class:

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
CKA_CERTIFICATE_TYPE <sub>1</sub>	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_UNSPECIFIED)
CKA_CHECK_VALUE	Byte array	Checksum
CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)
CKA_END_DATE	CK_DATE	End date for the certificate (default empty)
CKA_PUBLIC_KEY_INFO	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)

1525 Refer to Table 11 for footnotes

1526 Cryptoki does not enforce the relationship of the CKA\_PUBLIC\_KEY\_INFO to the public key in the  
1527 certificate, but does recommend that the key be extracted from the certificate to create this value.1528 The **CKA\_CERTIFICATE\_TYPE** attribute may not be modified after an object is created. This version of  
1529 Cryptoki supports the following certificate types:

- 1530     • X.509 public key certificate
- 
- 1531     • WTLS public key certificate
- 
- 1532     • X.509 attribute certificate

1533 The **CKA\_TRUSTED** attribute cannot be set to CK\_TRUE by an application. It MUST be set by a token  
1534 initialization application or by the token's SO. Trusted certificates cannot be modified.1535 The **CKA\_CERTIFICATE\_CATEGORY** attribute is used to indicate if a stored certificate is a user  
1536 certificate for which the corresponding private key is available on the token ("token user"), a CA certificate  
1537 ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an  
1538 object is created.1539 The **CKA\_CERTIFICATE\_CATEGORY** and **CKA\_TRUSTED** attributes will together be used to map to  
1540 the categorization of the certificates.1541 **CKA\_CHECK\_VALUE**: The value of this attribute is derived from the certificate by taking the first three  
1542 bytes of the SHA-1 hash of the certificate object's CKA\_VALUE attribute.1543 The **CKA\_START\_DATE** and **CKA\_END\_DATE** attributes are for reference only; Cryptoki does not  
1544 attach any special meaning to them. When present, the application is responsible to set them to values  
1545 that match the certificate's encoded "not before" and "not after" fields (if any).

#### 1546 **4.6.3 X.509 public key certificate objects**

1547 X.509 certificate objects (certificate type **CKC\_X\_509**) hold X.509 public key certificates. The following  
1548 table defines the X.509 certificate object attributes, in addition to the common attributes defined for this  
1549 object class:1550 *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_PUBLIC_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_PUBLICK_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_DOMAIN	CK_JAVA_MIDP_SECURITY_DOMAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

<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.

<sup>4</sup>Can only be empty if CKA\_URL is empty.

Only the **CKA\_ID**, **CKA\_ISSUER**, and **CKA\_SERIAL\_NUMBER** attributes may be modified after the object is created.

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

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

The **CKA\_ISSUER** and **CKA\_SERIAL\_NUMBER** attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the **CKA\_ID** value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.

1569 The **CKA\_URL** attribute enables the support for storage of the URL where the certificate can be found  
 1570 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile  
 1571 environments.  
 1572 The **CKA\_HASH\_OF SUBJECT PUBLIC KEY** and **CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY**  
 1573 attributes are used to store the hashes of the public keys of the subject and the issuer. They are  
 1574 particularly important when only the URL is available to be able to correlate a certificate with a private key  
 1575 and when searching for the certificate of the issuer. The hash algorithm is defined by  
 1576 **CKA\_NAME\_HASH\_ALGORITHM**.  
 1577 The **CKA\_JAVA\_MIDP\_SECURITY\_DOMAIN** attribute associates a certificate with a Java MIDP security  
 1578 domain.

1579 The following is a sample template for creating an X.509 certificate object:

```

1580 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1581 CK_CERTIFICATE_TYPE certType = CKC_X_509;
1582 CK_UTF8CHAR label[] = "A certificate object";
1583 CK_BYTE subject[] = {...};
1584 CK_BYTE id[] = {123};
1585 CK_BYTE certificate[] = {...};
1586 CK_BBOOL true = CK_TRUE;
1587 CK_ATTRIBUTE template[] = {
1588   {CKA_CLASS, &class, sizeof(class)},
1589   {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1590   {CKA_TOKEN, &true, sizeof(true)},
1591   {CKA_LABEL, label, sizeof(label)-1},
1592   {CKA_SUBJECT, subject, sizeof(subject)},
1593   {CKA_ID, id, sizeof(id)},
1594   {CKA_VALUE, certificate, sizeof(certificate)}}
1595 };

```

#### 1596 4.6.4 WTLS public key certificate objects

1597 WTLS certificate objects (certificate type **CKC\_WTLS**) hold WTLS public key certificates. The following  
 1598 table defines the WTLS certificate object attributes, in addition to the common attributes defined for this  
 1599 object class.

1600 *Table 21: WTLS Certificate Object Attributes*

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_KEY <sup>4</sup>	Byte array	SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by <b>CKA_NAME_HASH_ALGORITHM</b>
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 <b>CKA_NAME_HASH_ALGORITHM</b>
CKA_NAME_HASH_ALGORI THM	CK_MECHANI SM_TYPE	Defines the mechanism used to calculate <b>CKA_HASH_OF SUBJECT_PUBLIC</b>

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
		_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

1601 <sup>1</sup>MUST be specified when the object is created. Can only be empty if CKA\_VALUE is empty.

1602 <sup>2</sup>MUST be specified when the object is created. MUST be non-empty if CKA\_URL is empty.

1603 <sup>3</sup>MUST be non-empty if CKA\_VALUE is empty.

1604 <sup>4</sup>Can only be empty if CKA\_URL is empty.

1605

1606 Only the **CKA\_ISSUER** attribute may be modified after the object has been created.

1607 The encoding for the **CKA\_SUBJECT**, **CKA\_ISSUER**, and **CKA\_VALUE** attributes can be found in  
1608 [WTLS].

1609 The **CKA\_URL** attribute enables the support for storage of the URL where the certificate can be found  
1610 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile  
1611 environments.

1612 The **CKA\_HASH\_OF SUBJECT PUBLIC KEY** and **CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY**  
1613 attributes are used to store the hashes of the public keys of the subject and the issuer. They are  
1614 particularly important when only the URL is available to be able to correlate a certificate with a private key  
1615 and when searching for the certificate of the issuer. The hash algorithm is defined by  
1616 CKA\_NAME\_HASH\_ALGORITHM.

1617 The following is a sample template for creating a WTLS certificate object:

```
1618 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1619 CK_CERTIFICATE_TYPE certType = CKC_WTLS;
1620 CK_UTF8CHAR label[] = "A certificate object";
1621 CK_BYTE subject[] = {...};
1622 CK_BYTE certificate[] = {...};
1623 CK_BBOOL true = CK_TRUE;
1624 CK_ATTRIBUTE template[] =
1625 {
1626     {CKA_CLASS, &class, sizeof(class)},
1627     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1628     {CKA_TOKEN, &true, sizeof(true)},
1629     {CKA_LABEL, label, sizeof(label)-1},
1630     {CKA_SUBJECT, subject, sizeof(subject)},
1631     {CKA_VALUE, certificate, sizeof(certificate)}
1632 };
```

## 1633 4.6.5 X.509 attribute certificate objects

1634 X.509 attribute certificate objects (certificate type **CKC\_X\_509\_ATTR\_CERT**) hold X.509 attribute  
1635 certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the  
1636 common attributes defined for this object class:

1637 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.

1638 <sup>1</sup>MUST be specified when the object is created1639 Only the **CKA\_AC\_ISSUER**, **CKA\_SERIAL\_NUMBER** and **CKA\_ATTR\_TYPES** attributes may be  
1640 modified after the object is created.

1641 The following is a sample template for creating an X.509 attribute certificate object:

```

1642 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
1643 CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
1644 CK_UTF8CHAR label[] = "An attribute certificate object";
1645 CK_BYTE owner[] = {...};
1646 CK_BYTE certificate[] = {...};
1647 CK_BBOOL true = CK_TRUE;
1648 CK_ATTRIBUTE template[] = {
1649     {CKA_CLASS, &class, sizeof(class)},
1650     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
1651     {CKA_TOKEN, &true, sizeof(true)},
1652     {CKA_LABEL, label, sizeof(label)-1},
1653     {CKA_OWNER, owner, sizeof(owner)},
1654     {CKA_VALUE, certificate, sizeof(certificate)}}
1655 };

```

1656 

## 4.7 Key objects

1657 

### 4.7.1 Definitions

1658 There is no CKO\_ definition for the base key object class, only for the key types derived from it.

1659 This section defines the object class CKO\_PUBLIC\_KEY, CKO\_PRIVATE\_KEY and  
1660 CKO\_SECRET\_KEY for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.1661 

### 4.7.2 Overview

1662 Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret  
1663 keys. The following common footnotes apply to all the tables describing attributes of keys:1664 The following table defines the attributes common to public key, private key and secret key classes, in  
1665 addition to the common attributes defined for this object class:

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.e., 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 style="list-style-type: none"> <li>• generated locally (i.e., on the token) with a <b>C_GenerateKey</b> or <b>C_GenerateKeyPair</b> call</li> <li>• created with a <b>C_CopyObject</b> call as a copy of a key which had its <b>CKA_LOCAL</b> 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_MECHANISMS	CK_MECHANISM_TYPE_PTR, pointer to a CK_MECHANISM_TYPE array	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_MECHANISM_TYPE.

1667 Refer to Table 11 for footnotes

1668 The **CKA\_ID** field is intended to distinguish among multiple keys. In the case of public and private keys,  
 1669 this field assists in handling multiple keys held by the same subject; the key identifier for a public key and  
 1670 its corresponding private key should be the same. The key identifier should also be the same as for the  
 1671 corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See  
 1672 Section 4.6 for further commentary.)

1673 In the case of secret keys, the meaning of the **CKA\_ID** attribute is up to the application.

1674 Note that the **CKA\_START\_DATE** and **CKA\_END\_DATE** attributes are for reference only; Cryptoki does  
 1675 not attach any special meaning to them. In particular, it does not restrict usage of a key according to the  
 1676 dates; doing this is up to the application.

1677 The **CKA\_DERIVE** attribute has the value CK\_TRUE if and only if it is possible to derive other keys from  
 1678 the key.

1679 The **CKA\_LOCAL** attribute has the value CK\_TRUE if and only if the value of the key was originally  
 1680 generated on the token by a **C\_GenerateKey** or **C\_GenerateKeyPair** call.

1681 The **CKA\_KEY\_GEN\_MECHANISM** attribute identifies the key generation mechanism used to generate  
 1682 the key material. It contains a valid value only if the **CKA\_LOCAL** attribute has the value CK\_TRUE. If  
 1683 **CKA\_LOCAL** has the value CK\_FALSE, the value of the attribute is  
 1684 CK\_UNAVAILABLE\_INFORMATION.

## 1685 4.8 Public key objects

1686 Public key objects (object class **CKO\_PUBLIC\_KEY**) hold public keys. The following table defines the  
 1687 attributes common to all public keys, in addition to the common attributes defined for this object class:

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.e., 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)

1689 - Refer to Table 11 for footnotes

1690 It is intended in the interests of interoperability that the subject name and key identifier for a public key will  
1691 be the same as those for the corresponding certificate and private key. However, Cryptoki does not  
1692 enforce this, and it is not required that the certificate and private key also be stored on the token.1693 To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for  
1694 public keys, use the following table.

1695 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

1696 The value of the CKA\_PUBLIC\_KEY\_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:

```

1697     SubjectPublicKeyInfo ::= SEQUENCE {
1698         algorithm            AlgorithmIdentifier,
1699         subjectPublicKey      BIT_STRING }

```

1700 The encodings for the subjectPublicKey field are specified in the description of the public key types in the  
 1701 appropriate [PKCS11-Curr] document for the key types defined within this specification.

## 1702 4.9 Private key objects

1703 Private key objects (object class **CKO\_PRIVATE\_KEY**) hold private keys. The following table defines the  
 1704 attributes common to all private keys, in addition to the common attributes defined for this object class:

1705 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.e., 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)
CKA_DERIVE_TEMPLATE	CK_ATTRIBUTE_PTR	For deriving keys. The attribute template to match against any keys derived using this derivation 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.

- 1706 - Refer to Table 11 for footnotes
- 1707 It is intended in the interests of interoperability that the subject name and key identifier for a private key  
 1708 will be the same as those for the corresponding certificate and public key. However, this is not enforced  
 1709 by Cryptoki, and it is not required that the certificate and public key also be stored on the token.
- 1710 If the **CKA\_SENSITIVE** attribute is CK\_TRUE, or if the **CKA\_EXTRACTABLE** attribute is CK\_FALSE,  
 1711 then certain attributes of the private key cannot be revealed in plaintext outside the token. Which  
 1712 attributes these are is specified for each type of private key in the attribute table in the section describing  
 1713 that type of key.
- 1714 The **CKA\_ALWAYS\_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user  
 1715 to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such  
 1716 as sign or decrypt. This attribute may only be set to CK\_TRUE when **CKA\_PRIVATE** is also CK\_TRUE.  
 1717 Re-authentication occurs by calling **C\_Login** with *userType* set to **CKU\_CONTEXT\_SPECIFIC**  
 1718 immediately after a cryptographic operation using the key has been initiated (e.g. after **C\_SignInit**). In  
 1719 this call, the actual user type is implicitly given by the usage requirements of the active key. If **C\_Login**  
 1720 returns CKR\_OK the user was successfully authenticated and this sets the active key in an authenticated  
 1721 state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g.

1722 by **C\_Sign**, **C\_SignFinal**,..). A return value CKR\_PIN\_INCORRECT from **C\_Login** means that the user  
1723 was denied permission to use the key and continuing the cryptographic operation will result in a behavior  
1724 as if **C\_Login** had not been called. In both of these cases the session state will remain the same,  
1725 however repeated failed re-authentication attempts may cause the PIN to be locked. **C\_Login** returns in  
1726 this case CKR\_PIN\_LOCKED and this also logs the user out from the token. Failing or omitting to re-  
1727 authenticate when CKA\_ALWAYS\_AUTHENTICATE is set to CK\_TRUE will result in  
1728 CKR\_USER\_NOT\_LOGGED\_IN to be returned from calls using the key. **C\_Login** will return  
1729 CKR\_OPERATION\_NOT\_INITIALIZED, but the active cryptographic operation will not be affected, if an  
1730 attempt is made to re-authenticate when CKA\_ALWAYS\_AUTHENTICATE is set to CK\_FALSE.

1731 The **CKA\_PUBLIC\_KEY\_INFO** attribute represents the public key associated with this private key. The  
1732 data it represents may either be stored as part of the private key data, or regenerated as needed from the  
1733 private key.

1734 If this attribute is supplied as part of a template for **C\_CreateObject**, **C\_CopyObject** or  
1735 **C\_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key  
1736 data and the public key data as supplied in **CKA\_PUBLIC\_KEY\_INFO**. This can be done either by  
1737 deriving a public key from the private key and comparing the values, or by doing a sign and verify  
1738 operation. If there is a mismatch, the command SHALL return **CKR\_ATTRIBUTE\_VALUE\_INVALID**. A  
1739 token MAY choose not to support the **CKA\_PUBLIC\_KEY\_INFO** attribute for commands which create  
1740 new private keys. If it does not support the attribute, the command SHALL return  
1741 **CKR\_ATTRIBUTE\_TYPE\_INVALID**.

1742 As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public  
1743 key information. In particular, the RSA private key description has been modified in PKCS #11 V2.40 to  
1744 add the **CKA\_PUBLIC\_EXPONENT** to the list of attributes required for an RSA private key. All other  
1745 private key types described in this specification contain sufficient information to recover the associated  
1746 public key.

## 1747 4.10 Secret key objects

1748 Secret key objects (object class **CKO\_SECRET\_KEY**) hold secret keys. The following table defines the  
1749 attributes common to all secret keys, in addition to the common attributes defined for this object class:

1750 *Table 27, 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>

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
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 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 unwrapping 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.
A_DERIVE_TEMPLATE	CK_ATTRIBUTE_PTR	For deriving keys. The attribute template to match against any keys derived using this derivation 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.

1751 Refer to Table 11 for footnotes

1752 If the **CKA\_SENSITIVE** attribute is CK\_TRUE, or if the **CKA\_EXTRACTABLE** attribute is CK\_FALSE,  
1753 then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which  
1754 attributes these are is specified for each type of secret key in the attribute table in the section describing  
1755 that type of key.

1756 The key check value (KCV) attribute for symmetric key objects to be called **CKA\_CHECK\_VALUE**, of  
1757 type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to  
1758 be used to cross-check symmetric keys against other systems where the same key is shared, and as a  
1759 validity check after manual key entry or restore from backup. Refer to object definitions of specific key  
1760 types for KCV algorithms.

1761 Properties:

- 1762 1. For two keys that are cryptographically identical the value of this attribute should be identical.
- 1763 2. CKA\_CHECK\_VALUE should not be usable to obtain any part of the key value.
- 1764 3. Non-uniqueness. Two different keys can have the same CKA\_CHECK\_VALUE. This is unlikely  
(the probability can easily be calculated) but possible.

1766 The attribute is optional, but if supported, regardless of how the key object is created or derived, the value  
1767 of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is  
1768 forbidden (i.e. when CKA\_ENCRYPT is set to CK\_FALSE).

1769 If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST  
1770 match what the library calculates it to be or the library returns a CKR\_ATTRIBUTE\_VALUE\_INVALID. If  
1771 the library does not support the attribute then it should ignore it. Allowing the attribute in the template this  
1772 way does no harm and allows the attribute to be treated like any other attribute for the purposes of key  
1773 wrap and unwrap where the attributes are preserved also.

1774 The generation of the KCV may be prevented by the application supplying the attribute in the template as  
1775 a no-value (0 length) entry. The application can query the value at any time like any other attribute using  
1776 C\_GetAttributeValue. C\_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

1777 Unless otherwise specified for the object definition, the value of this attribute is derived from the key  
1778 object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the  
1779 default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

## 1780 **4.11 Domain parameter objects**

### 1781 **4.11.1 Definitions**

1782 This section defines the object class **CKO\_DOMAIN\_PARAMETERS** for type CK\_OBJECT\_CLASS as  
1783 used in the CKA\_CLASS attribute of objects.

### 1784 **4.11.2 Overview**

1785 This object class was created to support the storage of certain algorithm's extended parameters. DSA  
1786 and DH both use domain parameters in the key-pair generation step. In particular, some libraries support  
1787 the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

1788 To use a domain parameter object you MUST extract the attributes into a template and supply them (still  
1789 in the template) to the corresponding key-pair generation function.

1790 Domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**) hold public domain parameters.

1791 The following table defines the attributes common to domain parameter objects in addition to the common  
1792 attributes defined for this object class:

1793 *Table 28, 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 <ul style="list-style-type: none"> <li>• generated locally (<i>i.e.</i>, on the token) with a <b>C_GenerateKey</b></li> <li>• created with a <b>C_CopyObject</b> call as a copy of domain parameters which had its <b>CKA_LOCAL</b> attribute set to CK_TRUE</li> </ul>

1794 ^Refer to Table 11 for footnotes

1795 The **CKA\_LOCAL** attribute has the value CK\_TRUE if and only if the values of the domain parameters  
1796 were originally generated on the token by a **C\_GenerateKey** call.

## 1797 4.12 Mechanism objects

### 1798 4.12.1 Definitions

1799 This section defines the object class CKO\_MECHANISM for type CK\_OBJECT\_CLASS as used in the  
1800 CKA\_CLASS attribute of objects.

### 1801 4.12.2 Overview

1802 Mechanism objects provide information about mechanisms supported by a device beyond that given by  
1803 the **CK\_MECHANISM\_INFO** structure.

1804 When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, mechanism objects are not  
1805 returned unless the **CKA\_CLASS** attribute in the template has the value **CKO\_MECHANISM**. This  
1806 protects applications written to previous versions of Cryptoki from finding objects that they do not  
1807 understand.

1808 *Table 29, Common Mechanism Attributes*

Attribute	Data Type	Meaning
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object

1809 The **CKA\_MECHANISM\_TYPE** attribute may not be set.

1810

## 1811 4.13 Profile objects

### 1812 4.13.1 Definitions

1813 This section defines the object class CKO\_PROFILE for type CK\_OBJECT\_CLASS as used in the  
1814 CKA\_CLASS attribute of objects.

### 1815 4.13.2 Overview

1816 Profile objects (object class CKO\_PROFILE) describe which PKCS #11 profiles the token implements.  
1817 Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given  
1818 token can contain more than one profile ID. The following table lists the attributes supported by profile  
1819 objects, in addition to the common attributes defined for this object class:

1820 *Table 30, Profile Object Attributes*

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

- 1821 The **CKA\_PROFILE\_ID** attribute identifies a profile that the token supports.

---

## 5 Functions

Cryptoki's functions are organized into the following categories:

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

In addition to these functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

The Cryptoki API functions are presented in the following table:

*Table 31, Summary of Cryptoki Functions*

Category	Function	Description
General purpose functions	C_Initialize	initializes Cryptoki
	C_Finalize	clean up miscellaneous Cryptoki-associated resources
	C_GetInfo	obtains general information about Cryptoki
	C_GetFunctionList	obtains entry points of Cryptoki library functions
	C_GetInterfaceList	obtains list of interfaces supported by Cryptoki library
	C_GetInterface	obtains interface specific entry points to Cryptoki library functions
Slot and token management functions	C_GetSlotList	obtains a list of slots in the system
	C_GetSlotInfo	obtains information about a particular slot
	C_GetTokenInfo	obtains information about a particular token
	C_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur
	C_GetMechanismList	obtains a list of mechanisms supported by a token
	C_GetMechanismInfo	obtains information about a particular mechanism
	C_InitToken	initializes a token
	C_InitPIN	initializes the normal user's PIN

<b>Category</b>	<b>Function</b>	<b>Description</b>
	C_SetPIN	modifies the PIN of the current user
Session management functions	C_OpenSession	opens a connection between an application and a particular token or sets up an application callback for token insertion
	C_CloseSession	closes a session
	C_CloseAllSessions	closes all sessions with a token
	C_GetSessionInfo	obtains information about the session
	C_SessionCancel	terminates active session based operations
	C_GetOperationState	obtains the cryptographic operations state of a session
	C_SetOperationState	sets the cryptographic operations state of a session
	C_Login	logs into a token
	C_LoginUser	logs into a token with explicit user name
	C_Logout	logs out from a token
Object management functions	C_CreateObject	creates an object
	C_CopyObject	creates a copy of an object
	C_DestroyObject	destroys an object
	C_GetObjectSize	obtains the size of an object in bytes
	C_GetAttributeValue	obtains an attribute value of an object
	C_SetAttributeValue	modifies an attribute value of an object
	C_FindObjectsInit	initializes an object search operation
	C_FindObjects	continues an object search operation
	C_FindObjectsFinal	finishes an object search operation
Encryption functions	C_EncryptInit	initializes an encryption operation
	C_Encrypt	encrypts single-part data
	C_EncryptUpdate	continues a multiple-part encryption operation
	C_EncryptFinal	finishes a multiple-part encryption operation
Message-based Encryption Functions	C_MessageEncryptInit	initializes a message-based encryption process
	C_EncryptMessage	encrypts a single-part message
	C_EncryptMessageBegin	begins a multiple-part message encryption operation
	C_EncryptMessageNext	continues or finishes a multiple-part message encryption operation
	C_MessageEncryptFinal	finishes a message-based encryption process
Decryption Functions	C_DecryptInit	initializes a decryption operation
	C_Decrypt	decrypts single-part encrypted data
	C_DecryptUpdate	continues a multiple-part decryption operation
	C_DecryptFinal	finishes a multiple-part decryption operation
Message-based Decryption Functions	C_MessageDecryptInit	initializes a message decryption operation
	C_DecryptMessage	decrypts single-part data
	C_DecryptMessageBegin	starts a multiple-part message decryption operation
	C_DecryptMessageNext	Continues and finishes a multiple-part message decryption operation

<b>Category</b>	<b>Function</b>	<b>Description</b>
	C_MessageDecryptFinal	finishes a message decryption operation
Message Digesting Functions	C_DigestInit	initializes a message-digesting operation
	C_Digest	digests single-part data
	C_DigestUpdate	continues a multiple-part digesting operation
	C_DigestKey	digests a key
	C_DigestFinal	finishes a multiple-part digesting operation
Signing and MACing functions	C_SignInit	initializes a signature operation
	C_Sign	sends single-part data
	C_SignUpdate	continues a multiple-part signature operation
	C_SignFinal	finishes a multiple-part signature operation
	C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
	C_SignRecover	sends single-part data, where the data can be recovered from the signature
Message-based Signature functions	C_MessageSignInit	initializes a message signature operation
	C_SignMessage	sends single-part data
	C_SignMessageBegin	starts a multiple-part message signature operation
	C_SignMessageNext	continues and finishes a multiple-part message signature operation
	C_MessageSignFinal	finishes a message signature operation
Functions for verifying signatures and MACs	C_VerifyInit	initializes a verification operation
	C_Verify	verifies a signature on single-part data
	C_VerifyUpdate	continues a multiple-part verification operation
	C_VerifyFinal	finishes a multiple-part verification operation
	C_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature
	C_VerifyRecover	verifies a signature on single-part data, where the data is recovered from the signature
Message-based Functions for verifying signatures and MACs	C_MessageVerifyInit	initializes a message verification operation
	C_VerifyMessage	verifies single-part data
	C_VerifyMessageBegin	starts a multiple-part message verification operation
	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation
	C_MessageVerifyFinal	finishes a message verification operation
Dual-purpose cryptographic functions	C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations
	C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations
	C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations
	C_DecryptVerifyUpdate	continues simultaneous multiple-part decryption and verification operations
Key management	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair

Category	Function	Description
functions	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
Random number generation functions	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Parallel function management functions	C_GetFunctionStatus	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
	C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
Callback function		application-supplied function to process notifications from Cryptoki

1844

1845 Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes  
1846 either its entire goal, or nothing at all.

- 1847 • If a Cryptoki function executes successfully, it returns the value CKR\_OK.
- 1848 • If a Cryptoki function does not execute successfully, it returns some value other than CKR\_OK, and  
1849 the token is in the same state as it was in prior to the function call. If the function call was supposed  
1850 to modify the contents of certain memory addresses on the host computer, these memory addresses  
1851 may have been modified, despite the failure of the function.
- 1852 • In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value  
1853 CKR\_GENERAL\_ERROR. When this happens, the token and/or host computer may be in an  
1854 inconsistent state, and the goals of the function may have been partially achieved.

1855 There are a small number of Cryptoki functions whose return values do not behave precisely as  
1856 described above; these exceptions are documented individually with the description of the functions  
1857 themselves.

1858 A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported  
1859 function MUST have a “stub” in the library which simply returns the value  
1860 CKR\_FUNCTION\_NOT\_SUPPORTED. The function’s entry in the library’s **CK\_FUNCTION\_LIST**  
1861 structure (as obtained by **C\_GetFunctionList**) should point to this stub function (see Section 3.6).

## 1862 5.1 Function return values

1863 The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we  
1864 enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1  
1865 details the behavior of Cryptoki functions, including what values each of them may return.

1866 Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications  
1867 attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to  
1868 specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are  
1869 presumably some gaps. For example, it is possible that a particular error code which might apply to a  
1870 particular Cryptoki function is unfortunately not actually listed in the description of that function as a  
1871 possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit  
1872 his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful  
1873 if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that  
1874 error code for that function. It would be far preferable for the application to examine the function’s return  
1875 value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of  
1876 error), and behave accordingly.

1877 See Section 5.1.8 for some specific details on how a developer might attempt to make an application that  
1878 accommodates a range of behaviors from Cryptoki libraries.

### 1879    5.1.1 Universal Cryptoki function return values

1880 Any Cryptoki function can return any of the following values:

- 1881    • CKR\_GENERAL\_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is  
1882    possible that the function only partially succeeded, and that the computer and/or token is in an  
1883    inconsistent state.
- 1884    • CKR\_HOST\_MEMORY: The computer that the Cryptoki library is running on has insufficient memory  
1885    to perform the requested function.
- 1886    • CKR\_FUNCTION\_FAILED: The requested function could not be performed, but detailed information  
1887    about why not is not available in this error return. If the failed function uses a session, it is possible  
1888    that the **CK\_SESSION\_INFO** structure that can be obtained by calling **C\_GetSessionInfo** will hold  
1889    useful information about what happened in its *ulDeviceError* field. In any event, although the function  
1890    call failed, the situation is not necessarily totally hopeless, as it is likely to be when  
1891    CKR\_GENERAL\_ERROR is returned. Depending on what the root cause of the error actually was, it  
1892    is possible that an attempt to make the exact same function call again would succeed.
- 1893    • CKR\_OK: The function executed successfully. Technically, CKR\_OK is not *quite* a “universal” return  
1894    value; in particular, the legacy functions **C\_GetFunctionStatus** and **C\_CancelFunction** (see Section  
1895    5.20) cannot return CKR\_OK.

1896 The relative priorities of these errors are in the order listed above, e.g., if either of  
1897 CKR\_GENERAL\_ERROR or CKR\_HOST\_MEMORY would be an appropriate error return, then  
1898 CKR\_GENERAL\_ERROR should be returned.

### 1899    5.1.2 Cryptoki function return values for functions that use a session 1900    handle

1901 Any Cryptoki function that takes a session handle as one of its arguments (i.e., any Cryptoki function  
1902 except for **C\_Initialize**, **C\_Finalize**, **C\_GetInfo**, **C\_GetFunctionList**, **C\_GetSlotList**, **C\_GetSlotInfo**,  
1903 **C\_GetTokenInfo**, **C\_WaitForSlotEvent**, **C\_GetMechanismList**, **C\_GetMechanismInfo**, **C\_InitToken**,  
1904 **C\_OpenSession**, and **C\_CloseAllSessions**) can return the following values:

- 1905    • CKR\_SESSION\_HANDLE\_INVALID: The specified session handle was invalid *at the time that the*  
1906    *function was invoked*. Note that this can happen if the session’s token is removed before the function  
1907    invocation, since removing a token closes all sessions with it.
- 1908    • CKR\_DEVICE\_REMOVED: The token was removed from its slot *during the execution of the function*.
- 1909    • CKR\_SESSION\_CLOSED: The session was closed *during the execution of the function*. Note that,  
1910    as stated in [PKCS11-UG], the behavior of Cryptoki is *undefined* if multiple threads of an application  
1911    attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no  
1912    guarantee that a function invocation could ever return the value CKR\_SESSION\_CLOSED. An  
1913    example of multiple threads accessing a common session simultaneously is where one thread is  
1914    using a session when another thread closes that same session.

1915 The relative priorities of these errors are in the order listed above, e.g., if either of  
1916 CKR\_SESSION\_HANDLE\_INVALID or CKR\_DEVICE\_REMOVED would be an appropriate error return,  
1917 then CKR\_SESSION\_HANDLE\_INVALID should be returned.

1918 In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction  
1919 between a token being removed *before* a function invocation and a token being removed *during a*  
1920 function execution.

### 1921    5.1.3 Cryptoki function return values for functions that use a token

1922 Any Cryptoki function that uses a particular token (i.e., any Cryptoki function except for **C\_Initialize**,  
1923 **C\_Finalize**, **C\_GetInfo**, **C\_GetFunctionList**, **C\_GetSlotList**, **C\_GetSlotInfo**, or **C\_WaitForSlotEvent**)  
1924 can return any of the following values:

- 1925    • CKR\_DEVICE\_MEMORY: The token does not have sufficient memory to perform the requested  
1926    function.

- 1927 • CKR\_DEVICE\_ERROR: Some problem has occurred with the token and/or slot. This error code can  
1928 be returned by more than just the functions mentioned above; in particular, it is possible for  
1929 **C\_GetSlotInfo** to return CKR\_DEVICE\_ERROR.

- 1930 • CKR\_TOKEN\_NOT\_PRESENT: The token was not present in its slot *at the time that the function was*  
1931 *invoked*.

- 1932 • CKR\_DEVICE\_REMOVED: The token was removed from its slot *during the execution of the function*.

1933 The relative priorities of these errors are in the order listed above, e.g., if either of  
1934 CKR\_DEVICE\_MEMORY or CKR\_DEVICE\_ERROR would be an appropriate error return, then  
1935 CKR\_DEVICE\_MEMORY should be returned.

1936 In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction  
1937 between a token being removed *before* a function invocation and a token being removed *during* a  
1938 function execution.

#### 1939 **5.1.4 Special return value for application-supplied callbacks**

1940 There is a special-purpose return value which is not returned by any function in the actual Cryptoki API,  
1941 but which may be returned by an application-supplied callback function. It is:

- 1942 • CKR\_CANCEL: When a function executing in serial with an application decides to give the application  
1943 a chance to do some work, it calls an application-supplied function with a CKN\_SURRENDER  
1944 callback (see Section 5.21). If the callback returns the value CKR\_CANCEL, then the function aborts  
1945 and returns CKR\_FUNCTION\_CANCELED.

#### 1946 **5.1.5 Special return values for mutex-handling functions**

1947 There are two other special-purpose return values which are not returned by any actual Cryptoki  
1948 functions. These values may be returned by application-supplied mutex-handling functions, and they may  
1949 safely be ignored by application developers who are not using their own threading model. They are:

- 1950 • CKR\_MUTEX\_BAD: This error code can be returned by mutex-handling functions that are passed a  
1951 bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a  
1952 bad mutex object. There is therefore no guarantee that such a function will successfully detect bad  
1953 mutex objects and return this value.
- 1954 • CKR\_MUTEX\_NOT\_LOCKED: This error code can be returned by mutex-unlocking functions. It  
1955 indicates that the mutex supplied to the mutex-unlocking function was not locked.

#### 1956 **5.1.6 All other Cryptoki function return values**

1957 Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions  
1958 of particular error codes, there are in general no particular priorities among the errors listed below, i.e., if  
1959 more than one error code might apply to an execution of a function, then the function may return any  
1960 applicable error code.

- 1961 • CKR\_ACTION\_PROHIBITED: This value can only be returned by C\_CopyObject,  
1962 C\_SetAttributeValue and C\_DestroyObject. It denotes that the action may not be taken, either  
1963 because of underlying policy restrictions on the token, or because the object has the relevant  
1964 CKA\_COPYABLE, CKA\_MODIFIABLE or CKA\_DESTROYABLE policy attribute set to CK\_FALSE.
- 1965 • CKR\_ARGUMENTS\_BAD: This is a rather generic error code which indicates that the arguments  
1966 supplied to the Cryptoki function were in some way not appropriate.
- 1967 • CKR\_ATTRIBUTE\_READ\_ONLY: An attempt was made to set a value for an attribute which may not  
1968 be set by the application, or which may not be modified by the application. See Section 4.1 for more  
1969 information.
- 1970 • CKR\_ATTRIBUTE\_SENSITIVE: An attempt was made to obtain the value of an attribute of an object  
1971 which cannot be satisfied because the object is either sensitive or un-extractable.

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

- 2072        2. The library cannot function properly without being able to spawn new threads in the above  
2073        fashion.
- 2074     • CKR\_NO\_EVENT: This value can only be returned by **C\_WaitForSlotEvent**. It is returned when  
2075        **C\_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
- 2076     • CKR\_OBJECT\_HANDLE\_INVALID: The specified object handle is not valid. We reiterate here that 0  
2077        is never a valid object handle.
- 2078     • CKR\_OPERATION\_ACTIVE: There is already an active operation (or combination of active  
2079        operations) which prevents Cryptoki from activating the specified operation. For example, an active  
2080        object-searching operation would prevent Cryptoki from activating an encryption operation with  
2081        **C\_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent  
2082        Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous  
2083        dual cryptographic operations in a session (see the description of the  
2084        **CKF\_DUAL\_CRYPTO\_OPERATIONS** flag in the **CK\_TOKEN\_INFO** structure), an active signature  
2085        operation would prevent Cryptoki from activating an encryption operation.
- 2086     • CKR\_OPERATION\_NOT\_INITIALIZED: There is no active operation of an appropriate type in the  
2087        specified session. For example, an application cannot call **C\_Encrypt** in a session without having  
2088        called **C\_EncryptInit** first to activate an encryption operation.
- 2089     • CKR\_PIN\_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried  
2090        out unless **C\_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a  
2091        token ever expires varies from token to token.
- 2092     • CKR\_PIN\_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the  
2093        token. More generally-- when authentication to the token involves something other than a PIN-- the  
2094        attempt to authenticate the user has failed.
- 2095     • CKR\_PIN\_INVALID: The specified PIN has invalid characters in it. This return code only applies to  
2096        functions which attempt to set a PIN.
- 2097     • CKR\_PIN\_LEN\_RANGE: The specified PIN is too long or too short. This return code only applies to  
2098        functions which attempt to set a PIN.
- 2099     • CKR\_PIN\_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some  
2100        particular number of failed authentication attempts has been reached, the token is unwilling to permit  
2101        further attempts at authentication. Depending on the token, the specified PIN may or may not remain  
2102        locked indefinitely.
- 2103     • CKR\_PIN\_TOO\_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is  
2104        too short, **CKR\_PIN\_LEN\_RANGE** should be returned instead. This return code only applies to  
2105        functions which attempt to set a PIN.
- 2106     • CKR\_PUBLIC\_KEY\_INVALID: The public key fails a public key validation. For example, an EC  
2107        public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may  
2108        be returned by **C\_CreateObject**, when the public key is created, or by **C\_VerifyInit** or  
2109        **C\_VerifyRecoverInit**, when the public key is used. It may also be returned by **C\_DeriveKey**, in  
2110        preference to **CKR\_MECHANISM\_PARAM\_INVALID**, if the other party's public key specified in the  
2111        mechanism's parameters is invalid.
- 2112     • CKR\_RANDOM\_NO\_RNG: This value can be returned by **C\_SeedRandom** and  
2113        **C\_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.  
2114        This return value has higher priority than **CKR\_RANDOM\_SEED\_NOT\_SUPPORTED**.
- 2115     • CKR\_RANDOM\_SEED\_NOT\_SUPPORTED: This value can only be returned by **C\_SeedRandom**.  
2116        It indicates that the token's random number generator does not accept seeding from an application.  
2117        This return value has lower priority than **CKR\_RANDOM\_NO\_RNG**.
- 2118     • CKR\_SAVED\_STATE\_INVALID: This value can only be returned by **C\_SetOperationState**. It  
2119        indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be  
2120        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**. 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

2169       logged into the session. For example, if an application has an open SO session, and it attempts to  
2170       log the normal user into it, it will receive this error code.

- 2171     • CKR\_USER\_NOT\_LOGGED\_IN: The desired action cannot be performed because the appropriate  
2172       user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out  
2173       unless it is logged in. Another example is that a private object cannot be created on a token unless  
2174       the session attempting to create it is logged in as the normal user. A final example is that  
2175       cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 2176     • CKR\_USER\_PIN\_NOT\_INITIALIZED: This value can only be returned by **C\_Login**. It indicates that  
2177       the normal user's PIN has not yet been initialized with **C\_InitPIN**.
- 2178     • CKR\_USER\_TOO\_MANY\_TYPES: An attempt was made to have more distinct users simultaneously  
2179       logged into the token than the token and/or library permits. For example, if some application has an  
2180       open SO session, and another application attempts to log the normal user into a session, the attempt  
2181       may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be  
2182       supported does **C\_Login** have to return this value. Note that this error code generalizes to true multi-  
2183       user tokens.
- 2184     • CKR\_USER\_TYPE\_INVALID: An invalid value was specified as a **CK\_USER\_TYPE**. Valid types are  
2185       **CKU\_SO**, **CKU\_USER**, and **CKU\_CONTEXT\_SPECIFIC**.
- 2186     • CKR\_WWRAPPED\_KEY\_INVALID: This value can only be returned by **C\_UnwrapKey**. It indicates  
2187       that the provided wrapped key is not valid. If a call is made to **C\_UnwrapKey** to unwrap a particular  
2188       type of key (*i.e.*, some particular key type is specified in the template provided to **C\_UnwrapKey**),  
2189       and the wrapped key provided to **C\_UnwrapKey** is recognizably not a wrapped key of the proper  
2190       type, then **C\_UnwrapKey** should return CKR\_WWRAPPED\_KEY\_INVALID. This return value has  
2191       lower priority than CKR\_WWRAPPED\_KEY\_LEN\_RANGE.
- 2192     • CKR\_WWRAPPED\_KEY\_LEN\_RANGE: This value can only be returned by **C\_UnwrapKey**. It  
2193       indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.  
2194       This return value has higher priority than CKR\_WWRAPPED\_KEY\_INVALID.
- 2195     • CKR\_WWRAPPING\_KEY\_HANDLE\_INVALID: This value can only be returned by **C\_WrapKey**. It  
2196       indicates that the key handle specified to be used to wrap another key is not valid.
- 2197     • CKR\_WWRAPPING\_KEY\_SIZE\_RANGE: This value can only be returned by **C\_WrapKey**. It indicates  
2198       that although the requested wrapping operation could in principle be carried out, this Cryptoki library  
2199       (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range  
2200       of key sizes that it can handle.
- 2201     • CKR\_WWRAPPING\_KEY\_TYPE\_INCONSISTENT: This value can only be returned by **C\_WrapKey**. It  
2202       indicates that the type of the key specified to wrap another key is not consistent with the mechanism  
2203       specified for wrapping.
- 2204     • CKR\_OPERATION\_CANCEL\_FAILED: This value can only be returned by **C\_SessionCancel**. It  
2205       means that one or more of the requested operations could not be cancelled for implementation or  
2206       vendor-specific reasons.

### 2207     5.1.7 More on relative priorities of Cryptoki errors

2208     In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than CKR\_OK) take  
2209       precedence over error codes from Section 5.1.2, which take precedence over error codes from Section  
2210       5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that  
2211       functions that use a session handle (*i.e.*, *most* functions!) never return the error code  
2212       CKR\_TOKEN\_NOT\_PRESENT (they return CKR\_SESSION\_HANDLE\_INVALID instead). Other than  
2213       these precedences, if more than one error code applies to the result of a Cryptoki call, any of the  
2214       applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the  
2215       descriptions of functions.

## 2216 5.1.8 Error code “gotchas”

2217 Here is a short list of a few particular things about return values that Cryptoki developers might want to be  
2218 aware of:

- 2219 1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction  
2220 between a token being removed *before* a function invocation and a token being removed *during* a  
2221 function invocation.
- 2222 2. As mentioned in Section 5.1.2, an application should never count on getting a  
2223 CKR\_SESSION\_CLOSED error.
- 2224 3. The difference between CKR\_DATA\_INVALID and CKR\_DATA\_LEN\_RANGE can be somewhat  
2225 subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to  
2226 always treat them equivalently.
- 2227 4. Similarly, the difference between CKR\_ENCRYPTED\_DATA\_INVALID and  
2228 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, and between CKR\_WWRAPPED\_KEY\_INVALID and  
2229 CKR\_WWRAPPED\_KEY\_LEN\_RANGE, can be subtle, and it may be best to treat these return values  
2230 equivalently.
- 2231 5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which  
2232 of CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_TEMPLATE\_INCOMPLETE, or  
2233 CKR\_TEMPLATE\_INCONSISTENT to return. When possible, it is recommended that application  
2234 developers be generous in their interpretations of these error codes.

## 2235 5.2 Conventions for functions returning output in a variable-length 2236 buffer

2237 A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism.  
2238 The amount of output returned by these functions is returned in a variable-length application-supplied  
2239 buffer. An example of a function of this sort is **C\_Encrypt**, which takes some plaintext as an argument,  
2240 and outputs a buffer full of ciphertext.

2241 These functions have some common calling conventions, which we describe here. Two of the arguments  
2242 to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the  
2243 length of the output produced (say *pulBufLen*). There are two ways for an application to call such a  
2244 function:

- 2245 1. If *pBuf* is NULL\_PTR, then all that the function does is return (in \**pulBufLen*) a number of bytes which  
2246 would suffice to hold the cryptographic output produced from the input to the function. This number  
2247 may somewhat exceed the precise number of bytes needed, but should not exceed it by a large  
2248 amount. CKR\_OK is returned by the function.
- 2249 2. If *pBuf* is not NULL\_PTR, then \**pulBufLen* MUST contain the size in bytes of the buffer pointed to by  
2250 *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the  
2251 function, then that cryptographic output is placed there, and CKR\_OK is returned by the function and  
2252 \**pulBufLen* is set to the exact number of bytes returned. If the buffer is not large enough, then  
2253 CKR\_BUFFER\_TOO\_SMALL is returned and \**pulBufLen* is set to at least the number of bytes  
2254 needed to hold the cryptographic output produced from the input to the function.

2255 NOTE: This is a change from previous specs. The problem is that in some decrypt cases, the token  
2256 doesn't know how big a buffer is needed until the decrypt completes. The act of doing decrypt can mess  
2257 up the internal encryption state. Many tokens already implement this relaxed behavior, tokens which  
2258 implement the more precise behavior are still compliant. The one corner case is applications using a  
2259 token that knows exactly how big the decryption is (through some out of band means), could get  
2260 CKR\_BUFFER\_TOO\_SMALL returned when it supplied a buffer exactly big enough to hold the decrypted  
2261 value when it may previously have succeeded.

2262 All functions which use the above convention will explicitly say so.

2263 Cryptographic functions which return output in a variable-length buffer should always return as much  
2264 output as can be computed from what has been passed in to them thus far. As an example, consider a  
2265 session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode

2266 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C\_DecryptUpdate**  
2267 function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether  
2268 the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at  
2269 least 8 bytes. Hence the call to **C\_DecryptUpdate** should return 0 bytes of plaintext. If a single  
2270 additional byte of ciphertext is supplied by a subsequent call to **C\_DecryptUpdate**, then that call should  
2271 return 8 bytes of plaintext (one full DES block).

## 2272 **5.3 Disclaimer concerning sample code**

2273 For the remainder of this section, we enumerate the various functions defined in Cryptoki. Most functions  
2274 will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will  
2275 frequently be somewhat incomplete. In particular, sample code will generally ignore possible error  
2276 returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

## 2277 **5.4 General-purpose functions**

2278 Cryptoki provides the following general-purpose functions:

### 2279 **5.4.1 C\_Initialize**

```
2280 CK_DECLARE_FUNCTION(CK_RV, C_Initialize) {  
2281     CK_VOID_PTR pInitArgs  
2282 };
```

2283 **C\_Initialize** initializes the Cryptoki library. *pInitArgs* either has the value `NULL_PTR` or points to a  
2284 **CK\_C\_INITIALIZE\_ARGS** structure containing information on how the library should deal with multi-  
2285 threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously,  
2286 it can generally supply the value `NULL_PTR` to **C\_Initialize** (the consequences of supplying this value will  
2287 be explained below).

2288 If *pInitArgs* is non-`NULL_PTR`, **C\_Initialize** should cast it to a **CK\_C\_INITIALIZE\_ARGS\_PTR** and then  
2289 dereference the resulting pointer to obtain the **CK\_C\_INITIALIZE\_ARGS** fields *CreateMutex*,  
2290 *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of  
2291 *pReserved* thereby obtained MUST be `NULL_PTR`; if it's not, then **C\_Initialize** should return with the  
2292 value `CKR_ARGUMENTS_BAD`.

2293 If the **CKF\_LIBRARY\_CANT\_CREATE\_OS\_THREADS** flag in the *flags* field is set, that indicates that  
2294 application threads which are executing calls to the Cryptoki library are not permitted to use the native  
2295 operation system calls to spawn off new threads. In other words, the library's code may not create its  
2296 own threads. If the library is unable to function properly under this restriction, **C\_Initialize** should return  
2297 with the value `CKR_NEED_TO_CREATE_THREADS`.

2298 A call to **C\_Initialize** specifies one of four different ways to support multi-threaded access via the value of  
2299 the **CKF\_OS\_LOCKING\_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*,  
2300 *LockMutex*, and *UnlockMutex* function pointer fields:

- 2301 1. If the flag *isn't* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value  
2302 `NULL_PTR`), that means that the application *won't* be accessing the Cryptoki library from multiple  
2303 threads simultaneously.
- 2304 2. If the flag *is* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value  
2305 `NULL_PTR`), that means that the application *will* be performing multi-threaded Cryptoki access, and  
2306 the library needs to use the native operating system primitives to ensure safe multi-threaded access.  
2307 If the library is unable to do this, **C\_Initialize** should return with the value `CKR_CANT_LOCK`.
- 2308 3. If the flag *isn't* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-`NULL_PTR`  
2309 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the  
2310 library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded  
2311 access. If the library is unable to do this, **C\_Initialize** should return with the value  
2312 `CKR_CANT_LOCK`.

2313     4. If the flag *is* set, and the function pointer fields are supplied (*i.e.*, they all have non-NUL\_PTR  
 2314       values), that means that the application *will* be performing multi-threaded Cryptoki access, and the  
 2315       library needs to use either the native operating system primitives or the supplied function pointers for  
 2316       mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C\_Initialize**  
 2317       should return with the value CKR\_CANT\_LOCK.  
 2318     If some, but not all, of the supplied function pointers to **C\_Initialize** are non-NUL\_PTR, then **C\_Initialize**  
 2319       should return with the value CKR\_ARGUMENTS\_BAD.  
 2320     A call to **C\_Initialize** with *pInitArgs* set to NULL\_PTR is treated like a call to **C\_Initialize** with *pInitArgs*  
 2321       pointing to a **CK\_C\_INITIALIZE\_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*,  
 2322       *UnlockMutex*, and *pReserved* fields set to NULL\_PTR, and has the *flags* field set to 0.  
 2323     **C\_Initialize** should be the first Cryptoki call made by an application, except for calls to  
 2324       **C\_GetFunctionList**, **C\_GetInterfaceList**, or **C\_GetInterface**. What this function actually does is  
 2325       implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or  
 2326       any other resources it requires.  
 2327     If several applications are using Cryptoki, each one should call **C\_Initialize**. Every call to **C\_Initialize**  
 2328       should (eventually) be succeeded by a single call to **C\_Finalize**. See [**PKCS11-UG**] for further details.  
 2329     Return values: CKR\_ARGUMENTS\_BAD, CKR\_CANT\_LOCK,  
 2330       CKR\_CRYPTOKI\_ALREADY\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 2331       CKR\_HOST\_MEMORY, CKR\_NEED\_TO\_CREATE\_THREADS, CKR\_OK.  
 2332     Example: see **C\_GetInfo**.

## 2333     5.4.2 C\_Finalize

```

2334     CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (
2335       CK_VOID_PTR pReserved
2336 );

```

2337     **C\_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the  
 2338       last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for  
 2339       this version, it should be set to NULL\_PTR (if **C\_Finalize** is called with a non-NUL\_PTR value for  
 2340       *pReserved*, it should return the value CKR\_ARGUMENTS\_BAD).

2341     If several applications are using Cryptoki, each one should call **C\_Finalize**. Each application's call to  
 2342       **C\_Finalize** should be preceded by a single call to **C\_Initialize**; in between the two calls, an application  
 2343       can make calls to other Cryptoki functions. See [**PKCS11-UG**] for further details.

2344     *Despite the fact that the parameters supplied to **C\_Initialize** can in general allow for safe multi-threaded*  
 2345       *access to a Cryptoki library, the behavior of **C\_Finalize** is nevertheless undefined if it is called by an*  
 2346       *application while other threads of the application are making Cryptoki calls. The exception to this*  
 2347       *exceptional behavior of **C\_Finalize** occurs when a thread calls **C\_Finalize** while another of the*  
 2348       *application's threads is blocking on Cryptoki's **C\_WaitForSlotEvent** function. When this happens, the*  
 2349       *blocked thread becomes unblocked and returns the value CKR\_CRYPTOKI\_NOT\_INITIALIZED. See*  
 2350       ***C\_WaitForSlotEvent** for more information.*

2351     Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
 2352       CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

2353     Example: see **C\_GetInfo**.

## 2354     5.4.3 C\_GetInfo

```

2355     CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (
2356       CK_INFO_PTR pInfo
2357 );

```

2358     **C\_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the  
 2359       information.

2360 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
2361 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

2362 Example:

```
2363 CK_INFO info;
2364 CK_RV rv;
2365 CK_C_INITIALIZE_ARGS InitArgs;
2366
2367 InitArgs.CreateMutex = &MyCreateMutex;
2368 InitArgs.DestroyMutex = &MyDestroyMutex;
2369 InitArgs.LockMutex = &MyLockMutex;
2370 InitArgs.UnlockMutex = &MyUnlockMutex;
2371 InitArgs.flags = CKF_OS_LOCKING_OK;
2372 InitArgs.pReserved = NULL_PTR;
2373
2374 rv = C_Initialize((CK_VOID_PTR)&InitArgs);
2375 assert(rv == CKR_OK);
2376
2377 rv = C_GetInfo(&info);
2378 assert(rv == CKR_OK);
2379 if(info.cryptokiVersion.major == 2) {
2380     /* Do lots of interesting cryptographic things with the token */
2381     .
2382     .
2383 }
2384
2385 rv = C_Finalize(NULL_PTR);
2386 assert(rv == CKR_OK);
```

#### 2387 **5.4.4 C\_GetFunctionList**

```
2388 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList) (
2389     CK_FUNCTION_LIST_PTR_PTP pFunctionList
2390 );
```

2391 **C\_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*  
2392 points to a value which will receive a pointer to the library's **CK\_FUNCTION\_LIST** structure, which in turn  
2393 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*  
2394 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.*  
2395 Whether or not this is the case, no attempt should be made to write to this memory.

2396 **C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an  
2397 application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications  
2398 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2399 Return values: CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
2400 CKR\_HOST\_MEMORY, CKR\_OK.

2401 Example:

```
2402 CK_FUNCTION_LIST_PTR pFunctionList;
```

```

2403 CK_C_Initialize pC_Initialize;
2404 CK_RV rv;
2405
2406 /* It's OK to call C_GetFunctionList before calling C_Initialize */
2407 rv = C_GetFunctionList(&pFunctionList);
2408 assert(rv == CKR_OK);
2409 pC_Initialize = pFunctionList -> C_Initialize;
2410
2411 /* Call the C_Initialize function in the library */
2412 rv = (*pC_Initialize)(NULL_PTR);

```

## 2413 5.4.5 C\_GetInterfaceList

```

2414 CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList) (
2415     CK_INTERFACE_PTR      pInterfaceList,
2416     CK ULONG_PTR          pulCount
2417 );

```

2418 **C\_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points  
 2419 to the location that receives the number of interfaces.

2420 There are two ways for an application to call **C\_GetInterfaceList**:

- 2421 1. If *pInterfaceList* is NULL\_PTR, then all that **C\_GetInterfaceList** does is return (in \**pulCount*) the  
 2422 number of interfaces, without actually returning a list of interfaces. The contents of \**pulCount* on  
 2423 entry to **C\_GetInterfaceList** has no meaning in this case, and the call returns the value CKR\_OK.
- 2424 2. If *pInterfaceList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of  
 2425 **CK\_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to  
 2426 hold the list of interfaces, then the list is returned in it, and CKR\_OK is returned. If not, then the call  
 2427 to **C\_GetInterfaceList** returns the value CKR\_BUFFER\_TOO\_SMALL. In either case, the value  
 2428 \**pulCount* is set to hold the number of interfaces.

2429 Because **C\_GetInterfaceList** does not allocate any space of its own, an application will often call  
 2430 **C\_GetInterfaceList** twice. However, this behavior is by no means required.

2431 **C\_GetInterfaceList** obtains (in \**pFunctionList* of each interface) a pointer to the Cryptoki library's list of  
 2432 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*  
 2433 *library, and which may or may not be writable*. Whether or not this is the case, no attempt should be  
 2434 made to write to this memory. The same caveat applies to the interface names returned.

2435

2436 **C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an  
 2437 application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications  
 2438 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2439 Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED,  
 2440 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

2441 Example:

```

2442 CK ULONG ulCount=0;
2443 CK_INTERFACE_PTR interfaceList=NULL;
2444 CK_RV rv;
2445 int I;
2446
2447 /* get number of interfaces */

```

```

2448 rv = C_GetInterfaceList(NULL, &ulCount);
2449 if (rv == CKR_OK) {
2450     /* get copy of interfaces */
2451     interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
2452     rv = C_GetInterfaceList(interfaceList,&ulCount);
2453     for(i=0;i<ulCount;i++) {
2454         printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2455             interfaceList[i].pInterfaceName,
2456             ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
2457             ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
2458             interfaceList[i].pFunctionList,
2459             interfaceList[i].flags);
2460     }
2461 }
2462

```

## 2463 5.4.6 C\_GetInterface

```

2464 CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) (
2465     CK_UTF8CHAR_PTR          pInterfaceName,
2466     CK_VERSION_PTR           pVersion,
2467     CK_INTERFACE_PTR_PTR    ppInterface,
2468     CK_FLAGS                 flags
2469 );

```

2470 **C\_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies  
 2471 the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that  
 2472 receives the interface, *flags* specifies the required interface flags.

2473 There are multiple ways for an application to specify a particular interface when calling **C\_GetInterface**:

- 2474 1. If *pInterfaceName* is not NULL\_PTR, the name of the interface returned must match. If *pInterfaceName* is NULL\_PTR, the cryptoki library can return a default interface of its choice
- 2475 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
- 2476 3. 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

2477 **C\_GetInterface** obtains (in \**pFunctionList* of each interface) a pointer to the Cryptoki library's list of  
 2478 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki  
 2479 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.

2480 **C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an  
 2481 application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications  
 2482 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2483 Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED,  
 2484 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

2485 Example:

```

2490 CK_INTERFACE_PTR interface;
2491 CK_RV rv;

```

```

2492 CK_VERSION version;
2493 CK_FLAGS flags=CKF_INTERFACE_FORK_SAFE;
2494
2495 /* get default interface */
2496 rv = C_GetInterface(NULL,NULL,&interface,flags);
2497 if (rv == CKR_OK) {
2498     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2499            interface->pInterfaceName,
2500            ((CK_VERSION *)interface->pFunctionList)->major,
2501            ((CK_VERSION *)interface->pFunctionList)->minor,
2502            interface->pFunctionList,
2503            interface->flags);
2504 }
2505
2506 /* get default standard interface */
2507 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
2508 if (rv == CKR_OK) {
2509     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2510            interface->pInterfaceName,
2511            ((CK_VERSION *)interface->pFunctionList)->major,
2512            ((CK_VERSION *)interface->pFunctionList)->minor,
2513            interface->pFunctionList,
2514            interface->flags);
2515 }
2516
2517 /* get specific standard version interface */
2518 version.major=3;
2519 version.minor=0;
2520 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
2521 if (rv == CKR_OK) {
2522     CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
2523
2524     /* ... use the new functions */
2525     pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
2526                           pUsername,ulUsernameLen);
2527 }
2528
2529 /* get specific vendor version interface */
2530 version.major=1;
2531 version.minor=0;
2532 rv = C_GetInterface((CK_UTF8CHAR_PTR)
2533                      "Vendor VendorName",&version,&interface,flags);
2534 if (rv == CKR_OK) {

```

```

2535     CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
2536
2537     /* ... use vendor specific functions */
2538     pkcs11->C_VendorFunction1(param1,param2,param3);
2539 }
2540

```

## 2541 5.5 Slot and token management functions

2542 Cryptoki provides the following functions for slot and token management:

### 2543 5.5.1 C\_GetSlotList

```

2544 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotList) (
2545     CK_BBOOL tokenPresent,
2546     CK_SLOT_ID_PTR pSlotList,
2547     CK ULONG_PTR pulCount
2548 );

```

2549 **C\_GetSlotList** is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list  
2550 obtained includes only those slots with a token present (CK\_TRUE), or all slots (CK\_FALSE); *pulCount*  
2551 points to the location that receives the number of slots.

2552 There are two ways for an application to call **C\_GetSlotList**:

- 2553 1. If *pSlotList* is NULL\_PTR, then all that **C\_GetSlotList** does is return (in \**pulCount*) the number of  
2554 slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on  
2555 entry to **C\_GetSlotList** has no meaning in this case, and the call returns the value CKR\_OK.
- 2556 2. If *pSlotList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of CK\_SLOT\_ID  
2557 elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots,  
2558 then the list is returned in it, and CKR\_OK is returned. If not, then the call to **C\_GetSlotList** returns  
2559 the value CKR\_BUFFER\_TOO\_SMALL. In either case, the value \**pulCount* is set to hold the number  
2560 of slots.

2561 Because **C\_GetSlotList** does not allocate any space of its own, an application will often call  
2562 **C\_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots  
2563 with a token present, then the number of such slots can (unfortunately) change between when the  
2564 application asks for how many such slots there are and when the application asks for the slots  
2565 themselves). However, multiple calls to **C\_GetSlotList** are by no means required.

2566 All slots which **C\_GetSlotList** reports MUST be able to be queried as valid slots by **C\_GetSlotInfo**.  
2567 Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that  
2568 **C\_GetSlotList**, for list length prediction (NULL *pSlotList* argument) is called. If an application calls  
2569 **C\_GetSlotList** with a non-NULL *pSlotList*, and then the user adds or removes a hardware device, the  
2570 changed slot list will only be visible and effective if **C\_GetSlotList** is called again with NULL. Even if **C**  
2571 **GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be  
2572 successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11  
2573 compliant libraries, it may be necessary to successfully call **C\_Initialize** or to restart the entire system.

2574

2575 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
2576 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
2577 CKR\_HOST\_MEMORY, CKR\_OK.

2578 Example:

```

2579 CK ULONG ulSlotCount, ulSlotWithTokenCount;
2580 CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;

```

```

2581 CK_RV rv;
2582
2583 /* Get list of all slots */
2584 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
2585 if (rv == CKR_OK) {
2586     pSlotList =
2587         (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
2588     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
2589     if (rv == CKR_OK) {
2590         /* Now use that list of all slots */
2591         .
2592         .
2593     }
2594
2595     free(pSlotList);
2596 }
2597
2598 /* Get list of all slots with a token present */
2599 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
2600 ulSlotWithTokenCount = 0;
2601 while (1) {
2602     rv = C_GetSlotList(
2603         CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
2604     if (rv != CKR_BUFFER_TOO_SMALL)
2605         break;
2606     pSlotWithTokenList = realloc(
2607         pSlotWithTokenList,
2608         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
2609 }
2610
2611 if (rv == CKR_OK) {
2612     /* Now use that list of all slots with a token present */
2613     .
2614     .
2615 }
2616
2617 free(pSlotWithTokenList);

```

## 2618 5.5.2 C\_GetSlotInfo

```

2619 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo) (
2620     CK_SLOT_ID slotID,
2621     CK_SLOT_INFO_PTR pInfo
2622 );

```

2623 **C\_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo*  
2624 points to the location that receives the slot information.  
2625 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
2626 CKR\_DEVICE\_ERROR, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY,  
2627 CKR\_OK, CKR\_SLOT\_ID\_INVALID.  
2628 Example: see **C\_GetTokenInfo**.

### 2629 5.5.3 C\_GetTokenInfo

```
2630 CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo) (
2631     CK_SLOT_ID slotID,
2632     CK_TOKEN_INFO_PTR pInfo
2633 );
```

2634 **C\_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the  
2635 token's slot; *pInfo* points to the location that receives the token information.

2636 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
2637 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
2638 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT,  
2639 CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_ARGUMENTS\_BAD.

2640 Example:

```
2641 CK ULONG ulCount;
2642 CK_SLOT_ID_PTR pSlotList;
2643 CK_SLOT_INFO slotInfo;
2644 CK_TOKEN_INFO tokenInfo;
2645 CK_RV rv;
2646
2647 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulCount);
2648 if ((rv == CKR_OK) && (ulCount > 0)) {
2649     pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
2650     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulCount);
2651     assert(rv == CKR_OK);
2652
2653     /* Get slot information for first slot */
2654     rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
2655     assert(rv == CKR_OK);
2656
2657     /* Get token information for first slot */
2658     rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
2659     if (rv == CKR_TOKEN_NOT_PRESENT) {
2660         .
2661         .
2662     }
2663     .
2664     .
2665     free(pSlotList);
```

2666 }

## 2667 5.5.4 C\_WaitForSlotEvent

```
2668 CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent) (
2669     CK_FLAGS flags,
2670     CK_SLOT_ID_PTR pSlot,
2671     CK_VOID_PTR pReserved
2672 );
```

2673 **C\_WaitForSlotEvent** waits for a slot event, such as token insertion or token removal, to occur. *flags* determines whether or not the **C\_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot* points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved for future versions; for this version of Cryptoki, it should be `NULL_PTR`.

2677 At present, the only flag defined for use in the *flags* argument is `CKF_DONT_BLOCK`:

2678 Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any  
2679 unrecognized events involving that slot have occurred. When an application initially calls **C\_Initialize**,  
2680 every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in  
2681 which the event occurred is set.

2682 If **C\_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag set in the *flags* argument, and some  
2683 slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the  
2684 location pointed to by *pSlot*. If more than one slot's event flag is set at the time of the call, one such slot  
2685 is chosen by the library to have its event flag cleared and to have its slot ID returned.

2686 If **C\_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag set in the *flags* argument, and no  
2687 slot's event flag is set, then the call returns with the value `CKR_NO_EVENT`. In this case, the contents of  
2688 the location pointed to by *pSlot* when **C\_WaitForSlotEvent** are undefined.

2689 If **C\_WaitForSlotEvent** is called with the `CKF_DONT_BLOCK` flag clear in the *flags* argument, then the  
2690 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,  
2691 **C\_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has  
2692 a **C\_WaitForSlotEvent** call blocking when another thread of that application calls **C\_Finalize**, the  
2693 **C\_WaitForSlotEvent** call returns with the value `CKR_CRYPTOKI_NOT_INITIALIZED`.

2694 Although the parameters supplied to **C\_Initialize** can in general allow for safe multi-threaded access to a  
2695 Cryptoki library, **C\_WaitForSlotEvent** is exceptional in that the behavior of Cryptoki is undefined if  
2696 multiple threads of a single application make simultaneous calls to **C\_WaitForSlotEvent**.

2697 Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`,  
2698 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_NO_EVENT`,  
2699 `CKR_OK`.

2700 Example:

```
2701 CK_FLAGS flags = 0;
2702 CK_SLOT_ID slotID;
2703 CK_SLOT_INFO slotInfo;
2704 CK_RV rv;
2705 .
2706 .
2707 /* Block and wait for a slot event */
2708 rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);
2709 assert(rv == CKR_OK);
2710
2711 /* See what's up with that slot */
```

```
2712     rv = C_GetSlotInfo(slotID, &slotInfo);  
2713     assert(rv == CKR_OK);  
2714 }
```

## 2715 5.5.5 C\_GetMechanismList

```
2716 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList) (  
2717     CK_SLOT_ID slotID,  
2718     CK_MECHANISM_TYPE_PTR pMechanismList,  
2719     CK ULONG_PTR pulCount  
2720 );
```

2721 **C\_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID  
2722 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

2723 There are two ways for an application to call **C\_GetMechanismList**:

- 2724 1. If *pMechanismList* is NULL\_PTR, then all that **C\_GetMechanismList** does is return (in \**pulCount*)  
2725 the number of mechanisms, without actually returning a list of mechanisms. The contents of  
2726 \**pulCount* on entry to **C\_GetMechanismList** has no meaning in this case, and the call returns the  
2727 value CKR\_OK.
- 2728 2. If *pMechanismList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of  
2729 **CK\_MECHANISM\_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large  
2730 enough to hold the list of mechanisms, then the list is returned in it, and CKR\_OK is returned. If not,  
2731 then the call to **C\_GetMechanismList** returns the value CKR\_BUFFER\_TOO\_SMALL. In either  
2732 case, the value \**pulCount* is set to hold the number of mechanisms.

2733 Because **C\_GetMechanismList** does not allocate any space of its own, an application will often call  
2734 **C\_GetMechanismList** twice. However, this behavior is by no means required.

2735 Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
2736 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
2737 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
2738 CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED,  
2739 CKR\_ARGUMENTS\_BAD.

2740 Example:

```
2741 CK_SLOT_ID slotID;  
2742 CK ULONG ulCount;  
2743 CK_MECHANISM_TYPE_PTR pMechanismList;  
2744 CK_RV rv;  
2745  
2746 .  
2747 .  
2748 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);  
2749 if ((rv == CKR_OK) && (ulCount > 0)) {  
2750     pMechanismList =  
2751         (CK_MECHANISM_TYPE_PTR)  
2752         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));  
2753     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);  
2754     if (rv == CKR_OK) {  
2755         .  
2756     }
```

```
2757     }
2758     free(pMechanismList);
2759 }
```

## 2760 5.5.6 C\_GetMechanismInfo

```
2761 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (
2762     CK_SLOT_ID slotID,
2763     CK_MECHANISM_TYPE type,
2764     CK_MECHANISM_INFO_PTR pInfo
2765 );
```

2766 **C\_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token.  
2767 *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives  
2768 the mechanism information.

2769 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
2770 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
2771 CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_OK, CKR\_SLOT\_ID\_INVALID,  
2772 CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_ARGUMENTS\_BAD.

2773 Example:

```
2774 CK_SLOT_ID slotID;
2775 CK_MECHANISM_INFO info;
2776 CK_RV rv;
2777 .
2778 .
2779 .
2780 /* Get information about the CKM_MD2 mechanism for this token */
2781 rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
2782 if (rv == CKR_OK) {
2783     if (info.flags & CKF_DIGEST) {
2784         .
2785         .
2786     }
2787 }
```

## 2788 5.5.7 C\_InitToken

```
2789 CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
2790     CK_SLOT_ID slotID,
2791     CK_UTF8CHAR_PTR pPin,
2792     CK_ULONG ulPinLen,
2793     CK_UTF8CHAR_PTR pLabel
2794 );
```

2795 **C\_InitToken** initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN  
2796 (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-  
2797 byte label of the token (which MUST be padded with blank characters, and which MUST *not* be null-

2798 terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may  
 2799 impose subset restrictions.  
 2800 If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the  
 2801 initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the  
 2802 existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after  
 2803 the function completes successfully. If the SO PIN is lost, then the card MUST be reinitialized using a  
 2804 mechanism outside the scope of this standard. The **CKF\_TOKEN\_INITIALIZED** flag in the  
 2805 **CK\_TOKEN\_INFO** structure indicates the action that will result from calling **C\_InitToken**. If set, the token  
 2806 will be reinitialized, and the client MUST supply the existing SO password in *pPin*.  
 2807 When a token is initialized, all objects that can be destroyed are destroyed (i.e., all except for  
 2808 “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled  
 2809 until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created,  
 2810 and attributes of some objects may be set to default values.  
 2811 If the token has a “protected authentication path”, as indicated by the  
 2812 **CKF\_PROTECTED\_AUTHENTICATION\_PATH** flag in its **CK\_TOKEN\_INFO** being set, then that means  
 2813 that there is some way for a user to be authenticated to the token without having the application send a  
 2814 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the  
 2815 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*  
 2816 parameter to **C\_InitToken** should be **NULL\_PTR**. During the execution of **C\_InitToken**, the SO’s PIN will  
 2817 be entered through the protected authentication path.  
 2818 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether  
 2819 or not **C\_InitToken** can be used to initialize the token.  
 2820 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a  
 2821 call to **C\_InitToken** is made under such circumstances, the call fails with error **CKR\_SESSION\_EXISTS**.  
 2822 Unfortunately, it may happen when **C\_InitToken** is called that some other application *does* have an open  
 2823 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other  
 2824 applications using the token. If this is the case, then the consequences of the **C\_InitToken** call are  
 2825 undefined.  
 2826 The **C\_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,  
 2827 an initialization mechanism outside the scope of Cryptoki MUST be employed. The definition of “complex  
 2828 token” is product specific.  
 2829 Return values: **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_DEVICE\_ERROR**, **CKR\_DEVICE\_MEMORY**,  
 2830 **CKR\_DEVICE\_REMOVED**, **CKR\_FUNCTION\_CANCELED**, **CKR\_FUNCTION\_FAILED**,  
 2831 **CKR\_GENERAL\_ERROR**, **CKR\_HOST\_MEMORY**, **CKR\_OK**, **CKR\_PIN\_INCORRECT**,  
 2832 **CKR\_PIN\_LOCKED**, **CKR\_SESSION\_EXISTS**, **CKR\_SLOT\_ID\_INVALID**,  
 2833 **CKR\_TOKEN\_NOT\_PRESENT**, **CKR\_TOKEN\_NOT\_RECOGNIZED**,  
 2834 **CKR\_TOKEN\_WRITE\_PROTECTED**, **CKR\_ARGUMENTS\_BAD**.  
 2835 Example:  

```

2836 CK_SLOT_ID slotID;
2837 CK_UTF8CHAR pin[] = {"MyPIN"};
2838 CK_UTF8CHAR label[32];
2839 CK_RV rv;
2840
2841 .
2842 .
2843 memset(label, ' ', sizeof(label));
2844 memcpy(label, "My first token", strlen("My first token"));
2845 rv = C_InitToken(slotID, pin, strlen(pin), label);
2846 if (rv == CKR_OK) {
2847 .

```

```
2848 .
2849 }
```

## 2850 5.5.8 C\_InitPIN

```
2851 CK_DECLARE_FUNCTION(CK_RV, C_InitPIN) (
2852     CK_SESSION_HANDLE hSession,
2853     CK_UTF8CHAR_PTR pPin,
2854     CK ULONG ulPinLen
2855 );
```

2856 **C\_InitPIN** initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal  
2857 user's PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any  
2858 valid UTF8 character, but the token may impose subset restrictions.

2859 **C\_InitPIN** can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any  
2860 other state fails with error CKR\_USER\_NOT\_LOGGED\_IN.

2861 If the token has a "protected authentication path", as indicated by the  
2862 CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its **CK\_TOKEN\_INFO** being set, then that means  
2863 that there is some way for a user to be authenticated to the token without having to send a PIN through  
2864 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or  
2865 on the slot device. To initialize the normal user's PIN on a token with such a protected authentication  
2866 path, the *pPin* parameter to **C\_InitPIN** should be NULL\_PTR. During the execution of **C\_InitPIN**, the SO  
2867 will enter the new PIN through the protected authentication path.

2868 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether  
2869 or not **C\_InitPIN** can be used to initialize the normal user's token access.

2870 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
2871 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
2872 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_INVALID,  
2873 CKR\_PIN\_LEN\_RANGE, CKR\_SESSION\_CLOSED, CKR\_SESSION\_READ\_ONLY,  
2874 CKR\_SESSION\_HANDLE\_INVALID, CKR\_TOKEN\_WRITE\_PROTECTED,  
2875 CKR\_USER\_NOT\_LOGGED\_IN, CKR\_ARGUMENTS\_BAD.

2876 Example:

```
2877 CK_SESSION_HANDLE hSession;
2878 CK_UTF8CHAR newPin[] = {"NewPIN"};
2879 CK_RV rv;
2880
2881 rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);
2882 if (rv == CKR_OK) {
2883 .
2884 .
2885 }
```

## 2886 5.5.9 C\_SetPIN

```
2887 CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (
2888     CK_SESSION_HANDLE hSession,
2889     CK_UTF8CHAR_PTR pOldPin,
2890     CK ULONG ulOldLen,
2891     CK_UTF8CHAR_PTR pNewPin,
2892     CK ULONG ulNewLen
2893 );
```

2894   **C\_SetPIN** modifies the PIN of the user that is currently logged in, or the CKU\_USER PIN if the session is  
2895   not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in  
2896   bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This  
2897   standard allows PIN values to contain any valid UTF8 character, but the token may impose subset  
2898   restrictions.

2899   **C\_SetPIN** can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User  
2900   Functions" state. An attempt to call it from a session in any other state fails with error  
2901   CKR\_SESSION\_READ\_ONLY.

2902   If the token has a "protected authentication path", as indicated by the  
2903   CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its **CK\_TOKEN\_INFO** being set, then that means  
2904   that there is some way for a user to be authenticated to the token without having to send a PIN through  
2905   the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or  
2906   on the slot device. To modify the current user's PIN on a token with such a protected authentication path,  
2907   the *pOldPin* and *pNewPin* parameters to **C\_SetPIN** should be NULL\_PTR. During the execution of  
2908   **C\_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication  
2909   path. It is not specified how the PIN pad should be used to enter two PINs; this varies.

2910   If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether  
2911   or not **C\_SetPIN** can be used to modify the current user's PIN.

2912   Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
2913   CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
2914   CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_INCORRECT,  
2915   CKR\_PIN\_INVALID, CKR\_PIN\_LEN\_RANGE, CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED,  
2916   CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY,  
2917   CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_ARGUMENTS\_BAD.

2918   Example:

```
2919 CK_SESSION_HANDLE hSession;
2920 CK_UTF8CHAR oldPin[] = {"OldPIN"};
2921 CK_UTF8CHAR newPin[] = {"NewPIN"};
2922 CK_RV rv;
2923
2924 rv = C_SetPIN(
2925     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);
2926 if (rv == CKR_OK) {
2927     .
2928     .
2929 }
```

## 2930   5.6 Session management functions

2931   A typical application might perform the following series of steps to make use of a token (note that there  
2932   are other reasonable sequences of events that an application might perform):

- 2933   1. Select a token.
- 2934   2. Make one or more calls to **C\_OpenSession** to obtain one or more sessions with the token.
- 2935   3. Call **C\_Login** to log the user into the token. Since all sessions an application has with a token have a  
2936   shared login state, **C\_Login** only needs to be called for one of the sessions.
- 2937   4. Perform cryptographic operations using the sessions with the token.
- 2938   5. Call **C\_CloseSession** once for each session that the application has with the token, or call  
2939   **C\_CloseAllSessions** to close all the application's sessions simultaneously.

2940 As has been observed, an application may have concurrent sessions with more than one token. It is also  
2941 possible for a token to have concurrent sessions with more than one application.  
2942 Cryptoki provides the following functions for session management:

### 2943 5.6.1 C\_OpenSession

```
2944 CK_DECLARE_FUNCTION(CK_RV, C_OpenSession) (
2945     CK_SLOT_ID slotID,
2946     CK_FLAGS flags,
2947     CK_VOID_PTR pApplication,
2948     CK_NOTIFY Notify,
2949     CK_SESSION_HANDLE_PTR phSession
2950 );
```

2951 **C\_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the  
2952 slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to  
2953 the notification callback; *Notify* is the address of the notification callback function (see Section 5.21);  
2954 *phSession* points to the location that receives the handle for the new session.

2955 When opening a session with **C\_OpenSession**, the *flags* parameter consists of the logical OR of zero or  
2956 more bit flags defined in the **CK\_SESSION\_INFO** data type. For legacy reasons, the  
2957 **CKF\_SERIAL\_SESSION** bit MUST always be set; if a call to **C\_OpenSession** does not have this bit set,  
2958 the call should return unsuccessfully with the error code  
2959 **CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED**.

2960 There may be a limit on the number of concurrent sessions an application may have with the token, which  
2961 may depend on whether the session is "read-only" or "read/write". An attempt to open a session which  
2962 does not succeed because there are too many existing sessions of some type should return  
2963 **CKR\_SESSION\_COUNT**.

2964 If the token is write-protected (as indicated in the **CK\_TOKEN\_INFO** structure), then only read-only  
2965 sessions may be opened with it.

2966 If the application calling **C\_OpenSession** already has a R/W SO session open with the token, then any  
2967 attempt to open a R/O session with the token fails with error code  
2968 **CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS** (see [PKCS11-UG] for further details).

2969 The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the  
2970 application does not wish to support callbacks, it should pass a value of **NULL\_PTR** as the *Notify*  
2971 parameter. See Section 5.21 for more information about application callbacks.

2972 Return values: **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_DEVICE\_ERROR**, **CKR\_DEVICE\_MEMORY**,  
2973 **CKR\_DEVICE\_REMOVED**, **CKR\_FUNCTION\_FAILED**, **CKR\_GENERAL\_ERROR**,  
2974 **CKR\_HOST\_MEMORY**, **CKR\_OK**, **CKR\_SESSION\_COUNT**,  
2975 **CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED**, **CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS**,  
2976 **CKR\_SLOT\_ID\_INVALID**, **CKR\_TOKEN\_NOT\_PRESENT**, **CKR\_TOKEN\_NOT\_RECOGNIZED**,  
2977 **CKR\_TOKEN\_WRITE\_PROTECTED**, **CKR\_ARGUMENTS\_BAD**.

2978 Example: see **C\_CloseSession**.

### 2979 5.6.2 C\_CloseSession

```
2980 CK_DECLARE_FUNCTION(CK_RV, C_CloseSession) (
2981     CK_SESSION_HANDLE hSession
2982 );
```

2983 **C\_CloseSession** closes a session between an application and a token. *hSession* is the session's  
2984 handle.

2985 When a session is closed, all session objects created by the session are destroyed automatically, even if  
2986 the application has other sessions "using" the objects (see [PKCS11-UG] for further details).

2987 If this function is successful and it closes the last session between the application and the token, the login  
2988 state of the token for the application returns to public sessions. Any new sessions to the token opened by  
2989 the application will be either R/O Public or R/W Public sessions.  
2990 Depending on the token, when the last open session any application has with the token is closed, the  
2991 token may be "ejected" from its reader (if this capability exists).  
2992 Despite the fact this **C\_CloseSession** is supposed to close a session, the return value  
2993 CKR\_SESSION\_CLOSED is an error return. It actually indicates the (probably somewhat unlikely) event  
2994 that while this function call was executing, another call was made to **C\_CloseSession** to close this  
2995 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and  
2996 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of  
2997 behavior.  
2998 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
2999 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3000 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

3001 Example:

```
3002 CK_SLOT_ID slotID;
3003 CK_BYTE application;
3004 CK_NOTIFY MyNotify;
3005 CK_SESSION_HANDLE hSession;
3006 CK_RV rv;
3007 .
3008 .
3009 .
3010 application = 17;
3011 MyNotify = &EncryptionSessionCallback;
3012 rv = C_OpenSession(
3013     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,
3014             (CK_VOID_PTR) &application, MyNotify,
3015             &hSession);
3016 if (rv == CKR_OK) {
3017 .
3018 .
3019     C_CloseSession(hSession);
3020 }
```

### 3021 **5.6.3 C\_CloseAllSessions**

```
3022 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions) (
3023     CK_SLOT_ID slotID
3024 );
```

3025 **C\_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token's slot.  
3026 When a session is closed, all session objects created by the session are destroyed automatically.  
3027 After successful execution of this function, the login state of the token for the application returns to public  
3028 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W  
3029 Public sessions.  
3030 Depending on the token, when the last open session any application has with the token is closed, the  
3031 token may be "ejected" from its reader (if this capability exists).

3032 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3033 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3034 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT.

3035 Example:

```
3036 CK_SLOT_ID slotID;  
3037 CK_RV rv;  
3038 .  
3039 .  
3040 .  
3041 rv = C_CloseAllSessions(slotID);
```

#### 3042 5.6.4 C\_GetSessionInfo

```
3043 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (  
3044     CK_SESSION_HANDLE hSession,  
3045     CK_SESSION_INFO_PTR pInfo  
3046 );
```

3047 **C\_GetSessionInfo** obtains information about a session. *hSession* is the session's handle; *pInfo* points to  
3048 the location that receives the session information.

3049 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3050 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3051 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3052 CKR\_ARGUMENTS\_BAD.

3053 Example:

```
3054 CK_SESSION_HANDLE hSession;  
3055 CK_SESSION_INFO info;  
3056 CK_RV rv;  
3057 .  
3058 .  
3059 .  
3060 rv = C_GetSessionInfo(hSession, &info);  
3061 if (rv == CKR_OK) {  
3062     if (info.state == CKS_RW_USER_FUNCTIONS) {  
3063         .  
3064         .  
3065     }  
3066     .  
3067     .  
3068 }
```

#### 3069 5.6.5 C\_SessionCancel

```
3070 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (  
3071     CK_SESSION_HANDLE hSession  
3072     CK_FLAGS flags  
3073 );
```

3074   **C\_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*  
3075   indicates the operations to cancel.  
3076   To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical  
3077   bitwise OR of one or more of the bit flags defined in the **CK\_MECHANISM\_INFO** structure.  
3078   If no flags are set, the session state will not be modified and CKR\_OK will be returned.  
3079   If a flag is set for an operation that has not been initialized in the session, the operation flag will be  
3080   ignored and **C\_SessionCancel** will behave as if the operation flag was not set.  
3081   If any of the operations indicated by the *flags* parameter cannot be cancelled,  
3082   CKR\_OPERATION\_CANCEL\_FAILED must be returned. If multiple operation flags were set and  
3083   CKR\_OPERATION\_CANCEL\_FAILED is returned, this function does not provide any information about  
3084   which operation(s) could not be cancelled. If an application desires to know if any single operation could  
3085   not be cancelled, the application should not call **C\_SessionCancel** with multiple flags set.  
3086   If **C\_SessionCancel** is called from an application callback (see Section 5.21), no action will be taken by  
3087   the library and CKR\_FUNCTION\_FAILED must be returned.  
3088   If **C\_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation  
3089   should still be left in an active state. However, it is expected that some Cryptoki implementations may not  
3090   support this and return CKR\_OPERATION\_CANCEL\_FAILED unless flags for both operations are  
3091   provided.  
3092   Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3093   CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3094   CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_CANCEL\_FAILED,  
3095   CKR\_TOKEN\_NOT\_PRESENT.

3096   Example:

```
3097 CK_SESSION_HANDLE hSession;
3098 CK_RV rv;
3099
3100 rv = C_EncryptInit(hSession, &mechanism, hKey);
3101 if (rv != CKR_OK)
3102 {
3103 .
3104 .
3105 }
3106
3107 rv = C_SessionCancel (hSession, CKF_ENCRYPT);
3108 if (rv != CKR_OK)
3109 {
3110 .
3111 .
3112 }
3113
3114 rv = C_EncryptInit(hSession, &mechanism, hKey);
3115 if (rv != CKR_OK)
3116 {
3117 .
3118 .
3119 }
```

3120

3121

3122

3123

3124 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual  
3125 operations. The additional text is highlighted.

3126 **5.6.6 C\_GetOperationState**

```
3127 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (
3128     CK_SESSION_HANDLE hSession,
3129     CK_BYTE_PTR pOperationState,
3130     CK ULONG_PTR pulOperationStateLen
3131 );
```

3132 **C\_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a  
3133 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the  
3134 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

3135 Although the saved state output by **C\_GetOperationState** is not really produced by a "cryptographic  
3136 mechanism", **C\_GetOperationState** nonetheless uses the convention described in Section 5.2 on  
3137 producing output.

3138 Precisely what the "cryptographic operations state" this function saves is varies from token to token;  
3139 however, this state is what is provided as input to **C\_SetOperationState** to restore the cryptographic  
3140 activities of a session.

3141 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is  
3142 using the **CKM\_SHA\_1** mechanism). Suppose that the message digest operation was initialized  
3143 properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The  
3144 application now wants to "save the state" of this digest operation, so that it can continue it later. In this  
3145 particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic  
3146 operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit  
3147 internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data  
3148 indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken  
3149 together, these three pieces of information suffice to continue the current hashing operation at a later  
3150 time.

3151 Consider next a session which is performing an encryption operation with DES (a block cipher with a  
3152 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM\_DES\_CBC**  
3153 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been  
3154 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already  
3155 been produced and output. In this case, the cryptographic operations state of the session most likely  
3156 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-  
3157 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some  
3158 administrative data indicating that this saved state comes from a session which was performing DES  
3159 encryption in CBC mode; and possibly the DES key being used for encryption (see **C\_SetOperationState**  
3160 for more information on whether or not the key is present in the saved state).

3161 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the  
3162 cryptographic operations state of the session will contain all the necessary information to restore both  
3163 operations.

3164 An attempt to save the cryptographic operations state of a session which does not currently have some  
3165 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message  
3166 recovery, verification without message recovery, or some legal combination of two of these) should fail  
3167 with the error **CKR\_OPERATION\_NOT\_INITIALIZED**.

3168 An attempt to save the cryptographic operations state of a session which is performing an appropriate  
3169 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain  
3170 necessary state information and/or key information can't leave the token, for example) should fail with the  
3171 error **CKR\_STATE\_UNSAVEABLE**.

3172 Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
3173 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
3174 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
3175 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3176 CKR\_STATE\_UNSAVEABLE, CKR\_ARGUMENTS\_BAD.  
3177 Example: see **C\_SetOperationState**.

3178 **5.6.7 C\_SetOperationState**

```
3179 CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState) (
3180     CK_SESSION_HANDLE hSession,
3181     CK_BYTE_PTR pOperationState,
3182     CK ULONG ulOperationStateLen,
3183     CK_OBJECT_HANDLE hEncryptionKey,
3184     CK_OBJECT_HANDLE hAuthenticationKey
3185 );
```

3186 **C\_SetOperationState** restores the cryptographic operations state of a session from a string of bytes  
3187 obtained with **C\_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the  
3188 location holding the saved state; *ulOperationStateLen* holds the length of the saved state;  
3189 *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption  
3190 operation in the restored session (or 0 if no encryption or decryption key is needed, either because no  
3191 such operation is ongoing in the stored session or because all the necessary key information is present in  
3192 the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing  
3193 signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either  
3194 because no such operation is ongoing in the stored session or because all the necessary key information  
3195 is present in the saved state).

3196 The state need not have been obtained from the same session (the "source session") as it is being  
3197 restored to (the "destination session"). However, the source session and destination session should have  
3198 a common session state (e.g., CKS\_RW\_USER\_FUNCTIONS), and should be with a common token.  
3199 There is also no guarantee that cryptographic operations state may be carried across logins, or across  
3200 different Cryptoki implementations.

3201 If **C\_SetOperationState** is supplied with alleged saved cryptographic operations state which it can  
3202 determine is not valid saved state (or is cryptographic operations state from a session with a different  
3203 session state, or is cryptographic operations state from a different token), it fails with the error  
3204 CKR\_SAVED\_STATE\_INVALID.

3205 Saved state obtained from calls to **C\_GetOperationState** may or may not contain information about keys  
3206 in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing  
3207 encryption or decryption operation, and the key in use for the operation is not saved in the state, then it  
3208 MUST be supplied to **C\_SetOperationState** in the *hEncryptionKey* argument. If it is not, then  
3209 **C\_SetOperationState** will fail and return the error CKR\_KEY\_NEEDED. If the key in use for the  
3210 operation is saved in the state, then it can be supplied in the *hEncryptionKey* argument, but this is not  
3211 required.

3212 Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification  
3213 operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to  
3214 **C\_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C\_SetOperationState** will  
3215 fail with the error CKR\_KEY\_NEEDED. If the key in use for the operation is saved in the state, then it can  
3216 be supplied in the *hAuthenticationKey* argument, but this is not required.

3217 If an *irrelevant* key is supplied to **C\_SetOperationState** call (e.g., a nonzero key handle is submitted in  
3218 the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an  
3219 ongoing encryption or decryption operation, then **C\_SetOperationState** fails with the error  
3220 CKR\_KEY\_NOT\_NEEDED.

3221 If a key is supplied as an argument to **C\_SetOperationState**, and **C\_SetOperationState** can somehow  
3222 detect that this key was not the key being used in the source session for the supplied cryptographic

3223 operations state (it may be able to detect this if the key or a hash of the key is present in the saved state,  
3224 for example), then **C\_SetOperationState** fails with the error CKR\_KEY\_CHANGED.

3225 An application can look at the **CKF\_RESTORE\_KEY\_NOT\_NEEDED** flag in the flags field of the  
3226 **CK\_TOKEN\_INFO** field for a token to determine whether or not it needs to supply key handles to  
3227 **C\_SetOperationState** calls. If this flag is true, then a call to **C\_SetOperationState** never needs a key  
3228 handle to be supplied to it. If this flag is false, then at least some of the time, **C\_SetOperationState**  
3229 requires a key handle, and so the application should probably always pass in any relevant key handles  
3230 when restoring cryptographic operations state to a session.

3231 **C\_SetOperationState** can successfully restore cryptographic operations state to a session even if that  
3232 session has active cryptographic or object search operations when **C\_SetOperationState** is called (the  
3233 ongoing operations are abruptly cancelled).

3234 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3235 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3236 CKR\_HOST\_MEMORY, CKR\_KEY\_CHANGED, CKR\_KEY\_NEEDED, CKR\_KEY\_NOT\_NEEDED,  
3237 CKR\_OK, CKR\_SAVED\_STATE\_INVALID, CKR\_SESSION\_CLOSED,  
3238 CKR\_SESSION\_HANDLE\_INVALID, CKR\_ARGUMENTS\_BAD.

3239 Example:

```
3240 CK_SESSION_HANDLE hSession;
3241 CK_MECHANISM digestMechanism;
3242 CK_BYTE_PTR pState;
3243 CK_ULONG ulStateLen;
3244 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};
3245 CK_BYTE data2[] = {0x02, 0x04, 0x08};
3246 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};
3247 CK_BYTE pDigest[20];
3248 CK_ULONG ulDigestLen;
3249 CK_RV rv;

3250 .
3251 .
3252 .
3253 /* Initialize hash operation */
3254 rv = C_DigestInit(hSession, &digestMechanism);
3255 assert(rv == CKR_OK);

3256 .
3257 /* Start hashing */
3258 rv = C_DigestUpdate(hSession, data1, sizeof(data1));
3259 assert(rv == CKR_OK);

3260 .
3261 /* Find out how big the state might be */
3262 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);
3263 assert(rv == CKR_OK);

3264 .
3265 /* Allocate some memory and then get the state */
3266 pState = (CK_BYTE_PTR) malloc(ulStateLen);
3267 rv = C_GetOperationState(hSession, pState, &ulStateLen);
```

```

3269 /* Continue hashing */
3270 rv = C_DigestUpdate(hSession, data2, sizeof(data2));
3271 assert(rv == CKR_OK);
3272
3273 /* Restore state. No key handles needed */
3274 rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);
3275 assert(rv == CKR_OK);
3276
3277 /* Continue hashing from where we saved state */
3278 rv = C_DigestUpdate(hSession, data3, sizeof(data3));
3279 assert(rv == CKR_OK);
3280
3281 /* Conclude hashing operation */
3282 ulDigestLen = sizeof(pDigest);
3283 rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
3284 if (rv == CKR_OK) {
3285 /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
3286 .
3287 .
3288 }

```

## 3289 5.6.8 C\_Login

```

3290 CK_DECLARE_FUNCTION(CK_RV, C_Login) (
3291     CK_SESSION_HANDLE hSession,
3292     CK_USER_TYPE userType,
3293     CK_UTF8CHAR_PTR pPin,
3294     CK_ULONG ulPinLen
3295 );

```

3296 **C\_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to  
 3297 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid  
 3298 UTF8 character, but the token may impose subset restrictions.

3299 When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's  
 3300 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User  
 3301 Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of **C\_Login** depends on the  
 3302 context in which it is called. Improper use of this user type will result in a return value  
 3303 CKR\_OPERATION\_NOT\_INITIALIZED..

3304 If the token has a "protected authentication path", as indicated by the  
 3305 **CKF\_PROTECTED\_AUTHENTICATION\_PATH** flag in its **CK\_TOKEN\_INFO** being set, then that means  
 3306 that there is some way for a user to be authenticated to the token without having to send a PIN through  
 3307 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or  
 3308 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some  
 3309 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*  
 3310 parameter to **C\_Login** should be NULL\_PTR. When **C\_Login** returns, whatever authentication method  
 3311 supported by the token will have been performed; a return value of CKR\_OK means that the user was  
 3312 successfully authenticated, and a return value of CKR\_PIN\_INCORRECT means that the user was  
 3313 denied access.

3314 If there are any active cryptographic or object finding operations in an application's session, and then  
3315 **C\_Login** is successfully executed by that application, it may or may not be the case that those operations  
3316 are still active. Therefore, before logging in, any active operations should be finished.  
3317 If the application calling **C\_Login** has a R/O session open with the token, then it will be unable to log the  
3318 SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code  
3319 CKR\_SESSION\_READ\_ONLY\_EXISTS.  
3320 **C\_Login** may be called repeatedly, without intervening **C\_Logout** calls, if (and only if) a key with the  
3321 CKA\_ALWAYS\_AUTHENTICATE attribute set to CK\_TRUE exists, and the user needs to do  
3322 cryptographic operation on this key. See further Section 4.9.  
3323 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
3324 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
3325 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3326 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_PIN\_INCORRECT,  
3327 CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3328 CKR\_SESSION\_READ\_ONLY\_EXISTS, CKR\_USER\_ALREADY\_LOGGED\_IN,  
3329 CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN, CKR\_USER\_PIN\_NOT\_INITIALIZED,  
3330 CKR\_USER\_TOO\_MANY\_TYPES, CKR\_USER\_TYPE\_INVALID.

3331 Example: see **C\_Logout**.

### 3332 5.6.9 **C\_LoginUser**

```
3333 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (
3334     CK_SESSION_HANDLE hSession,
3335     CK_USER_TYPE userType,
3336     CK_UTF8CHAR_PTR pPin,
3337     CK ULONG ulPinLen,
3338     CK_UTF8CHAR_PTR pUsername,
3339     CK ULONG ulUsernameLen
3340 );
```

3341 **C\_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*  
3342 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,  
3343 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to  
3344 contain any valid UTF8 character, but the token may impose subset restrictions.

3345 When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's  
3346 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User  
3347 Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of **C\_LoginUser** depends  
3348 on the context in which it is called. Improper use of this user type will result in a return value  
3349 CKR\_OPERATION\_NOT\_INITIALIZED.

3350 If the token has a "protected authentication path", as indicated by the  
3351 CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its CK\_TOKEN\_INFO being set, then that means  
3352 that there is some way for a user to be authenticated to the token without having to send a PIN through  
3353 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or  
3354 on the slot device. The user might not even use a PIN—authentication could be achieved by some  
3355 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*  
3356 parameter to **C\_LoginUser** should be NULL\_PTR. When **C\_LoginUser** returns, whatever authentication  
3357 method supported by the token will have been performed; a return value of CKR\_OK means that the user  
3358 was successfully authenticated, and a return value of CKR\_PIN\_INCORRECT means that the user was  
3359 denied access.

3360 If there are any active cryptographic or object finding operations in an application's session, and then  
3361 **C\_LoginUser** is successfully executed by that application, it may or may not be the case that those  
3362 operations are still active. Therefore, before logging in, any active operations should be finished.

3363 If the application calling **C\_LoginUser** has a R/O session open with the token, then it will be unable to log  
3364 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error  
3365 code CKR\_SESSION\_READ\_ONLY\_EXISTS.

3366 **C\_LoginUser** may be called repeatedly, without intervening **C\_Logout** calls, if (and only if) a key with the  
3367 CKA\_ALWAYS\_AUTHENTICATE attribute set to CK\_TRUE exists, and the user needs to do  
3368 cryptographic operation on this key. See further Section 4.9.  
3369 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
3370 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
3371 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3372 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_PIN\_INCORRECT,  
3373 CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3374 CKR\_SESSION\_READ\_ONLY\_EXISTS, CKR\_USER\_ALREADY\_LOGGED\_IN,  
3375 CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN, CKR\_USER\_PIN\_NOT\_INITIALIZED,  
3376 CKR\_USER\_TOO\_MANY\_TYPES, CKR\_USER\_TYPE\_INVALID.

3377 Example:

```
3378 CK_SESSION_HANDLE hSession;
3379 CK_UTF8CHAR userPin[] = {"MyPIN"};
3380 CK_UTF8CHAR userName[] = {"MyUserName"};
3381 CK_RV rv;
3382
3383 rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName,
3384 sizeof(userName)-1);
3385 if (rv == CKR_OK) {
3386 .
3387 .
3388 rv = C_Logout(hSession);
3389 if (rv == CKR_OK) {
3390 .
3391 .
3392 }
3393 }
```

### 3394 5.6.10 C\_Logout

```
3395 CK_DECLARE_FUNCTION(CK_RV, C_Logout) (
3396     CK_SESSION_HANDLE hSession
3397 );
```

3398 **C\_Logout** logs a user out from a token. *hSession* is the session's handle.

3399 Depending on the current user type, if the call succeeds, each of the application's sessions will enter  
3400 either the "R/W Public Session" state or the "R/O Public Session" state.

3401 When **C\_Logout** successfully executes, any of the application's handles to private objects become invalid  
3402 (even if a user is later logged back into the token, those handles remain invalid). In addition, all private  
3403 session objects from sessions belonging to the application are destroyed.

3404 If there are any active cryptographic or object-finding operations in an application's session, and then  
3405 **C\_Logout** is successfully executed by that application, it may or may not be the case that those  
3406 operations are still active. Therefore, before logging out, any active operations should be finished.

3407 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3408 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3409 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3410 CKR\_USER\_NOT\_LOGGED\_IN.

3411 Example:

```

3412 CK_SESSION_HANDLE hSession;
3413 CK_UTF8CHAR userPin[] = {"MyPIN"};
3414 CK_RV rv;
3415
3416 rv = C_Login(hSession, CKU_USER, userPin, sizeof(userPin)-1);
3417 if (rv == CKR_OK) {
3418 .
3419 .
3420     rv = C_Logout(hSession);
3421     if (rv == CKR_OK) {
3422 .
3423 .
3424 }
3425 }
```

## 3426 5.7 Object management functions

3427 Cryptoki provides the following functions for managing objects. Additional functions provided specifically  
 3428 for managing key objects are described in Section 5.18.

### 3429 5.7.1 C\_CreateObject

```

3430 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject) (
3431     CK_SESSION_HANDLE hSession,
3432     CK_ATTRIBUTE_PTR pTemplate,
3433     CK ULONG ulCount,
3434     CK_OBJECT_HANDLE_PTR phObject
3435 );
```

3436 **C\_CreateObject** creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's  
 3437 template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives  
 3438 the new object's handle.

3439 If a call to **C\_CreateObject** cannot support the precise template supplied to it, it will fail and return without  
 3440 creating any object.

3441 If **C\_CreateObject** is used to create a key object, the key object will have its **CKA\_LOCAL** attribute set to  
 3442 **CK\_FALSE**. If that key object is a secret or private key then the new key will have the  
**CKA\_ALWAYS\_SENSITIVE** attribute set to **CK\_FALSE**, and the **CKA\_NEVER\_EXTRACTABLE**  
 3444 attribute set to **CK\_FALSE**.

3445 Only session objects can be created during a read-only session. Only public objects can be created  
 3446 unless the normal user is logged in.

3447 Whenever an object is created, a value for **CKA\_UNIQUE\_ID** is generated and assigned to the new  
 3448 object (See Section 4.4.1).

3449 Return values: **CKR\_ARGUMENTS\_BAD**, **CKR\_ATTRIBUTE\_READ\_ONLY**,  
 3450 **CKR\_ATTRIBUTE\_TYPE\_INVALID**, **CKR\_ATTRIBUTE\_VALUE\_INVALID**,  
**CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_CURVE\_NOT\_SUPPORTED**, **CKR\_DEVICE\_ERROR**,  
 3452 **CKR\_DEVICE\_MEMORY**, **CKR\_DEVICE\_REMOVED**, **CKR\_DOMAIN\_PARAMS\_INVALID**,  
**CKR\_FUNCTION\_FAILED**, **CKR\_GENERAL\_ERROR**, **CKR\_HOST\_MEMORY**, **CKR\_OK**,  
 3454 **CKR\_PIN\_EXPIRED**, **CKR\_SESSION\_CLOSED**, **CKR\_SESSION\_HANDLE\_INVALID**,  
**CKR\_SESSION\_READ\_ONLY**, **CKR\_TEMPLATE\_INCOMPLETE**, **CKR\_TEMPLATE\_INCONSISTENT**,  
 3456 **CKR\_TOKEN\_WRITE\_PROTECTED**, **CKR\_USER\_NOT\_LOGGED\_IN**.

3457 Example:

```

3458 CK_SESSION_HANDLE hSession;
3459 CK_OBJECT_HANDLE
3460     hData,
3461     hCertificate,
3462     hKey;
3463 CK_OBJECT_CLASS
3464     dataClass = CKO_DATA,
3465     certificateClass = CKO_CERTIFICATE,
3466     keyClass = CKO_PUBLIC_KEY;
3467 CK_KEY_TYPE keyType = CKK_RSA;
3468 CK_UTF8CHAR application[] = {"My Application"};
3469 CK_BYTE dataValue[] = {...};
3470 CK_BYTE subject[] = {...};
3471 CK_BYTE id[] = {...};
3472 CK_BYTE certificateValue[] = {...};
3473 CK_BYTE modulus[] = {...};
3474 CK_BYTE exponent[] = {...};
3475 CK_BBOOL true = CK_TRUE;
3476 CK_ATTRIBUTE dataTemplate[] = {
3477     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3478     {CKA_TOKEN, &true, sizeof(true)},
3479     {CKA_APPLICATION, application, sizeof(application)-1},
3480     {CKA_VALUE, dataValue, sizeof(dataValue)}
3481 };
3482 CK_ATTRIBUTE certificateTemplate[] = {
3483     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
3484     {CKA_TOKEN, &true, sizeof(true)},
3485     {CKA_SUBJECT, subject, sizeof(subject)},
3486     {CKA_ID, id, sizeof(id)},
3487     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
3488 };
3489 CK_ATTRIBUTE keyTemplate[] = {
3490     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3491     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3492     {CKA_WRAP, &true, sizeof(true)},
3493     {CKA_MODULUS, modulus, sizeof(modulus)},
3494     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
3495 };
3496 CK_RV rv;
3497 .
3498 .
3499 .
3500 /* Create a data object */

```

```

3501 rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
3502 if (rv == CKR_OK) {
3503 .
3504 .
3505 }
3506
3507 /* Create a certificate object */
3508 rv = C_CreateObject(
3509     hSession, certificateTemplate, 5, &hCertificate);
3510 if (rv == CKR_OK) {
3511 .
3512 .
3513 }
3514
3515 /* Create an RSA public key object */
3516 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3517 if (rv == CKR_OK) {
3518 .
3519 .
3520 }

```

## 3521 5.7.2 C\_CopyObject

```

3522 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject) (
3523     CK_SESSION_HANDLE hSession,
3524     CK_OBJECT_HANDLE hObject,
3525     CK_ATTRIBUTE_PTR pTemplate,
3526     CK ULONG ulCount,
3527     CK_OBJECT_HANDLE_PTR phNewObject
3528 );

```

3529 **C\_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;  
 3530 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number  
 3531 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of  
 3532 the object.

3533 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,  
 3534 in the course of copying a secret key, a key's **CKA\_EXTRACTABLE** attribute may be changed from  
 3535 CK\_TRUE to CK\_FALSE, but not the other way around. If this change is made, the new key's  
 3536 **CKA\_NEVER\_EXTRACTABLE** attribute will have the value CK\_FALSE. Similarly, the template may  
 3537 specify that the new key's **CKA\_SENSITIVE** attribute be CK\_TRUE; the new key will have the same  
 3538 value for its **CKA\_ALWAYS\_SENSITIVE** attribute as the original key). It may also specify new values of  
 3539 the **CKA\_TOKEN** and **CKA\_PRIVATE** attributes (e.g., to copy a session object to a token object). If the  
 3540 template specifies a value of an attribute which is incompatible with other existing attributes of the object,  
 3541 the call fails with the return code CKR\_TEMPLATE\_INCONSISTENT.

3542 If a call to **C\_CopyObject** cannot support the precise template supplied to it, it will fail and return without  
 3543 creating any object. If the object indicated by *hObject* has its **CKA\_COPYABLE** attribute set to  
 3544 CK\_FALSE, **C\_CopyObject** will return CKR\_ACTION\_PROHIBITED.

3545 Whenever an object is copied, a new value for **CKA\_UNIQUE\_ID** is generated and assigned to the new  
 3546 object (See Section 4.4.1).

3547 Only session objects can be created during a read-only session. Only public objects can be created  
3548 unless the normal user is logged in.  
3549 Return values: , CKR\_ACTION\_PROHIBITED, CKR\_ARGUMENTS\_BAD,  
3550 CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID,  
3551 CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR,  
3552 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED,  
3553 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK,  
3554 CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
3555 CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCONSISTENT,  
3556 CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

3557 Example:

```
3558 CK_SESSION_HANDLE hSession;
3559 CK_OBJECT_HANDLE hKey, hNewKey;
3560 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
3561 CK_KEY_TYPE keyType = CKK_DES;
3562 CK_BYTE id[] = {...};
3563 CK_BYTE keyValue[] = {...};
3564 CK_BBOOL false = CK_FALSE;
3565 CK_BBOOL true = CK_TRUE;
3566 CK_ATTRIBUTE keyTemplate[] = {
3567     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3568     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3569     {CKA_TOKEN, &false, sizeof(false)},
3570     {CKA_ID, id, sizeof(id)},
3571     {CKA_VALUE, keyValue, sizeof(keyValue)}}
3572 };
3573 CK_ATTRIBUTE copyTemplate[] = {
3574     {CKA_TOKEN, &true, sizeof(true)}}
3575 };
3576 CK_RV rv;
3577 .
3578 .
3579 .
3580 /* Create a DES secret key session object */
3581 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3582 if (rv == CKR_OK) {
3583     /* Create a copy which is a token object */
3584     rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);
3585     .
3586     .
3587 }
```

### 3588 5.7.3 C\_DestroyObject

```
3589 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject) (
3590     CK_SESSION_HANDLE hSession,
```

```

3591     CK_OBJECT_HANDLE hObject
3592 ) ;
3593
C_DestroyObject destroys an object. hSession is the session's handle; and hObject is the object's
3594 handle.
3595 Only session objects can be destroyed during a read-only session. Only public objects can be destroyed
3596 unless the normal user is logged in.
3597 Certain objects may not be destroyed. Calling C_DestroyObject on such objects will result in the
3598 CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_DESTROYABLE
3599 attribute to determine if an object may be destroyed or not.
3600 Return values: CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED,
3601 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3602 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
3603 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3604 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
3605 CKR_TOKEN_WRITE_PROTECTED.
3606 Example: see C_GetObjectSize.

```

#### 3607 5.7.4 C\_GetObjectSize

```

3608 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize) (
3609     CK_SESSION_HANDLE hSession,
3610     CK_OBJECT_HANDLE hObject,
3611     CK_ULONG_PTR pulSize
3612 );
3613
C_GetObjectSize gets the size of an object in bytes. hSession is the session's handle; hObject is the
3614 object's handle; pulSize points to the location that receives the size in bytes of the object.
3615 Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure
3616 of how much token memory the object takes up. If an application deletes (say) a private object of size S,
3617 it might be reasonable to assume that the ulFreePrivateMemory field of the token's CK_TOKEN_INFO
3618 structure increases by approximately S.
3619 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3620 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3621 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
3622 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3623 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
3624 Example:

```

```

3625 CK_SESSION_HANDLE hSession;
3626 CK_OBJECT_HANDLE hObject;
3627 CK_OBJECT_CLASS dataClass = CKO_DATA;
3628 CK_UTF8CHAR application[] = {"My Application"};
3629 CK_BYTE value[] = {...};
3630 CK_BBOOL true = CK_TRUE;
3631 CK_ATTRIBUTE template[] = {
3632     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3633     {CKA_TOKEN, &true, sizeof(true)},
3634     {CKA_APPLICATION, application, sizeof(application)-1},
3635     {CKA_VALUE, value, sizeof(value)}
3636 };
3637 CK_ULONG ulSize;

```

```

3638 CK_RV rv;
3639 .
3640 .
3641 .
3642 rv = C_CreateObject(hSession, template, 4, &hObject);
3643 if (rv == CKR_OK) {
3644     rv = C_GetObjectSize(hSession, hObject, &ulSize);
3645     if (rv != CKR_INFORMATION_SENSITIVE) {
3646         .
3647         .
3648     }
3649     .
3650     rv = C_DestroyObject(hSession, hObject);
3651     .
3652     .
3653 }

```

## 3654 5.7.5 C\_GetAttributeValue

```

3655 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (
3656     CK_SESSION_HANDLE hSession,
3657     CK_OBJECT_HANDLE hObject,
3658     CK_ATTRIBUTE_PTR pTemplate,
3659     CK ULONG ulCount
3660 );

```

3661 **C\_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's  
 3662 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute  
 3663 values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the  
 3664 template.

3665 For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C\_GetAttributeValue** performs the following  
 3666 algorithm:

- 3667 1. If the specified attribute (i.e., the attribute specified by the type field) for the object cannot be revealed  
 3668 because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to  
 3669 hold the value CK\_UNAVAILABLE\_INFORMATION.
- 3670 2. Otherwise, if the specified value for the object is invalid (the object does not possess such an  
 3671 attribute), then the *ulValueLen* field in that triple is modified to hold the value  
 3672 CK\_UNAVAILABLE\_INFORMATION.
- 3673 3. Otherwise, if the *pValue* field has the value NULL\_PTR, then the *ulValueLen* field is modified to hold  
 3674 the exact length of the specified attribute for the object.
- 3675 4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified  
 3676 attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the  
 3677 *ulValueLen* field is modified to hold the exact length of the attribute.
- 3678 5. Otherwise, the *ulValueLen* field is modified to hold the value CK\_UNAVAILABLE\_INFORMATION.

3679 If case 1 applies to any of the requested attributes, then the call should return the value  
 3680 CKR\_ATTRIBUTE\_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should  
 3681 return the value CKR\_ATTRIBUTE\_TYPE\_INVALID. If case 5 applies to any of the requested attributes,  
 3682 then the call should return the value CKR\_BUFFER\_TOO\_SMALL. As usual, if more than one of these  
 3683 error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the  
 3684 requested attributes will CKR\_OK be returned.

3685 In the special case of an attribute whose value is an array of attributes, for example  
 3686 CKA\_WRAP\_TEMPLATE, where it is passed in with pValue not NULL, the length specified in ulValueLen  
 3687 MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is  
 3688 NULL\_PTR then the ulValueLen of elements within the array will be set to the required length. If the  
 3689 pValue of elements within the array is not NULL\_PTR, then the ulValueLen element of attributes within  
 3690 the array MUST reflect the space that the corresponding pValue points to, and pValue is filled in if there is  
 3691 sufficient room. Therefore it is important to initialize the contents of a buffer before calling  
 3692 C\_GetAttributeValue to get such an array value. Note that the type element of attributes within the array  
 3693 MUST be ignored on input and MUST be set on output. If any ulValueLen within the array isn't large  
 3694 enough, it will be set to CK\_UNAVAILABLE\_INFORMATION and the function will return  
 3695 CKR\_BUFFER\_TOO\_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too  
 3696 small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute  
 3697 type having the CKF\_ARRAY\_ATTRIBUTE bit set.  
 3698 Note that the error codes CKR\_ATTRIBUTE\_SENSITIVE, CKR\_ATTRIBUTE\_TYPE\_INVALID, and  
 3699 CKR\_BUFFER\_TOO\_SMALL do not denote true errors for **C\_GetAttributeValue**. If a call to  
 3700 **C\_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed  
 3701 every attribute in the template supplied to **C\_GetAttributeValue**. Each attribute in the template whose  
 3702 value *can be* returned by the call to **C\_GetAttributeValue** *will be* returned by the call to  
 3703 **C\_GetAttributeValue**.  
 3704 Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_SENSITIVE,  
 3705 CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_BUFFER\_TOO\_SMALL,  
 3706 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
 3707 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 3708 CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_SESSION\_CLOSED,  
 3709 CKR\_SESSION\_HANDLE\_INVALID.

3710 Example:

```

3711 CK_SESSION_HANDLE hSession;
3712 CK_OBJECT_HANDLE hObject;
3713 CK_BYTE_PTR pModulus, pExponent;
3714 CK_ATTRIBUTE template[] = {
3715   {CKA_MODULUS, NULL_PTR, 0},
3716   {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
3717 };
3718 CK_RV rv;
3719 .
3720 .
3721 .
3722 rv = C_GetAttributeValue(hSession, hObject, template, 2);
3723 if (rv == CKR_OK) {
3724   pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
3725   template[0].pValue = pModulus;
3726   /* template[0].ulValueLen was set by C_GetAttributeValue */
3727
3728   pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
3729   template[1].pValue = pExponent;
3730   /* template[1].ulValueLen was set by C_GetAttributeValue */
3731
3732   rv = C_GetAttributeValue(hSession, hObject, template, 2);

```

```

3733     if (rv == CKR_OK) {
3734         .
3735         .
3736     }
3737     free(pModulus);
3738     free(pExponent);
3739 }

```

## 3740 5.7.6 C\_SetAttributeValue

```

3741 CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue) (
3742     CK_SESSION_HANDLE hSession,
3743     CK_OBJECT_HANDLE hObject,
3744     CK_ATTRIBUTE_PTR pTemplate,
3745     CK_ULONG ulCount
3746 );

```

3747 **C\_SetAttributeValue** modifies the value of one or more attributes of an object. *hSession* is the session's  
 3748 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute  
 3749 values are to be modified and their new values; *ulCount* is the number of attributes in the template.

3750 Certain objects may not be modified. Calling **C\_SetAttributeValue** on such objects will result in the  
 3751 **CKR\_ACTION\_PROHIBITED** error code. An application can consult the object's **CKA\_MODIFIABLE**  
 3752 attribute to determine if an object may be modified or not.

3753 Only session objects can be modified during a read-only session.

3754 The template may specify new values for any attributes of the object that can be modified. If the template  
 3755 specifies a value of an attribute which is incompatible with other existing attributes of the object, the call  
 3756 fails with the return code **CKR\_TEMPLATE\_INCONSISTENT**.

3757 Not all attributes can be modified; see Section 4.1.2 for more details.

3758 Return values: **CKR\_ACTION\_PROHIBITED**, **CKR\_ARGUMENTS\_BAD**,  
 3759 **CKR\_ATTRIBUTE\_READ\_ONLY**, **CKR\_ATTRIBUTE\_TYPE\_INVALID**,  
 3760 **CKR\_ATTRIBUTE\_VALUE\_INVALID**, **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_DEVICE\_ERROR**,  
 3761 **CKR\_DEVICE\_MEMORY**, **CKR\_DEVICE\_REMOVED**, **CKR\_FUNCTION\_FAILED**,  
 3762 **CKR\_GENERAL\_ERROR**, **CKR\_HOST\_MEMORY**, **CKR\_OBJECT\_HANDLE\_INVALID**, **CKR\_OK**,  
 3763 **CKR\_SESSION\_CLOSED**, **CKR\_SESSION\_HANDLE\_INVALID**, **CKR\_SESSION\_READ\_ONLY**,  
 3764 **CKR\_TEMPLATE\_INCONSISTENT**, **CKR\_TOKEN\_WRITE\_PROTECTED**,  
 3765 **CKR\_USER\_NOT\_LOGGED\_IN**.

3766 Example:

```

3767 CK_SESSION_HANDLE hSession;
3768 CK_OBJECT_HANDLE hObject;
3769 CK_UTF8CHAR label[] = {"New label"};
3770 CK_ATTRIBUTE template[] = {
3771     {CKA_LABEL, label, sizeof(label)-1}
3772 };
3773 CK_RV rv;
3774
3775 .
3776 .
3777 rv = C_SetAttributeValue(hSession, hObject, template, 1);
3778 if (rv == CKR_OK) {

```

```
3779 .  
3780 .  
3781 }
```

## 3782 5.7.7 C\_FindObjectsInit

```
3783 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit) (  
3784     CK_SESSION_HANDLE hSession,  
3785     CK_ATTRIBUTE_PTR pTemplate,  
3786     CK ULONG ulCount  
3787 );
```

3788 **C\_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is  
3789 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;  
3790 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-  
3791 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

3792 After calling **C\_FindObjectsInit**, the application may call **C\_FindObjects** one or more times to obtain  
3793 handles for objects matching the template, and then eventually call **C\_FindObjectsFinal** to finish the  
3794 active search operation. At most one search operation may be active at a given time in a given session.

3795 The object search operation will only find objects that the session can view. For example, an object  
3796 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the  
3797 search template specifies that the search is for private objects).

3798 If a search operation is active, and objects are created or destroyed which fit the search template for the  
3799 active search operation, then those objects may or may not be found by the search operation. Note that  
3800 this means that, under these circumstances, the search operation may return invalid object handles.

3801 Even though **C\_FindObjectsInit** can return the values CKR\_ATTRIBUTE\_TYPE\_INVALID and  
3802 CKR\_ATTRIBUTE\_VALUE\_INVALID, it is not required to. For example, if it is given a search template  
3803 with nonexistent attributes in it, it can return CKR\_ATTRIBUTE\_TYPE\_INVALID, or it can initialize a  
3804 search operation which will match no objects and return CKR\_OK.

3805 If the CKA\_UNIQUE\_ID attribute is present in the search template, either zero or one objects will be  
3806 found, since at most one object can have any particular CKA\_UNIQUE\_ID value.

3807 Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_TYPE\_INVALID,  
3808 CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR,  
3809 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED,  
3810 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE,  
3811 CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

3812 Example: see **C\_FindObjectsFinal**.

## 3813 5.7.8 C\_FindObjects

```
3814 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects) (  
3815     CK_SESSION_HANDLE hSession,  
3816     CK_OBJECT_HANDLE_PTR phObject,  
3817     CK ULONG ulMaxObjectCount,  
3818     CK ULONG_PTR pulObjectCount  
3819 );
```

3820 **C\_FindObjects** continues a search for token and session objects that match a template, obtaining  
3821 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives  
3822 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles  
3823 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles  
3824 returned.

3825 If there are no more objects matching the template, then the location that *pulObjectCount* points to  
3826 receives the value 0.

3827 The search MUST have been initialized with **C\_FindObjectsInit**.  
3828 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
3829 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
3830 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
3831 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.  
3832 Example: see **C\_FindObjectsFinal**.

### 3833 **5.7.9 C\_FindObjectsFinal**

```
3834 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal) (
3835     CK_SESSION_HANDLE hSession
3836 );
```

3837 **C\_FindObjectsFinal** terminates a search for token and session objects. *hSession* is the session's  
3838 handle.

3839 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
3840 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3841 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
3842 CKR\_SESSION\_HANDLE\_INVALID.

3843 Example:

```
3844 CK_SESSION_HANDLE hSession;
3845 CK_OBJECT_HANDLE hObject;
3846 CK_ULONG ulObjectCount;
3847 CK_RV rv;
3848 .
3849 .
3850 .
3851 rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
3852 assert(rv == CKR_OK);
3853 while (1) {
3854     rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
3855     if (rv != CKR_OK || ulObjectCount == 0)
3856         break;
3857     .
3858     .
3859 }
3860 .
3861 rv = C_FindObjectsFinal(hSession);
3862 assert(rv == CKR_OK);
```

## 3863 **5.8 Encryption functions**

3864 Cryptoki provides the following functions for encrypting data:

### 3865 **5.8.1 C\_EncryptInit**

```
3866 CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit) (
3867     CK_SESSION_HANDLE hSession,
3868     CK_MECHANISM_PTR pMechanism,
```

```

3869     CK_OBJECT_HANDLE hKey
390 ) ;
391
392 C_EncryptInit initializes an encryption operation. hSession is the session's handle; pMechanism points
393 to the encryption mechanism; hKey is the handle of the encryption key.
394
395 The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports
396 encryption, MUST be CK_TRUE.
397
398 After calling C_EncryptInit, the application can either call C_Encrypt to encrypt data in a single part; or
399 call C_EncryptUpdate zero or more times, followed by C_EncryptFinal, to encrypt data in multiple parts.
400 The encryption operation is active until the application uses a call to C_Encrypt or C_EncryptFinal to
401 actually obtain the final piece of ciphertext. To process additional data (in single or multiple parts), the
402 application MUST call C_EncryptInit again.
403
404 C_EncryptInit can be called with pMechanism set to NULL_PTR to terminate an active encryption
405 operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
406 must be returned.
407
408 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
409 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
410 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
411 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
412 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
413 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
414 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
415 CKR_OPERATION_CANCEL_FAILED.
416
417 Example: see C_EncryptFinal.

```

## 3892 5.8.2 C\_Encrypt

```

3903 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt) (
3904     CK_SESSION_HANDLE hSession,
3905     CK_BYTE_PTR pData,
3906     CK ULONG ulDataLen,
3907     CK_BYTE_PTR pEncryptedData,
3908     CK ULONG_PTR pulEncryptedDataLen
3909 ) ;
3910
3911 C_Encrypt encrypts single-part data. hSession is the session's handle; pData points to the data;
3912 ulDataLen is the length in bytes of the data; pEncryptedData points to the location that receives the
3913 encrypted data; pulEncryptedDataLen points to the location that holds the length in bytes of the encrypted
3914 data.
3915
3916 C_Encrypt uses the convention described in Section 5.2 on producing output.
3917 The encryption operation MUST have been initialized with C_EncryptInit. A call to C_Encrypt always
3918 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3919 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
3920 ciphertext.
3921
3922 C_Encrypt cannot be used to terminate a multi-part operation, and MUST be called after C_EncryptInit
3923 without intervening C_EncryptUpdate calls.
3924
3925 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
3926 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
3927 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
3928 C_Encrypt will fail with return code CKR_DATA_LEN_RANGE.
3929
3930 The plaintext and ciphertext can be in the same place, i.e., it is OK if pData and pEncryptedData point to
3931 the same location.
3932
3933 For most mechanisms, C_Encrypt is equivalent to a sequence of C_EncryptUpdate operations followed
3934 by C_EncryptFinal.

```

3919 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
3920 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
3921 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
3922 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
3923 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
3924 CKR\_SESSION\_HANDLE\_INVALID.

3925 Example: see **C\_EncryptFinal** for an example of similar functions.

### 3926 5.8.3 C\_EncryptUpdate

```
3927 CK_DECLARE_FUNCTION(CK_RV, C_EncryptUpdate) (
3928     CK_SESSION_HANDLE hSession,
3929     CK_BYTE_PTR pPart,
3930     CK_ULONG ulPartLen,
3931     CK_BYTE_PTR pEncryptedPart,
3932     CK_ULONG_PTR pulEncryptedPartLen
3933 );
```

3934 **C\_EncryptUpdate** continues a multiple-part encryption operation, processing another data part.  
3935 *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part;  
3936 *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points  
3937 to the location that holds the length in bytes of the encrypted data part.

3938 **C\_EncryptUpdate** uses the convention described in Section 5.2 on producing output.

3939 The encryption operation MUST have been initialized with **C\_EncryptInit**. This function may be called  
3940 any number of times in succession. A call to **C\_EncryptUpdate** which results in an error other than  
3941 CKR\_BUFFER\_TOO\_SMALL terminates the current encryption operation.

3942 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to  
3943 the same location.

3944 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
3945 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
3946 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
3947 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
3948 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

3949 Example: see **C\_EncryptFinal**.

### 3950 5.8.4 C\_EncryptFinal

```
3951 CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal) (
3952     CK_SESSION_HANDLE hSession,
3953     CK_BYTE_PTR pLastEncryptedPart,
3954     CK_ULONG_PTR pulLastEncryptedPartLen
3955 );
```

3956 **C\_EncryptFinal** finishes a multiple-part encryption operation. *hSession* is the session's handle;  
3957 *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any;  
3958 *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

3959 **C\_EncryptFinal** uses the convention described in Section 5.2 on producing output.

3960 The encryption operation MUST have been initialized with **C\_EncryptInit**. A call to **C\_EncryptFinal**  
3961 always terminates the active encryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
3962 successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the  
3963 ciphertext.

3964 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,  
3965 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints  
3966 are not satisfied, then **C\_EncryptFinal** will fail with return code CKR\_DATA\_LEN\_RANGE.

3967 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
3968 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
3969 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
3970 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
3971 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.  
3972 Example:

```
3973 #define PLAINTEXT_BUF_SZ 200
3974 #define CIPHERTEXT_BUF_SZ 256
3975
3976 CK ULONG firstPieceLen, secondPieceLen;
3977 CK SESSION_HANDLE hSession;
3978 CK OBJECT_HANDLE hKey;
3979 CK BYTE iv[8];
3980 CK_MECHANISM mechanism = {
3981     CKM DES_CBC_PAD, iv, sizeof(iv)
3982 };
3983 CK BYTE data[PLAINTEXT_BUF_SZ];
3984 CK BYTE encryptedData[CIPHERTEXT_BUF_SZ];
3985 CK ULONG ulEncryptedData1Len;
3986 CK ULONG ulEncryptedData2Len;
3987 CK ULONG ulEncryptedData3Len;
3988 CK RV rv;
3989
3990 .
3991 .
3992 firstPieceLen = 90;
3993 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
3994 rv = C_EncryptInit(hSession, &mechanism, hKey);
3995 if (rv == CKR_OK) {
3996     /* Encrypt first piece */
3997     ulEncryptedData1Len = sizeof(encryptedData);
3998     rv = C_EncryptUpdate(
3999         hSession,
4000         &data[0], firstPieceLen,
4001         &encryptedData[0], &ulEncryptedData1Len);
4002     if (rv != CKR_OK) {
4003         .
4004         .
4005     }
4006
4007     /* Encrypt second piece */
4008     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
4009     rv = C_EncryptUpdate(
4010         hSession,
```

```

4011     &data[firstPieceLen], secondPieceLen,
4012     &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
4013     if (rv != CKR_OK) {
4014         .
4015         .
4016     }
4017
4018     /* Get last little encrypted bit */
4019     ulEncryptedData3Len =
4020         sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
4021     rv = C_EncryptFinal(
4022         hSession,
4023         &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
4024         &ulEncryptedData3Len);
4025     if (rv != CKR_OK) {
4026         .
4027         .
4028     }
4029 }
```

## 4030 5.9 Message-based encryption functions

4031 Message-based encryption refers to the process of encrypting multiple messages using the same  
 4032 encryption mechanism and encryption key. The encryption mechanism can be either an authenticated  
 4033 encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

4034 Cryptoki provides the following functions for message-based encryption:

### 4035 5.9.1 C\_MessageEncryptInit

```

4036 CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptInit) (
4037     CK_SESSION_HANDLE hSession,
4038     CK_MECHANISM_PTR pMechanism,
4039     CK_OBJECT_HANDLE hKey
4040 );
```

4041 **C\_MessageEncryptInit** prepares a session for one or more encryption operations that use the same  
 4042 encryption mechanism and encryption key. *hSession* is the session's handle; *pMechanism* points to the  
 4043 encryption mechanism; *hKey* is the handle of the encryption key.

4044 The **CKA\_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption,  
 4045 MUST be **CK\_TRUE**.

4046 After calling **C\_MessageEncryptInit**, the application can either call **C\_EncryptMessage** to encrypt a  
 4047 message in a single part, or call **C\_EncryptMessageBegin**, followed by **C\_EncryptMessageNext** one or  
 4048 more times, to encrypt a message in multiple parts. This may be repeated several times. The message-  
 4049 based encryption process is active until the application calls **C\_MessageEncryptFinal** to finish the  
 4050 message-based encryption process.

4051 **C\_MessageEncryptInit** can be called with *pMechanism* set to **NULL\_PTR** to terminate a message-based  
 4052 encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an  
 4053 active operation has been initialized and it cannot be cancelled, **CKR\_OPERATION\_CANCEL\_FAILED**  
 4054 must be returned.

4055 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4056 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4057 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED,  
4058 CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT,  
4059 CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK,  
4060 CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED,  
4061 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4062 CKR\_OPERATION\_CANCEL FAILED.

## 4063 5.9.2 C\_EncryptMessage

```
4064 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage) (
4065     CK_SESSION_HANDLE hSession,
4066     CK_VOID_PTR pParameter,
4067     CK_ULONG ulParameterLen,
4068     CK_BYTE_PTR pAssociatedData,
4069     CK_ULONG ulAssociatedDataLen,
4070     CK_BYTE_PTR pPlaintext,
4071     CK_ULONG ulPlaintextLen,
4072     CK_BYTE_PTR pCiphertext,
4073     CK_ULONG_PTR pulCiphertextLen
4074 );
```

4075 **C\_EncryptMessage** encrypts a message in a single part. *hSession* is the session's handle; *pParameter*  
4076 and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation;  
4077 *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism;  
4078 *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data;  
4079 *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location  
4080 that holds the length in bytes of the encrypted data.

4081 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter  
4082 passed to **C\_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For  
4083 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV  
4084 generator will be output to the *pParameter* buffer.

4085 If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and  
4086 should be set to (NULL, 0).

4087 **C\_EncryptMessage** uses the convention described in Section 5.2 on producing output.

4088 The message-based encryption process MUST have been initialized with **C\_MessageEncryptInit**. A call  
4089 to **C\_EncryptMessage** begins and terminates a message encryption operation.

4090 **C\_EncryptMessage** cannot be called in the middle of a multi-part message encryption operation.

4091 For some encryption mechanisms, the input plaintext data has certain length constraints (either because  
4092 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input  
4093 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then  
4094 **C\_EncryptMessage** will fail with return code CKR\_DATA\_LEN\_RANGE. The plaintext and ciphertext can  
4095 be in the same place, i.e., it is OK if *pPlaintext* and *pCiphertext* point to the same location.

4096 For most mechanisms, **C\_EncryptMessage** is equivalent to **C\_EncryptMessageBegin** followed by a  
4097 sequence of **C\_EncryptMessageNext** operations.

4098 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4099 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
4100 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4101 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4102 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

## 4103 5.9.3 C\_EncryptMessageBegin

```
4104 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin) (
```

```
4105     CK_SESSION_HANDLE hSession,  
4106     CK_VOID_PTR pParameter,  
4107     CK ULONG ulParameterLen,  
4108     CK_BYTE_PTR pAssociatedData,  
4109     CK ULONG ulAssociatedDataLen  
4110 );
```

4111 **C\_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the  
4112 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the  
4113 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data  
4114 for an AEAD mechanism.

4115 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter  
4116 passed to **C\_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For  
4117 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV  
4118 generator will be output to the *pParameter* buffer.

4119 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be  
4120 set to (NULL, 0).

4121 After calling **C\_EncryptMessageBegin**, the application should call **C\_EncryptMessageNext** one or  
4122 more times to encrypt the message in multiple parts. The message encryption operation is active until the  
4123 application uses a call to **C\_EncryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE to actually  
4124 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the  
4125 application MUST call **C\_EncryptMessage** or **C\_EncryptMessageBegin** again.

4126 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4127 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4128 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE,  
4129 CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
4130 CKR\_USER\_NOT\_LOGGED\_IN.

#### 4131 5.9.4 C\_EncryptMessageNext

```
4132 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (  
4133     CK_SESSION_HANDLE hSession,  
4134     CK_VOID_PTR pParameter,  
4135     CK ULONG ulParameterLen,  
4136     CK_BYTE_PTR pPlaintextPart,  
4137     CK ULONG ulPlaintextPartLen,  
4138     CK_BYTE_PTR pCiphertextPart,  
4139     CK ULONG_PTR pulCiphertextPartLen,  
4140     CK_FLAGS flags  
4141 );
```

4142 **C\_EncryptMessageNext** continues a multiple-part message encryption operation, processing another  
4143 message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any  
4144 mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the  
4145 plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart*  
4146 points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the  
4147 location that holds the length in bytes of the encrypted message part; *flags* is set to 0 if there is more  
4148 plaintext data to follow, or set to CKF\_END\_OF\_MESSAGE if this is the last plaintext part.

4149 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter  
4150 passed to **C\_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For  
4151 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV  
4152 generator will be output to the *pParameter* buffer.

4153 **C\_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

4154 The message encryption operation MUST have been started with **C\_EncryptMessageBegin**. This  
4155 function may be called any number of times in succession. A call to **C\_EncryptMessageNext** with flags=0  
4156 which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message

4157 encryption operation. A call to **C\_EncryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE always  
4158 terminates the active message encryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
4159 successful call (i.e., one which returns **CKR\_OK**) to determine the length of the buffer needed to hold the  
4160 ciphertext.

4161 Although the last **C\_EncryptMessageNext** call ends the encryption of a message, it does not finish the  
4162 message-based encryption process. Additional **C\_EncryptMessage** or **C\_EncryptMessageBegin** and  
4163 **C\_EncryptMessageNext** calls may be made on the session.

4164 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*  
4165 point to the same location.

4166 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,  
4167 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints  
4168 are not satisfied when the final message part is supplied (i.e., with flags=CKF\_END\_OF\_MESSAGE),  
4169 then **C\_EncryptMessageNext** will fail with return code CKR\_DATA\_LEN\_RANGE.

4170 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4171 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
4172 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
4173 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
4174 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

## 4175 5.9.5 C\_MessageEncryptFinal

```
4176 CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptFinal) (
4177     CK_SESSION_HANDLE hSession
4178 );
```

4179 **C\_MessageEncryptFinal** finishes a message-based encryption process. *hSession* is the session's  
4180 handle.

4181 The message-based encryption process MUST have been initialized with **C\_MessageEncryptInit**.

4182 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4183 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4184 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4185 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4186 CKR\_SESSION\_HANDLE\_INVALID.

4187 Example:

```
4188 #define PLAINTEXT_BUF_SZ 200
4189 #define AUTH_BUF_SZ 100
4190 #define CIPHERTEXT_BUF_SZ 256
4191
4192 CK_SESSION_HANDLE hSession;
4193 CK_OBJECT_HANDLE hKey;
4194 CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
4195 CK_BYTE tag[16];
4196 CK_GCM_MESSAGE_PARAMS gcmParams = {
4197     iv,
4198     sizeof(iv) * 8,
4199     0,
4200     CKG_NO_GENERATE,
4201     tag,
4202     sizeof(tag) * 8
```

```

4203    } ;
4204    CK_MECHANISM mechanism = {
4205        CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
4206    } ;
4207    CK_BYTE data[2][PLAINTEXT_BUF_SZ];
4208    CK_BYTE auth[2][AUTH_BUF_SZ];
4209    CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
4210    CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
4211    CK ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;
4212
4213 /* error handling is omitted for better readability */
4214 .
4215 .
4216 C_MessageEncryptInit(hSession, &mechanism, hKey);
4217 /* encrypt message en bloc with given IV */
4218 ulEncryptedDataLen = sizeof(encryptedData[0]);
4219 C_EncryptMessage(hSession,
4220     &gcmParams, sizeof(gcmParams),
4221     &auth[0][0], sizeof(auth[0]),
4222     &data[0][0], sizeof(data[0]),
4223     &encryptedData[0][0], &ulEncryptedDataLen);
4224 /* iv and tag are set now for message */
4225
4226 /* encrypt message in two steps with generated IV */
4227 gcmParams.ivGenerator = CKG_GENERATE;
4228 C_EncryptMessageBegin(hSession,
4229     &gcmParams, sizeof(gcmParams),
4230     &auth[1][0], sizeof(auth[1])
4231 );
4232 /* encrypt first piece */
4233 ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
4234 C_EncryptMessageNext(hSession,
4235     &gcmParams, sizeof(gcmParams),
4236     &data[1][0], firstPieceLen,
4237     &encryptedData[1][0], &ulFirstEncryptedDataLen,
4238     0
4239 );
4240 /* encrypt second piece */
4241 ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
4242 C_EncryptMessageNext(hSession,
4243     &gcmParams, sizeof(gcmParams),
4244     &data[1][firstPieceLen], sizeof(data[1])-firstPieceLen,
4245     &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,

```

```

4246     CKF_END_OF_MESSAGE
4247 );
4248 /* tag is set now for message */
4249
4250 /* finalize */
4251 C_MessageEncryptFinal(hSession);

```

## 4252 5.10 Decryption functions

4253 Cryptoki provides the following functions for decrypting data:

### 4254 5.10.1 C\_DecryptInit

```

4255 CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit) (
4256     CK_SESSION_HANDLE hSession,
4257     CK_MECHANISM_PTR pMechanism,
4258     CK_OBJECT_HANDLE hKey
4259 );

```

4260 **C\_DecryptInit** initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to  
4261 the decryption mechanism; *hKey* is the handle of the decryption key.

4262 The **CKA\_DECRYPT** attribute of the decryption key, which indicates whether the key supports  
4263 decryption, MUST be CK\_TRUE.

4264 After calling **C\_DecryptInit**, the application can either call **C\_Decrypt** to decrypt data in a single part; or  
4265 call **C\_DecryptUpdate** zero or more times, followed by **C\_DecryptFinal**, to decrypt data in multiple parts.  
4266 The decryption operation is active until the application uses a call to **C\_Decrypt** or **C\_DecryptFinal** to  
4267 actually obtain the final piece of plaintext. To process additional data (in single or multiple parts), the  
4268 application MUST call **C\_DecryptInit** again.

4269 **C\_DecryptInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active decryption  
4270 operation. If an active operation cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be  
4271 returned.

4272 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4273 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4274 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4275 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,  
4276 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
4277 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
4278 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4279 CKR\_OPERATION\_CANCEL\_FAILED.

4280 Example: see **C\_DecryptFinal**.

### 4281 5.10.2 C\_Decrypt

```

4282 CK_DECLARE_FUNCTION(CK_RV, C_Decrypt) (
4283     CK_SESSION_HANDLE hSession,
4284     CK_BYTE_PTR pEncryptedData,
4285     CK_ULONG ulEncryptedDataLen,
4286     CK_BYTE_PTR pData,
4287     CK_ULONG_PTR pulDataLen
4288 );

```

4289 **C\_Decrypt** decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData*  
4290 points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the

4291 location that receives the recovered data; *pulDataLen* points to the location that holds the length of the  
4292 recovered data.

4293 **C\_Decrypt** uses the convention described in Section 5.2 on producing output.

4294 The decryption operation MUST have been initialized with **C\_DecryptInit**. A call to **C\_Decrypt** always  
4295 terminates the active decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
4296 successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the  
4297 plaintext.

4298 **C\_Decrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C\_DecryptInit**  
4299 without intervening **C\_DecryptUpdate** calls.

4300 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to  
4301 the same location.

4302 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either  
4303 CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

4304 For most mechanisms, **C\_Decrypt** is equivalent to a sequence of **C\_DecryptUpdate** operations followed  
4305 by **C\_DecryptFinal**.

4306 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4307 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4308 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
4309 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4310 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4311 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

4312 Example: see **C\_DecryptFinal** for an example of similar functions.

### 4313 5.10.3 C\_DecryptUpdate

```
4314 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (
4315     CK_SESSION_HANDLE hSession,
4316     CK_BYTE_PTR pEncryptedPart,
4317     CK_ULONG ulEncryptedPartLen,
4318     CK_BYTE_PTR pPart,
4319     CK_ULONG_PTR pulPartLen
4320 );
```

4321 **C\_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data  
4322 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;  
4323 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the  
4324 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

4325 **C\_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

4326 The decryption operation MUST have been initialized with **C\_DecryptInit**. This function may be called  
4327 any number of times in succession. A call to **C\_DecryptUpdate** which results in an error other than  
4328 CKR\_BUFFER\_TOO\_SMALL terminates the current decryption operation.

4329 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to  
4330 the same location.

4331 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4332 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4333 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
4334 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4335 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4336 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

4337 Example: See **C\_DecryptFinal**.

4338 **5.10.4 C\_DecryptFinal**

```
4339 CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal) (
4340     CK_SESSION_HANDLE hSession,
4341     CK_BYTE_PTR pLastPart,
4342     CK ULONG_PTR pulLastPartLen
4343 );
```

4344 **C\_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session's handle;  
4345 *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to  
4346 the location that holds the length of the last recovered data part.

4347 **C\_DecryptFinal** uses the convention described in Section 5.2 on producing output.

4348 The decryption operation MUST have been initialized with **C\_DecryptInit**. A call to **C\_DecryptFinal**  
4349 always terminates the active decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
4350 successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the  
4351 plaintext.

4352 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either  
4353 CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

4354 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4355 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4356 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
4357 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4358 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4359 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

4360 Example:

```
4361 #define CIPHERTEXT_BUF_SZ 256
4362 #define PLAINTEXT_BUF_SZ 256
4363
4364 CK ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
4365 CK SESSION_HANDLE hSession;
4366 CK OBJECT_HANDLE hKey;
4367 CK BYTE iv[8];
4368 CK_MECHANISM mechanism = {
4369     CKM DES CBC PAD, iv, sizeof(iv)
4370 };
4371 CK BYTE data[PLAINTEXT_BUF_SZ];
4372 CK BYTE encryptedData[CIPHERTEXT_BUF_SZ];
4373 CK ULONG ulData1Len, ulData2Len, ulData3Len;
4374 CK RV rv;
4375 .
4376 .
4377 .
4378 firstEncryptedPieceLen = 90;
4379 secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ - firstEncryptedPieceLen;
4380 rv = C_DecryptInit(hSession, &mechanism, hKey);
4381 if (rv == CKR_OK) {
4382     /* Decrypt first piece */
4383     ulData1Len = sizeof(data);
```

```

4384     rv = C_DecryptUpdate(
4385         hSession,
4386         &encryptedData[0], firstEncryptedPieceLen,
4387         &data[0], &ulData1Len);
4388     if (rv != CKR_OK) {
4389         .
4390         .
4391     }
4392
4393     /* Decrypt second piece */
4394     ulData2Len = sizeof(data)-ulData1Len;
4395     rv = C_DecryptUpdate(
4396         hSession,
4397         &encryptedData[firstEncryptedPieceLen],
4398         secondEncryptedPieceLen,
4399         &data[ulData1Len], &ulData2Len);
4400     if (rv != CKR_OK) {
4401         .
4402         .
4403     }
4404
4405     /* Get last little decrypted bit */
4406     ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
4407     rv = C_DecryptFinal(
4408         hSession,
4409         &data[ulData1Len+ulData2Len], &ulData3Len);
4410     if (rv != CKR_OK) {
4411         .
4412         .
4413     }
4414 }
```

## 4415 5.11 Message-based decryption functions

4416 Message-based decryption refers to the process of decrypting multiple encrypted messages using the  
 4417 same decryption mechanism and decryption key. The decryption mechanism can be either an  
 4418 authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.  
 4419 Cryptoki provides the following functions for message-based decryption.

### 4420 5.11.1 C\_MessageDecryptInit

```

4421 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit) (
4422     CK_SESSION_HANDLE hSession,
4423     CK_MECHANISM_PTR pMechanism,
4424     CK_OBJECT_HANDLE hKey
4425 );
```

4426 **C\_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or  
 4427 more decryption operations that use the same decryption mechanism and decryption key. *hSession* is  
 4428 the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the  
 4429 decryption key.  
 4430 The CKA\_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption,  
 4431 MUST be CK\_TRUE.  
 4432 After calling **C\_MessageDecryptInit**, the application can either call **C\_DecryptMessage** to decrypt an  
 4433 encrypted message in a single part; or call **C\_DecryptMessageBegin**, followed by  
 4434 **C\_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This  
 4435 may be repeated several times. The message-based decryption process is active until the application  
 4436 uses a call to **C\_MessageDecryptFinal** to finish the message-based decryption process.  
 4437 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
 4438 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
 4439 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 4440 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,  
 4441 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
 4442 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
 4443 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
 4444 CKR\_OPERATION\_CANCEL\_FAILED.

## 4445 5.11.2 C\_DecryptMessage

```

4446 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (
4447   CK_SESSION_HANDLE hSession,
4448   CK_VOID_PTR pParameter,
4449   CK ULONG ulParameterLen,
4450   CK_BYTE_PTR pAssociatedData,
4451   CK ULONG ulAssociatedDataLen,
4452   CK_BYTE_PTR pCiphertext,
4453   CK ULONG ulCiphertextLen,
4454   CK_BYTE_PTR pPlaintext,
4455   CK ULONG_PTR pulPlaintextLen
4456 );

```

4457 **C\_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session's handle;  
 4458 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption  
 4459 operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD  
 4460 mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted  
 4461 message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to  
 4462 the location that holds the length of the recovered message.

4463 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of  
 4464 **C\_EncryptMessage**, *pParameter* is always an input parameter.

4465 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and  
 4466 should be set to (NULL, 0).

4467 **C\_DecryptMessage** uses the convention described in Section 5.2 on producing output.

4468 The message-based decryption process MUST have been initialized with **C\_MessageDecryptInit**. A call  
 4469 to **C\_DecryptMessage** begins and terminates a message decryption operation.

4470 **C\_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

4471 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to  
 4472 the same location.

4473 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either  
 4474 CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

4475 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or  
 4476 ciphertext cannot be verified, then CKR\_AEAD\_DECRYPT\_FAILED is returned.

4477 For most mechanisms, **C\_DecryptMessage** is equivalent to **C\_DecryptMessageBegin** followed by a  
4478 sequence of **C\_DecryptMessageNext** operations.  
4479 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4480 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4481 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
4482 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_AEAD\_DECRYPT\_FAILED,  
4483 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4484 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4485 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4486 CKR\_OPERATION\_CANCEL\_FAILED.

### 4487 5.11.3 C\_DecryptMessageBegin

```
4488 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (
4489     CK_SESSION_HANDLE hSession,
4490     CK_VOID_PTR pParameter,
4491     CK ULONG ulParameterLen,
4492     CK_BYTE_PTR pAssociatedData,
4493     CK ULONG ulAssociatedDataLen
4494 );
```

4495 **C\_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the  
4496 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the  
4497 message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data  
4498 for an AEAD mechanism.

4499 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of  
4500 **C\_EncryptMessageBegin**, *pParameter* is always an input parameter.

4501 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and  
4502 should be set to (NULL, 0).

4503 After calling **C\_DecryptMessageBegin**, the application should call **C\_DecryptMessageNext** one or  
4504 more times to decrypt the encrypted message in multiple parts. The message decryption operation is  
4505 active until the application uses a call to **C\_DecryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE  
4506 to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or  
4507 multiple parts), the application MUST call **C\_DecryptMessage** or **C\_DecryptMessageBegin** again.

4508 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4509 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4510 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4511 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
4512 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### 4513 5.11.4 C\_DecryptMessageNext

```
4514 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (
4515     CK_SESSION_HANDLE hSession,
4516     CK_VOID_PTR pParameter,
4517     CK ULONG ulParameterLen,
4518     CK_BYTE_PTR pCiphertextPart,
4519     CK ULONG ulCiphertextPartLen,
4520     CK_BYTE_PTR pPlaintextPart,
4521     CK ULONG_PTR pulPlaintextPartLen,
4522     CK_FLAGS flags
4523 );
```

4524 **C\_DecryptMessageNext** continues a multiple-part message decryption operation, processing another  
4525 encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any  
4526 mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the

4527 encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart*  
 4528 points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location  
 4529 that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to  
 4530 follow, or set to CKF\_END\_OF\_MESSAGE if this is the last ciphertext part.  
 4531 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of  
 4532 **C\_EncryptMessageNext**, *pParameter* is always an input parameter.  
 4533 **C\_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.  
 4534 The message decryption operation MUST have been started with **C\_DecryptMessageBegin**. This  
 4535 function may be called any number of times in succession. A call to **C\_DecryptMessageNext** with  
 4536 flags=0 which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message  
 4537 decryption operation. A call to **C\_DecryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE always  
 4538 terminates the active message decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
 4539 successful call (i.e., one which returns CKR\_OK) to determine the length of the buffer needed to hold the  
 4540 plaintext.  
 4541 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart*  
 4542 point to the same location.  
 4543 Although the last **C\_DecryptMessageNext** call ends the decryption of a message, it does not finish the  
 4544 message-based decryption process. Additional **C\_DecryptMessage** or **C\_DecryptMessageBegin** and  
 4545 **C\_DecryptMessageNext** calls may be made on the session.  
 4546 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either  
 4547 CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned by  
 4548 the last **C\_DecryptMessageNext** call.  
 4549 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or  
 4550 ciphertext cannot be verified, then CKR\_AEAD\_DECRYPT\_FAILED is returned by the last  
 4551 **C\_DecryptMessageNext** call.  
 4552 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
 4553 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
 4554 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
 4555 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_AEAD\_DECRYPT\_FAILED,  
 4556 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 4557 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
 4558 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

## 4559 **5.11.5 C\_MessageDecryptFinal**

```

4560 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal) (
4561   CK_SESSION_HANDLE hSession
4562 );

```

4563 **C\_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's  
 4564 handle.

4565 The message-based decryption process MUST have been initialized with **C\_MessageDecryptInit**.  
 4566 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
 4567 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
 4568 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 4569 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
 4570 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

## 4571 **5.12 Message digesting functions**

4572 Cryptoki provides the following functions for digesting data:

4573 **5.12.1 C\_DigestInit**

```
4574 CK_DECLARE_FUNCTION(CK_RV, C_DigestInit) (
4575     CK_SESSION_HANDLE hSession,
4576     CK_MECHANISM_PTR pMechanism
4577 );
```

4578 **C\_DigestInit** initializes a message-digesting operation. *hSession* is the session's handle; *pMechanism* points to the digesting mechanism.

4580 After calling **C\_DigestInit**, the application can either call **C\_Digest** to digest data in a single part; or call  
4581 **C\_DigestUpdate** zero or more times, followed by **C\_DigestFinal**, to digest data in multiple parts. The  
4582 message-digesting operation is active until the application uses a call to **C\_Digest** or **C\_DigestFinal** to  
4583 actually obtain the message digest. To process additional data (in single or multiple parts), the  
4584 application MUST call **C\_DigestInit** again.

4585 **C\_DigestInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active message-digesting  
4586 operation. If an operation has been initialized and it cannot be cancelled,  
4587 CKR\_OPERATION\_CANCEL\_FAILED must be returned.

4588 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4589 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4590 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4591 CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID,  
4592 CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED,  
4593 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4594 CKR\_OPERATION\_CANCEL\_FAILED.

4595 Example: see **C\_DigestFinal**.

4596 **5.12.2 C\_Digest**

```
4597 CK_DECLARE_FUNCTION(CK_RV, C_Digest) (
4598     CK_SESSION_HANDLE hSession,
4599     CK_BYTE_PTR pData,
4600     CK_ULONG ulDataLen,
4601     CK_BYTE_PTR pDigest,
4602     CK_ULONG_PTR pulDigestLen
4603 );
```

4604 **C\_Digest** digests data in a single part. *hSession* is the session's handle, *pData* points to the data;  
4605 *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest;  
4606 *pulDigestLen* points to the location that holds the length of the message digest.

4607 **C\_Digest** uses the convention described in Section 5.2 on producing output.

4608 The digest operation MUST have been initialized with **C\_DigestInit**. A call to **C\_Digest** always  
4609 terminates the active digest operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful  
4610 call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the message  
4611 digest.

4612 **C\_Digest** cannot be used to terminate a multi-part operation, and MUST be called after **C\_DigestInit**  
4613 without intervening **C\_DigestUpdate** calls.

4614 The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the  
4615 same location.

4616 **C\_Digest** is equivalent to a sequence of **C\_DigestUpdate** operations followed by **C\_DigestFinal**.

4617 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4618 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4619 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4620 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4621 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

4622 Example: see **C\_DigestFinal** for an example of similar functions.

### 4623 5.12.3 C\_DigestUpdate

```
4624 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (
4625     CK_SESSION_HANDLE hSession,
4626     CK_BYTE_PTR pPart,
4627     CK ULONG ulPartLen
4628 );
```

4629 **C\_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.  
4630 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4631 The message-digesting operation MUST have been initialized with **C\_DigestInit**. Calls to this function  
4632 and **C\_DigestKey** may be interspersed any number of times in any order. A call to **C\_DigestUpdate**  
4633 which results in an error terminates the current digest operation.

4634 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4635 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4636 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4637 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4638 CKR\_SESSION\_HANDLE\_INVALID.

4639 Example: see **C\_DigestFinal**.

### 4640 5.12.4 C\_DigestKey

```
4641 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (
4642     CK_SESSION_HANDLE hSession,
4643     CK_OBJECT_HANDLE hKey
4644 );
```

4645 **C\_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret  
4646 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

4647 The message-digesting operation MUST have been initialized with **C\_DigestInit**. Calls to this function  
4648 and **C\_DigestUpdate** may be interspersed any number of times in any order.

4649 If the value of the supplied key cannot be digested purely for some reason related to its length,  
4650 **C\_DigestKey** should return the error code CKR\_KEY\_SIZE\_RANGE.

4651 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4652 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4653 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID,  
4654 CKR\_KEY\_INDIGESTIBLE, CKR\_KEY\_SIZE\_RANGE, CKR\_OK,  
4655 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

4656 Example: see **C\_DigestFinal**.

### 4657 5.12.5 C\_DigestFinal

```
4658 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (
4659     CK_SESSION_HANDLE hSession,
4660     CK_BYTE_PTR pDigest,
4661     CK_ULONG_PTR pulDigestLen
4662 );
```

4663 **C\_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.  
4664 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;  
4665 *pulDigestLen* points to the location that holds the length of the message digest.

4666 **C\_DigestFinal** uses the convention described in Section 5.2 on producing output.

4667 The digest operation MUST have been initialized with **C\_DigestInit**. A call to **C\_DigestFinal** always  
4668 terminates the active digest operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful

4669 call (i.e., one which returns CKR\_OK) to determine the length of the buffer needed to hold the message  
4670 digest.

4671 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4672 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4673 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4674 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4675 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

4676 Example:

```
4677 CK_SESSION_HANDLE hSession;
4678 CK_OBJECT_HANDLE hKey;
4679 CK_MECHANISM mechanism = {
4680     CKM_MD5, NULL_PTR, 0
4681 };
4682 CK_BYTE data[] = {...};
4683 CK_BYTE digest[16];
4684 CK_ULONG ulDigestLen;
4685 CK_RV rv;
4686
4687 .
4688 .
4689 rv = C_DigestInit(hSession, &mechanism);
4690 if (rv != CKR_OK) {
4691     .
4692     .
4693 }
4694
4695 rv = C_DigestUpdate(hSession, data, sizeof(data));
4696 if (rv != CKR_OK) {
4697     .
4698     .
4699 }
4700
4701 rv = C_DigestKey(hSession, hKey);
4702 if (rv != CKR_OK) {
4703     .
4704     .
4705 }
4706
4707 ulDigestLen = sizeof(digest);
4708 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
4709 .
4710 .
```

4711 **5.13 Signing and MACing functions**

4712 Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations  
4713 also encompass message authentication codes).

4714 **5.13.1 C\_SignInit**

```
4715 CK_DECLARE_FUNCTION(CK_RV, C_SignInit) (
4716     CK_SESSION_HANDLE hSession,
4717     CK_MECHANISM_PTR pMechanism,
4718     CK_OBJECT_HANDLE hKey
4719 );
```

4720 **C\_SignInit** initializes a signature operation, where the signature is an appendix to the data. *hSession* is  
4721 the session's handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature  
4722 key.

4723 The **CKA\_SIGN** attribute of the signature key, which indicates whether the key supports signatures with  
4724 appendix, MUST be CK\_TRUE.

4725 After calling **C\_SignInit**, the application can either call **C\_Sign** to sign in a single part; or call  
4726 **C\_SignUpdate** one or more times, followed by **C\_SignFinal**, to sign data in multiple parts. The signature  
4727 operation is active until the application uses a call to **C\_Sign** or **C\_SignFinal** to actually obtain the  
4728 signature. To process additional data (in single or multiple parts), the application MUST call **C\_SignInit**  
4729 again.

4730 **C\_SignInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active signature operation.  
4731 If an operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED  
4732 must be returned.

4733 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4734 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4735 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4736 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,  
4737 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
4738 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
4739 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4740 CKR\_OPERATION\_CANCEL\_FAILED.

4741 Example: see **C\_SignFinal**.

4742 **5.13.2 C\_Sign**

```
4743 CK_DECLARE_FUNCTION(CK_RV, C_Sign) (
4744     CK_SESSION_HANDLE hSession,
4745     CK_BYTE_PTR pData,
4746     CK_ULONG ulDataLen,
4747     CK_BYTE_PTR pSignature,
4748     CK_ULONG_PTR pulSignatureLen
4749 );
```

4750 **C\_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the  
4751 session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the  
4752 location that receives the signature; *pulSignatureLen* points to the location that holds the length of the  
4753 signature.

4754 **C\_Sign** uses the convention described in Section 5.2 on producing output.

4755 The signing operation MUST have been initialized with **C\_SignInit**. A call to **C\_Sign** always terminates  
4756 the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (i.e.,  
4757 one which returns CKR\_OK) to determine the length of the buffer needed to hold the signature.

4758 **C\_Sign** cannot be used to terminate a multi-part operation, and MUST be called after **C\_SignInit** without  
4759 intervening **C\_SignUpdate** calls.

4760 For most mechanisms, **C\_Sign** is equivalent to a sequence of **C\_SignUpdate** operations followed by  
4761 **C\_SignFinal**.

4762 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4763 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
4764 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4765 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4766 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4767 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED,  
4768 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

4769 Example: see **C\_SignFinal** for an example of similar functions.

### 4770 5.13.3 C\_SignUpdate

```
4771 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate) (
4772     CK_SESSION_HANDLE hSession,
4773     CK_BYTE_PTR pPart,
4774     CK_ULONG ulPartLen
4775 );
```

4776 **C\_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is  
4777 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4778 The signature operation MUST have been initialized with **C\_SignInit**. This function may be called any  
4779 number of times in succession. A call to **C\_SignUpdate** which results in an error terminates the current  
4780 signature operation.

4781 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4782 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
4783 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
4784 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
4785 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4786 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

4787 Example: see **C\_SignFinal**.

### 4788 5.13.4 C\_SignFinal

```
4789 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal) (
4790     CK_SESSION_HANDLE hSession,
4791     CK_BYTE_PTR pSignature,
4792     CK_ULONG_PTR pulSignatureLen
4793 );
```

4794 **C\_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the  
4795 session's handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to  
4796 the location that holds the length of the signature.

4797 **C\_SignFinal** uses the convention described in Section 5.2 on producing output.

4798 The signing operation MUST have been initialized with **C\_SignInit**. A call to **C\_SignFinal** always  
4799 terminates the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful  
4800 call (i.e., one which returns CKR\_OK) to determine the length of the buffer needed to hold the signature.

4801 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4802 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
4803 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
4804 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
4805 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,

4806 CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED,  
4807 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

4808 Example:

```
4809 CK_SESSION_HANDLE hSession;
4810 CK_OBJECT_HANDLE hKey;
4811 CK_MECHANISM mechanism = {
4812     CKM_DES_MAC, NULL_PTR, 0
4813 };
4814 CK_BYTE data[] = {...};
4815 CK_BYTE mac[4];
4816 CK ULONG ulMacLen;
4817 CK_RV rv;
4818 .
4819 .
4820 .
4821 rv = C_SignInit(hSession, &mechanism, hKey);
4822 if (rv == CKR_OK) {
4823     rv = C_SignUpdate(hSession, data, sizeof(data));
4824     .
4825     .
4826     ulMacLen = sizeof(mac);
4827     rv = C_SignFinal(hSession, mac, &ulMacLen);
4828     .
4829     .
4830 }
```

#### 4831 5.13.5 C\_SignRecoverInit

```
4832 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit) (
4833     CK_SESSION_HANDLE hSession,
4834     CK_MECHANISM_PTR pMechanism,
4835     CK_OBJECT_HANDLE hKey
4836 );
```

4837 **C\_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature.  
4838 *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature  
4839 mechanism; *hKey* is the handle of the signature key.

4840 The **CKA\_SIGN\_RECOVER** attribute of the signature key, which indicates whether the key supports  
4841 signatures where the data can be recovered from the signature, MUST be CK\_TRUE.

4842 After calling **C\_SignRecoverInit**, the application may call **C\_SignRecover** to sign in a single part. The  
4843 signature operation is active until the application uses a call to **C\_SignRecover** to actually obtain the  
4844 signature. To process additional data in a single part, the application MUST call **C\_SignRecoverInit**  
4845 again.

4846 **C\_SignRecoverInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active signature  
4847 with data recovery operation. If an active operation has been initialized and it cannot be cancelled,  
4848 CKR\_OPERATION\_CANCEL\_FAILED must be returned.

4849 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4850 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,

```
4851 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,  
4852 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,  
4853 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,  
4854 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,  
4855 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,  
4856 CKR_OPERATION_CANCEL_FAILED.
```

4857 Example: see **C\_SignRecover**.

## 4858 5.13.6 C\_SignRecover

```
4859 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (  
4860     CK_SESSION_HANDLE hSession,  
4861     CK_BYTE_PTR pData,  
4862     CK_ULONG ulDataLen,  
4863     CK_BYTE_PTR pSignature,  
4864     CK_ULONG_PTR pulSignatureLen  
4865 );
```

4866 **C\_SignRecover** signs data in a single operation, where the data can be recovered from the signature.  
4867 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;  
4868 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that  
4869 holds the length of the signature.

4870 **C\_SignRecover** uses the convention described in Section 5.2 on producing output.

4871 The signing operation MUST have been initialized with **C\_SignRecoverInit**. A call to **C\_SignRecover**  
4872 always terminates the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a  
4873 successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the  
4874 signature.

4875 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4876 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
4877 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4878 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4879 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4880 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4881 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

4882 Example:

```
4883 CK_SESSION_HANDLE hSession;  
4884 CK_OBJECT_HANDLE hKey;  
4885 CK_MECHANISM mechanism = {  
4886     CKM_RSA_9796, NULL_PTR, 0  
4887 };  
4888 CK_BYTE data[] = {...};  
4889 CK_BYTE signature[128];  
4890 CK_ULONG ulSignatureLen;  
4891 CK_RV rv;  
4892  
4893 .  
4894 .  
4895 rv = C_SignRecoverInit(hSession, &mechanism, hKey);  
4896 if (rv == CKR_OK) {  
4897     ulSignatureLen = sizeof(signature);
```

```

4898     rv = C_SignRecover(
4899         hSession, data, sizeof(data), signature, &ulSignatureLen);
4900     if (rv == CKR_OK) {
4901         .
4902         .
4903     }
4904 }
4905

```

## 4906 5.14 Message-based signing and MACing functions

4907 Message-based signature refers to the process of signing multiple messages using the same signature  
 4908 mechanism and signature key.

4909 Cryptoki provides the following functions for signing messages (for the purposes of Cryptoki, these  
 4910 operations also encompass message authentication codes).

### 4911 5.14.1 C\_MessageSignInit

```

4912 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit) (
4913     CK_SESSION_HANDLE hSession,
4914     CK_MECHANISM_PTR pMechanism,
4915     CK_OBJECT_HANDLE hKey
4916 );

```

4917 **C\_MessageSignInit** initializes a message-based signature process, preparing a session for one or more  
 4918 signature operations (where the signature is an appendix to the data) that use the same signature  
 4919 mechanism and signature key. *hSession* is the session's handle; *pMechanism* points to the signature  
 4920 mechanism; *hKey* is the handle of the signature key.

4921 The **CKA\_SIGN** attribute of the signature key, which indicates whether the key supports signatures with  
 4922 appendix, MUST be CK\_TRUE.

4923 After calling **C\_MessageSignInit**, the application can either call **C\_SignMessage** to sign a message in a  
 4924 single part; or call **C\_SignMessageBegin**, followed by **C\_SignMessageNext** one or more times, to sign  
 4925 a message in multiple parts. This may be repeated several times. The message-based signature process  
 4926 is active until the application calls **C\_MessageSignFinal** to finish the message-based signature process.

4927 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
 4928 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
 4929 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
 4930 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,  
 4931 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
 4932 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
 4933 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### 4934 5.14.2 C\_SignMessage

```

4935 CK_DECLARE_FUNCTION(CK_RV, C_SignMessage) (
4936     CK_SESSION_HANDLE hSession,
4937     CK_VOID_PTR pParameter,
4938     CK_ULONG ulParameterLen,
4939     CK_BYTE_PTR pData,
4940     CK_ULONG ulDataLen,

```

```
4941     CK_BYTE_PTR pSignature,  
4942     CK ULONG_PTR pulSignatureLen  
4943 );
```

4944 **C\_SignMessage** signs a message in a single part, where the signature is an appendix to the message.  
4945 **C\_MessageSignInit** must previously been called on the session. *hSession* is the session's handle;  
4946 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature  
4947 operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location  
4948 that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.  
4949 Depending on the mechanism parameter passed to **C\_MessageSignInit**, *pParameter* may be either an  
4950 input or an output parameter.

4951 **C\_SignMessage** uses the convention described in Section 5.2 on producing output.  
4952 The message-based signing process MUST have been initialized with **C\_MessageSignInit**. A call to  
4953 **C\_SignMessage** begins and terminates a message signing operation unless it returns  
4954 CKR\_BUFFER\_TOO\_SMALL to determine the length of the buffer needed to hold the signature, or is a  
4955 successful call (i.e., one which returns CKR\_OK).

4956 **C\_SignMessage** cannot be called in the middle of a multi-part message signing operation.  
4957 **C\_SignMessage** does not finish the message-based signing process. Additional **C\_SignMessage** or  
4958 **C\_SignMessageBegin** and **C\_SignMessageNext** calls may be made on the session.  
4959 For most mechanisms, **C\_SignMessage** is equivalent to **C\_SignMessageBegin** followed by a sequence  
4960 of **C\_SignMessageNext** operations.

4961 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
4962 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
4963 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4964 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4965 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
4966 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED,  
4967 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### 4968 5.14.3 C\_SignMessageBegin

```
4969 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (  
4970     CK_SESSION_HANDLE hSession,  
4971     CK_VOID_PTR pParameter,  
4972     CK ULONG ulParameterLen  
4973 );
```

4974 **C\_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an  
4975 appendix to the message. **C\_MessageSignInit** must previously been called on the session. *hSession* is  
4976 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for  
4977 the message signature operation.

4978 Depending on the mechanism parameter passed to **C\_MessageSignInit**, *pParameter* may be either an  
4979 input or an output parameter.

4980 After calling **C\_SignMessageBegin**, the application should call **C\_SignMessageNext** one or more times  
4981 to sign the message in multiple parts. The message signature operation is active until the application  
4982 uses a call to **C\_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.  
4983 To process additional messages (in single or multiple parts), the application MUST call **C\_SignMessage**  
4984 or **C\_SignMessageBegin** again.

4985 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
4986 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
4987 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
4988 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,

4989 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
4990 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

#### 4991 5.14.4 C\_SignMessageNext

```
4992 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (
4993     CK_SESSION_HANDLE hSession,
4994     CK_VOID_PTR pParameter,
4995     CK ULONG ulParameterLen,
4996     CK_BYTE_PTR pDataPart,
4997     CK ULONG ulDataPartLen,
4998     CK_BYTE_PTR pSignature,
4999     CK ULONG_PTR pulSignatureLen
5000 );
```

5001 **C\_SignMessageNext** continues a multiple-part message signature operation, processing another data  
5002 part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the  
5003 session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any  
5004 mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the  
5005 data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the  
5006 location that holds the length of the signature.

5007 The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NUL  
5008 value (to receive the signature length) if this is the last data part.

5009 **C\_SignMessageNext** uses the convention described in Section 5.2 on producing output.

5010 The message signing operation MUST have been started with **C\_SignMessageBegin**. This function may  
5011 be called any number of times in succession. A call to **C\_SignMessageNext** with a NULL  
5012 *pulSignatureLen* which results in an error terminates the current message signature operation. A call to  
5013 **C\_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing  
5014 operation unless it returns CKR\_BUFFER\_TOO\_SMALL to determine the length of the buffer needed to  
5015 hold the signature, or is a successful call (i.e., one which returns CKR\_OK).

5016 Although the last **C\_SignMessageNext** call ends the signing of a message, it does not finish the  
5017 message-based signing process. Additional **C\_SignMessage** or **C\_SignMessageBegin** and  
5018 **C\_SignMessageNext** calls may be made on the session.

5019 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
5020 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
5021 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
5022 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
5023 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
5024 CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED,  
5025 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

#### 5026 5.14.5 C\_MessageSignFinal

```
5027 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) (
5028     CK_SESSION_HANDLE hSession
5029 );
```

5030 **C\_MessageSignFinal** finishes a message-based signing process. *hSession* is the session's handle.

5031 The message-based signing process MUST have been initialized with **C\_MessageSignInit**.

5032 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5033 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
5034 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,

5035 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
5036 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED,  
5037 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

## 5038 5.15 Functions for verifying signatures and MACs

5039 Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki,  
5040 these operations also encompass message authentication codes):

### 5041 5.15.1 C\_VerifyInit

```
5042 CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (
5043     CK_SESSION_HANDLE hSession,
5044     CK_MECHANISM_PTR pMechanism,
5045     CK_OBJECT_HANDLE hKey
5046 );
```

5047 **C\_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*  
5048 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;  
5049 *hKey* is the handle of the verification key.

5050 The **CKA\_VERIFY** attribute of the verification key, which indicates whether the key supports verification  
5051 where the signature is an appendix to the data, MUST be CK\_TRUE.

5052 After calling **C\_VerifyInit**, the application can either call **C\_Verify** to verify a signature on data in a single  
5053 part; or call **C\_VerifyUpdate** one or more times, followed by **C\_VerifyFinal**, to verify a signature on data  
5054 in multiple parts. The verification operation is active until the application calls **C\_Verify** or **C\_VerifyFinal**.  
5055 To process additional data (in single or multiple parts), the application MUST call **C\_VerifyInit** again.

5056 **C\_VerifyInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active verification  
5057 operation. If an active operation has been initialized and it cannot be cancelled,  
5058 CKR\_OPERATION\_CANCEL\_FAILED must be returned.

5059 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5060 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
5061 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
5062 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,  
5063 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
5064 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
5065 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
5066 CKR\_OPERATION\_CANCEL\_FAILED.

5067 Example: see **C\_VerifyFinal**.

### 5068 5.15.2 C\_Verify

```
5069 CK_DECLARE_FUNCTION(CK_RV, C_Verify) (
5070     CK_SESSION_HANDLE hSession,
5071     CK_BYTE_PTR pData,
5072     CK_ULONG ulDataLen,
5073     CK_BYTE_PTR pSignature,
5074     CK_ULONG ulSignatureLen
5075 );
```

5076 **C\_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.  
5077 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;  
5078 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5079 The verification operation MUST have been initialized with **C\_VerifyInit**. A call to **C\_Verify** always  
5080 terminates the active verification operation.

5081 A successful call to **C\_Verify** should return either the value CKR\_OK (indicating that the supplied  
5082 signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the

5083 signature can be seen to be invalid purely on the basis of its length, then  
5084 CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active signing operation  
5085 is terminated.

5086 **C\_Verify** cannot be used to terminate a multi-part operation, and MUST be called after **C\_VerifyInit**  
5087 without intervening **C\_VerifyUpdate** calls.

5088 For most mechanisms, **C\_Verify** is equivalent to a sequence of **C\_VerifyUpdate** operations followed by  
5089 **C\_VerifyFinal**.

5090 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID,  
5091 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
5092 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
5093 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
5094 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID,  
5095 CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

5096 Example: see **C\_VerifyFinal** for an example of similar functions.

### 5097 5.15.3 C\_VerifyUpdate

```
5098 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate) (
5099     CK_SESSION_HANDLE hSession,
5100     CK_BYTE_PTR pPart,
5101     CK_ULONG ulPartLen
5102 );
```

5103 **C\_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*  
5104 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

5105 The verification operation MUST have been initialized with **C\_VerifyInit**. This function may be called any  
5106 number of times in succession. A call to **C\_VerifyUpdate** which results in an error terminates the current  
5107 verification operation.

5108 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5109 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
5110 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
5111 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
5112 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
5113 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

5114 Example: see **C\_VerifyFinal**.

### 5115 5.15.4 C\_VerifyFinal

```
5116 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal) (
5117     CK_SESSION_HANDLE hSession,
5118     CK_BYTE_PTR pSignature,
5119     CK_ULONG ulSignatureLen
5120 );
```

5121 **C\_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the  
5122 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5123 The verification operation MUST have been initialized with **C\_VerifyInit**. A call to **C\_VerifyFinal** always  
5124 terminates the active verification operation.

5125 A successful call to **C\_VerifyFinal** should return either the value CKR\_OK (indicating that the supplied  
5126 signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the  
5127 signature can be seen to be invalid purely on the basis of its length, then  
5128 CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active verifying  
5129 operation is terminated.

5130 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5131 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,

```
5132 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,  
5133 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,  
5134 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,  
5135 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.
```

5136 Example:

```
5137 CK_SESSION_HANDLE hSession;  
5138 CK_OBJECT_HANDLE hKey;  
5139 CK_MECHANISM mechanism = {  
5140     CKM_DES_MAC, NULL_PTR, 0  
5141 };  
5142 CK_BYTE data[] = {...};  
5143 CK_BYTE mac[4];  
5144 CK_RV rv;  
5145 .  
5146 .  
5147 rv = C_VerifyInit(hSession, &mechanism, hKey);  
5149 if (rv == CKR_OK) {  
5150     rv = C_VerifyUpdate(hSession, data, sizeof(data));  
5151     .  
5152     .  
5153     rv = C_VerifyFinal(hSession, mac, sizeof(mac));  
5154     .  
5155     .  
5156 }
```

## 5157 5.15.5 C\_VerifyRecoverInit

```
5158 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit) (  
5159     CK_SESSION_HANDLE hSession,  
5160     CK_MECHANISM_PTR pMechanism,  
5161     CK_OBJECT_HANDLE hKey  
5162 );
```

5163 **C\_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the  
5164 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the  
5165 verification mechanism; *hKey* is the handle of the verification key.

5166 The **CKA\_VERIFY\_RECOVER** attribute of the verification key, which indicates whether the key supports  
5167 verification where the data is recovered from the signature, MUST be CK\_TRUE.

5168 After calling **C\_VerifyRecoverInit**, the application may call **C\_VerifyRecover** to verify a signature on  
5169 data in a single part. The verification operation is active until the application uses a call to  
5170 **C\_VerifyRecover** to actually obtain the recovered message.

5171 **C\_VerifyRecoverInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active verification  
5172 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,  
5173 CKR\_OPERATION\_CANCEL\_FAILED must be returned.

5174 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5175 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
5176 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
5177 CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID,

5178 CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID,  
5179 CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,  
5180 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
5181 CKR\_OPERATION\_CANCEL\_FAILED.

5182 Example: see **C\_VerifyRecover**.

## 5183 5.15.6 C\_VerifyRecover

```
5184 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover) {
5185     CK_SESSION_HANDLE hSession,
5186     CK_BYTE_PTR pSignature,
5187     CK_ULONG ulSignatureLen,
5188     CK_BYTE_PTR pData,
5189     CK_ULONG_PTR pulDataLen
5190 };
```

5191 **C\_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the  
5192 signature. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the  
5193 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*  
5194 points to the location that holds the length of the recovered data.

5195 **C\_VerifyRecover** uses the convention described in Section 5.2 on producing output.

5196 The verification operation MUST have been initialized with **C\_VerifyRecoverInit**. A call to  
5197 **C\_VerifyRecover** always terminates the active verification operation unless it returns  
5198 CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the  
5199 length of the buffer needed to hold the recovered data.

5200 A successful call to **C\_VerifyRecover** should return either the value CKR\_OK (indicating that the  
5201 supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is  
5202 invalid). If the signature can be seen to be invalid purely on the basis of its length, then  
5203 CKR\_SIGNATURE\_LEN\_RANGE should be returned. The return codes CKR\_SIGNATURE\_INVALID  
5204 and CKR\_SIGNATURE\_LEN\_RANGE have a higher priority than the return code  
5205 CKR\_BUFFER\_TOO\_SMALL, *i.e.*, if **C\_VerifyRecover** is supplied with an invalid signature, it will never  
5206 return CKR\_BUFFER\_TOO\_SMALL.

5207 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
5208 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE,  
5209 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
5210 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
5211 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED,  
5212 CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_SIGNATURE\_INVALID,  
5213 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

5214 Example:

```
5215 CK_SESSION_HANDLE hSession;
5216 CK_OBJECT_HANDLE hKey;
5217 CK_MECHANISM mechanism = {
5218     CKM_RSA_9796, NULL_PTR, 0
5219 };
5220 CK_BYTE data[] = {...};
5221 CK_ULONG ulDataLen;
5222 CK_BYTE signature[128];
5223 CK_RV rv;
5224
5225 .
```

```

5226 .
5227     rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
5228     if (rv == CKR_OK) {
5229         ulDataLen = sizeof(data);
5230         rv = C_VerifyRecover(
5231             hSession, signature, sizeof(signature), data, &ulDataLen);
5232     .
5233     .
5234 }

```

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

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

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

### 5.16.1 C\_MessageVerifyInit

```

5241 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit) (
5242     CK_SESSION_HANDLE hSession,
5243     CK_MECHANISM_PTR pMechanism,
5244     CK_OBJECT_HANDLE hKey
5245 );

```

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

The **CKA\_VERIFY** attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, MUST be CK\_TRUE.

After calling **C\_MessageVerifyInit**, the application can either call **C\_VerifyMessage** to verify a signature on a message in a single part; or call **C\_VerifyMessageBegin**, followed by **C\_VerifyMessageNext** one or more times, to verify a signature on a message in multiple parts. This may be repeated several times. The message-based verification process is active until the application calls **C\_MessageVerifyFinal** to finish the message-based verification process.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### 5.16.2 C\_VerifyMessage

```

5265 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
5266     CK_SESSION_HANDLE hSession,
5267     CK_VOID_PTR pParameter,
5268     CK_ULONG ulParameterLen,
5269     CK_BYTE_PTR pData,

```

```

5270     CK ULONG ulDataLen,
5271     CK_BYTE_PTR pSignature,
5272     CK ULONG ulSignatureLen
5273 ) ;

```

5274 **C\_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an appendix to the data. **C\_MessageVerifyInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5275 Unlike the *pParameter* parameter of **C\_SignMessage**, *pParameter* is always an input parameter.

5276 The message-based verification process MUST have been initialized with **C\_MessageVerifyInit**. A call to **C\_VerifyMessage** starts and terminates a message verification operation.

5277 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 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.

5278 **C\_VerifyMessage** does not finish the message-based verification process. Additional **C\_VerifyMessage** or **C\_VerifyMessageBegin** and **C\_VerifyMessageNext** calls may be made on the session.

5279 For most mechanisms, **C\_VerifyMessage** is equivalent to **C\_VerifyMessageBegin** followed by a sequence of **C\_VerifyMessageNext** operations.

5280 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID,

5281 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,

5282 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,

5283 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,

5284 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID,

5285 CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### 5.16.3 C\_VerifyMessageBegin

```

5297 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (
5298     CK_SESSION_HANDLE hSession,
5299     CK_VOID_PTR pParameter,
5300     CK ULONG ulParameterLen
5301 ) ;

```

5302 **C\_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an appendix to the message. **C\_MessageVerifyInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation.

5303 Unlike the *pParameter* parameter of **C\_SignMessageBegin**, *pParameter* is always an input parameter.

5304 After calling **C\_VerifyMessageBegin**, the application should call **C\_VerifyMessageNext** one or more times to verify a signature on a message in multiple parts. The message verification operation is active until the application calls **C\_VerifyMessageNext** with a non-NULL *pSignature*. To process additional messages (in single or multiple parts), the application MUST call **C\_VerifyMessage** or **C\_VerifyMessageBegin** again.

5305 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,

5306 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,

5307 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,

5308 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED,

5309 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

5317 **5.16.4 C\_VerifyMessageNext**

```
5318 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (
5319     CK_SESSION_HANDLE hSession,
5320     CK_VOID_PTR pParameter,
5321     CK ULONG ulParameterLen,
5322     CK_BYTE_PTR pDataPart,
5323     CK ULONG ulDataPartLen,
5324     CK_BYTE_PTR pSignature,
5325     CK ULONG ulSignatureLen
5326 );
```

5327 **C\_VerifyMessageNext** continues a multiple-part message verification operation, processing another data  
5328 part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the  
5329 session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the  
5330 message verification operation, *pDataPart* points to the data part; *ulPartLen* is the length of the data part;  
5331 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5332 The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value  
5333 (pointing to the signature to verify) if this is the last data part.

5334 The message verification operation MUST have been started with **C\_VerifyMessageBegin**. This function  
5335 may be called any number of times in succession. A call to **C\_VerifyMessageNext** with a NULL  
5336 *pSignature* which results in an error terminates the current message verification operation. A call to  
5337 **C\_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification  
5338 operation.

5339 A successful call to **C\_VerifyMessageNext** with a non-NULL *pSignature* should return either the value  
5340 CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that  
5341 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its  
5342 length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active  
5343 message verifying operation is terminated.

5344 Although the last **C\_VerifyMessageNext** call ends the verification of a message, it does not finish the  
5345 message-based verification process. Additional **C\_VerifyMessage** or **C\_VerifyMessageBegin** and  
5346 **C\_VerifyMessageNext** calls may be made on the session.

5347 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5348 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
5349 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
5350 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
5351 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID,  
5352 CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

5353 **5.16.5 C\_MessageVerifyFinal**

```
5354 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal) (
5355     CK_SESSION_HANDLE hSession
5356 );
```

5357 **C\_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session's handle.

5358 The message-based verification process MUST have been initialized with **C\_MessageVerifyInit**.

5359 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
5360 CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
5361 CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
5362 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,

5363 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
5364 CKR\_TOKEN\_RESOURCE\_EXCEEDED.

## 5365 5.17 Dual-function cryptographic functions

5366 Cryptoki provides the following functions to perform two cryptographic operations “simultaneously” within  
5367 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and  
5368 from a token.

### 5369 5.17.1 C\_DigestEncryptUpdate

```
5370 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (
5371     CK_SESSION_HANDLE hSession,
5372     CK_BYTE_PTR pPart,
5373     CK ULONG ulPartLen,
5374     CK_BYTE_PTR pEncryptedPart,
5375     CK ULONG_PTR pulEncryptedPartLen
5376 );
```

5377 **C\_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another  
5378 data part. *hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the  
5379 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;  
5380 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5381 **C\_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a  
5382 **C\_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because  
5383 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire  
5384 encrypted part output), then no plaintext is passed to the active digest operation.

5385 Digest and encryption operations MUST both be active (they MUST have been initialized with  
5386 **C\_DigestInit** and **C\_EncryptInit**, respectively). This function may be called any number of times in  
5387 succession, and may be interspersed with **C\_DigestUpdate**, **C\_DigestKey**, and **C\_EncryptUpdate** calls  
5388 (it would be somewhat unusual to intersperse calls to **C\_DigestEncryptUpdate** with calls to  
5389 **C\_DigestKey**, however).

5390 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
5391 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR,  
5392 CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED,  
5393 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK,  
5394 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.  
5395 Example:

```
5396 #define BUF_SZ 512
5397
5398 CK_SESSION_HANDLE hSession;
5399 CK_OBJECT_HANDLE hKey;
5400 CK_BYTE iv[8];
5401 CK_MECHANISM digestMechanism = {
5402     CKM_MD5, NULL_PTR, 0
5403 };
5404 CK_MECHANISM encryptionMechanism = {
5405     CKM_DES_ECB, iv, sizeof(iv)
5406 };
5407 CK_BYTE encryptedData[BUF_SZ];
5408 CK ULONG ulEncryptedDataLen;
```

```

5409 CK_BYTE digest[16];
5410 CK_ULONG ulDigestLen;
5411 CK_BYTE data[(2*BUF_SZ)+8];
5412 CK_RV rv;
5413 int i;
5414 .
5415 .
5416 .
5417 memset(iv, 0, sizeof(iv));
5418 memset(data, 'A', ((2*BUF_SZ)+5));
5419 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
5420 if (rv != CKR_OK) {
5421 .
5422 .
5423 }
5424 rv = C_DigestInit(hSession, &digestMechanism);
5425 if (rv != CKR_OK) {
5426 .
5427 .
5428 }
5429 .
5430 ulEncryptedDataLen = sizeof(encryptedData);
5431 rv = C_DigestEncryptUpdate(
5432     hSession,
5433     &data[0], BUF_SZ,
5434     encryptedData, &ulEncryptedDataLen);
5435 .
5436 .
5437 ulEncryptedDataLen = sizeof(encryptedData);
5438 rv = C_DigestEncryptUpdate(
5439     hSession,
5440     &data[BUF_SZ], BUF_SZ,
5441     encryptedData, &ulEncryptedDataLen);
5442 .
5443 .
5444 .
5445 /*
5446 * The last portion of the buffer needs to be
5447 * handled with separate calls to deal with
5448 * padding issues in ECB mode
5449 */
5450 .
5451 /* First, complete the digest on the buffer */

```

```

5452 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5453 .
5454 .
5455 ulDigestLen = sizeof(digest);
5456 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5457 .
5458 .
5459
5460 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
5461 for(i=0;i<3;i++)
5462     data[((BUF_SZ*2)+5)+i] = 0x00;
5463
5464 /* Now, get second-to-last piece of ciphertext */
5465 ulEncryptedDataLen = sizeof(encryptedData);
5466 rv = C_EncryptUpdate(
5467     hSession,
5468     &data[BUF_SZ*2], 8,
5469     encryptedData, &ulEncryptedDataLen);
5470 .
5471 .
5472
5473 /* Get last piece of ciphertext (should have length 0, here) */
5474 ulEncryptedDataLen = sizeof(encryptedData);
5475 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5476 .
5477 .

```

## 5478 5.17.2 C\_DecryptDigestUpdate

```

5479 CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
5480     CK_SESSION_HANDLE hSession,
5481     CK_BYTE_PTR pEncryptedPart,
5482     CK_ULONG ulEncryptedPartLen,
5483     CK_BYTE_PTR pPart,
5484     CK ULONG_PTR pulPartLen
5485 );

```

5486 **C\_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation,  
 5487 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted  
 5488 data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that  
 5489 receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered  
 5490 data part.

5491 **C\_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a  
 5492 **C\_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because  
 5493 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire decrypted part  
 5494 output), then no plaintext is passed to the active digest operation.

5495 Decryption and digesting operations MUST both be active (they MUST have been initialized with  
 5496 **C\_DecryptInit** and **C\_DigestInit**, respectively). This function may be called any number of times in

5497 succession, and may be interspersed with **C\_DecryptUpdate**, **C\_DigestUpdate**, and **C\_DigestKey** calls  
5498 (it would be somewhat unusual to intersperse calls to **C\_DigestEncryptUpdate** with calls to  
5499 **C\_DigestKey**, however).

5500 Use of **C\_DecryptDigestUpdate** involves a pipelining issue that does not arise when using  
5501 **C\_DigestEncryptUpdate**, the “inverse function” of **C\_DecryptDigestUpdate**. This is because when  
5502 **C\_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting  
5503 operation and the active encryption operation; however, when **C\_DecryptDigestUpdate** is called, the  
5504 input passed to the active digesting operation is the *output* of the active decryption operation. This issue  
5505 comes up only when the mechanism used for decryption performs padding.

5506 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with  
5507 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this  
5508 ciphertext and digest the original plaintext thereby obtained.

5509 After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES  
5510 blocks) into **C\_DecryptDigestUpdate**. **C\_DecryptDigestUpdate** returns exactly 16 bytes of plaintext,  
5511 since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of  
5512 ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

5513 Since there is no more ciphertext, the application calls **C\_DecryptFinal**. This tells Cryptoki that there’s  
5514 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active  
5515 decryption and digesting operations are linked *only* through the **C\_DecryptDigestUpdate** call, these 2  
5516 bytes of plaintext are *not* passed on to be digested.

5517 A call to **C\_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the*  
5518 *plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C\_DigestFinal** is called,  
5519 the last 2 bytes of plaintext get passed into the active digesting operation via a **C\_DigestUpdate** call.

5520 Because of this, it is critical that when an application uses a padded decryption mechanism with  
5521 **C\_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting  
5522 operation. *Extreme caution is warranted when using a padded decryption mechanism with*  
5523 **C\_DecryptDigestUpdate**.

5524 Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
5525 CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
5526 CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID,  
5527 CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
5528 CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED,  
5529 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

5530 Example:

```
5531 #define BUF_SZ 512
5532
5533 CK_SESSION_HANDLE hSession;
5534 CK_OBJECT_HANDLE hKey;
5535 CK_BYTE iv[8];
5536 CK_MECHANISM decryptionMechanism = {
5537     CKM_DES_ECB, iv, sizeof(iv)
5538 };
5539 CK_MECHANISM digestMechanism = {
5540     CKM_MD5, NULL_PTR, 0
5541 };
5542 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5543 CK_BYTE digest[16];
5544 CK_ULONG ulDigestLen;
5545 CK_BYTE data[BUF_SZ];
```

```

5546 CK ULONG ulDataLen, ulLastUpdateSize;
5547 CK_RV rv;
5548 .
5549 .
5550 memset(iv, 0, sizeof(iv));
5551 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5552 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
5553 if (rv != CKR_OK) {
5554 .
5555 .
5556 }
5557 rv = C_DigestInit(hSession, &digestMechanism);
5558 if (rv != CKR_OK) {
5559 .
5560 .
5561 }
5562 }
5563 .
5564 ulDataLen = sizeof(data);
5565 rv = C_DecryptDigestUpdate(
5566     hSession,
5567     &encryptedData[0], BUF_SZ,
5568     data, &ulDataLen);
5569 .
5570 .
5571 ulDataLen = sizeof(data);
5572 rv = C_DecryptDigestUpdate(
5573     hSession,
5574     &encryptedData[BUF_SZ], BUF_SZ,
5575     data, &ulDataLen);
5576 .
5577 .
5578 /*
5579 * The last portion of the buffer needs to be handled with
5580 * separate calls to deal with padding issues in ECB mode
5581 */
5582 */
5583 /*
5584 * First, complete the decryption of the buffer */
5585 ullLastUpdateSize = sizeof(data);
5586 rv = C_DecryptUpdate(
5587     hSession,
5588     &encryptedData[BUF_SZ*2], 8,

```

```

5589     data, &ulLastUpdateSize);
5590 .
5591 .
5592 /* Get last piece of plaintext (should have length 0, here) */
5593 ulDataLen = sizeof(data)-ulLastUpdateSize;
5594 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5595 if (rv != CKR_OK) {
5596 .
5597 .
5598 }
5599 .
5600 /* Digest last bit of plaintext */
5601 rv = C_DigestUpdate(hSession, data, 5);
5602 if (rv != CKR_OK) {
5603 .
5604 .
5605 }
5606 ulDigestLen = sizeof(digest);
5607 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5608 if (rv != CKR_OK) {
5609 .
5610 .
5611 }

```

### 5612 5.17.3 C\_SignEncryptUpdate

```

5613 CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate) (
5614     CK_SESSION_HANDLE hSession,
5615     CK_BYTE_PTR pPart,
5616     CK ULONG ulPartLen,
5617     CK_BYTE_PTR pEncryptedPart,
5618     CK ULONG_PTR pulEncryptedPartLen
5619 );

```

5620 **C\_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation,  
 5621 processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is  
 5622 the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted  
 5623 data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5624 **C\_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a  
 5625 **C\_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because  
 5626 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire  
 5627 encrypted part output), then no plaintext is passed to the active signing operation.

5628 Signature and encryption operations MUST both be active (they MUST have been initialized with  
 5629 **C\_SignInit** and **C\_EncryptInit**, respectively). This function may be called any number of times in  
 5630 succession, and may be interspersed with **C\_SignUpdate** and **C\_EncryptUpdate** calls.

5631 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,  
 5632 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,  
 5633 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,  
 5634 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,

5635 CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID,  
5636 CKR\_USER\_NOT\_LOGGED\_IN.

5637 Example:

```
5638 #define BUF_SZ 512
5639
5640 CK_SESSION_HANDLE hSession;
5641 CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
5642 CK_BYTE iv[8];
5643 CK_MECHANISM signMechanism = {
5644     CKM_DES_MAC, NULL_PTR, 0
5645 };
5646 CK_MECHANISM encryptionMechanism = {
5647     CKM_DES_ECB, iv, sizeof(iv)
5648 };
5649 CK_BYTE encryptedData[BUF_SZ];
5650 CK_ULONG ulEncryptedDataLen;
5651 CK_BYTE MAC[4];
5652 CK_ULONG ulMacLen;
5653 CK_BYTE data[(2*BUF_SZ)+8];
5654 CK_RV rv;
5655 int i;
5656
5657 .
5658 .
5659 memset(iv, 0, sizeof(iv));
5660 memset(data, 'A', ((2*BUF_SZ)+5));
5661 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
5662 if (rv != CKR_OK) {
5663     .
5664     .
5665 }
5666 rv = C_SignInit(hSession, &signMechanism, hMacKey);
5667 if (rv != CKR_OK) {
5668     .
5669     .
5670 }
5671
5672 ulEncryptedDataLen = sizeof(encryptedData);
5673 rv = C_SignEncryptUpdate(
5674     hSession,
5675     &data[0], BUF_SZ,
5676     encryptedData, &ulEncryptedDataLen);
5677 .
```

```

5678 .
5679 ulEncryptedDataLen = sizeof(encryptedData);
5680 rv = C_SignEncryptUpdate(
5681     hSession,
5682     &data[BUF_SZ], BUF_SZ,
5683     encryptedData, &ulEncryptedDataLen);
5684 .
5685 .
5686 .
5687 /*
5688 * The last portion of the buffer needs to be handled with
5689 * separate calls to deal with padding issues in ECB mode
5690 */
5691 .
5692 /* First, complete the signature on the buffer */
5693 rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
5694 .
5695 .
5696 ulMacLen = sizeof(MAC);
5697 rv = C_SignFinal(hSession, MAC, &ulMacLen);
5698 .
5699 .
5700 .
5701 /* Then pad last part with 3 0x00 bytes, and complete encryption */
5702 for(i=0;i<3;i++)
5703     data[((BUF_SZ*2)+5)+i] = 0x00;
5704 .
5705 /* Now, get second-to-last piece of ciphertext */
5706 ulEncryptedDataLen = sizeof(encryptedData);
5707 rv = C_EncryptUpdate(
5708     hSession,
5709     &data[BUF_SZ*2], 8,
5710     encryptedData, &ulEncryptedDataLen);
5711 .
5712 .
5713 .
5714 /* Get last piece of ciphertext (should have length 0, here) */
5715 ulEncryptedDataLen = sizeof(encryptedData);
5716 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5717 .
5718 .

```

5719 **5.17.4 C\_DecryptVerifyUpdate**

```
5720 CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate) (
5721     CK_SESSION_HANDLE hSession,
5722     CK_BYTE_PTR pEncryptedPart,
5723     CK ULONG ulEncryptedPartLen,
5724     CK_BYTE_PTR pPart,
5725     CK ULONG_PTR pulPartLen
5726 );
```

5727 **C\_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation,  
5728 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted  
5729 data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the  
5730 recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

5731 **C\_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a  
5732 **C\_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because  
5733 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part  
5734 output), then no plaintext is passed to the active verification operation.

5735 Decryption and signature operations MUST both be active (they MUST have been initialized with  
5736 **C\_DecryptInit** and **C\_VerifyInit**, respectively). This function may be called any number of times in  
5737 succession, and may be interspersed with **C\_DecryptUpdate** and **C\_VerifyUpdate** calls.

5738 Use of **C\_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using  
5739 **C\_SignEncryptUpdate**, the "inverse function" of **C\_DecryptVerifyUpdate**. This is because when  
5740 **C\_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation  
5741 and the active encryption operation; however, when **C\_DecryptVerifyUpdate** is called, the input passed  
5742 to the active verifying operation is the *output* of the active decryption operation. This issue comes up only  
5743 when the mechanism used for decryption performs padding.

5744 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with  
5745 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this  
5746 ciphertext and verify a signature on the original plaintext thereby obtained.

5747 After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3  
5748 DES blocks) into **C\_DecryptVerifyUpdate**. **C\_DecryptVerifyUpdate** returns exactly 16 bytes of  
5749 plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of  
5750 ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

5751 Since there is no more ciphertext, the application calls **C\_DecryptFinal**. This tells Cryptoki that there's  
5752 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active  
5753 decryption and verification operations are linked *only* through the **C\_DecryptVerifyUpdate** call, these 2  
5754 bytes of plaintext are *not* passed on to the verification mechanism.

5755 A call to **C\_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature  
5756 on the *first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C\_VerifyFinal** is  
5757 called, the last 2 bytes of plaintext get passed into the active verification operation via a **C\_VerifyUpdate**  
5758 call.

5759 Because of this, it is critical that when an application uses a padded decryption mechanism with  
5760 **C\_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active  
5761 verification operation. *Extreme caution is warranted when using a padded decryption mechanism with*  
5762 **C\_DecryptVerifyUpdate**.

5763 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,  
5764 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,  
5765 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_ENCRYPTED_DATA_INVALID`,  
5766 `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`,  
5767 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`,  
5768 `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

5769 Example:

```
5770 #define BUF_SZ 512
```

```

5771
5772     CK_SESSION_HANDLE hSession;
5773     CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
5774     CK_BYTE iv[8];
5775     CK_MECHANISM decryptionMechanism = {
5776         CKM_DES_ECB, iv, sizeof(iv)
5777     };
5778     CK_MECHANISM verifyMechanism = {
5779         CKM_DES_MAC, NULL_PTR, 0
5780     };
5781     CK_BYTE encryptedData[(2*BUF_SZ)+8];
5782     CK_BYTE MAC[4];
5783     CK ULONG ulMacLen;
5784     CK_BYTE data[BUF_SZ];
5785     CK ULONG ulDataLen, ulLastUpdateSize;
5786     CK_RV rv;
5787
5788 .
5789 .
5790     memset(iv, 0, sizeof(iv));
5791     memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5792     rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
5793     if (rv != CKR_OK) {
5794     .
5795     .
5796 }
5797     rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
5798     if (rv != CKR_OK) {
5799     .
5800     .
5801 }
5802
5803     ulDataLen = sizeof(data);
5804     rv = C_DecryptVerifyUpdate(
5805         hSession,
5806         &encryptedData[0], BUF_SZ,
5807         data, &ulDataLen);
5808 .
5809 .
5810     ulDataLen = sizeof(data);
5811     rv = C_DecryptVerifyUpdate(
5812         hSession,
5813         &encryptedData[BUF_SZ], BUF_SZ,

```

```

5814     data, &ulDataLen);
5815 .
5816 .
5817 /*
5818 * The last portion of the buffer needs to be handled with
5819 * separate calls to deal with padding issues in ECB mode
5820 */
5821
5822
5823 /* First, complete the decryption of the buffer */
5824 ulLastUpdateSize = sizeof(data);
5825 rv = C_DecryptUpdate(
5826     hSession,
5827     &encryptedData[BUF_SZ*2], 8,
5828     data, &ulLastUpdateSize);
5829 .
5830 .
5831 /* Get last little piece of plaintext. Should have length 0 */
5832 ulDataLen = sizeof(data)-ulLastUpdateSize;
5833 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5834 if (rv != CKR_OK) {
5835     .
5836     .
5837 }
5838
5839 /* Send last bit of plaintext to verification operation */
5840 rv = C_VerifyUpdate(hSession, data, 5);
5841 if (rv != CKR_OK) {
5842     .
5843     .
5844 }
5845 rv = C_VerifyFinal(hSession, MAC, ulMacLen);
5846 if (rv == CKR_SIGNATURE_INVALID) {
5847     .
5848     .
5849 }

```

## 5850 5.18 Key management functions

5851 Cryptoki provides the following functions for key management:

### 5852 5.18.1 C\_GenerateKey

```

5853 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey) (
5854     CK_SESSION_HANDLE hSession

```

```
5855     CK_MECHANISM_PTR pMechanism,  
5856     CK_ATTRIBUTE_PTR pTemplate,  
5857     CK_ULONG ulCount,  
5858     CK_OBJECT_HANDLE_PTR phKey  
5859 );
```

5860 **C\_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is  
5861 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template  
5862 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*  
5863 points to the location that receives the handle of the new key or set of domain parameters.

5864 If the generation mechanism is for domain parameter generation, the **CKA\_CLASS** attribute will have the  
5865 value **CKO\_DOMAIN\_PARAMETERS**; otherwise, it will have the value **CKO\_SECRET\_KEY**.

5866 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the  
5867 template does not need to supply a key type. If it does supply a key type which is inconsistent with the  
5868 generation mechanism, **C\_GenerateKey** fails and returns the error code  
5869 **CKR\_TEMPLATE\_INCONSISTENT**. The **CKA\_CLASS** attribute is treated similarly.

5870 If a call to **C\_GenerateKey** cannot support the precise template supplied to it, it will fail and return without  
5871 creating an object.

5872 The object created by a successful call to **C\_GenerateKey** will have its **CKA\_LOCAL** attribute set to  
5873 **CK\_TRUE**. In addition, the object created will have a value for **CKA\_UNIQUE\_ID** generated and  
5874 assigned (See Section 4.4.1).

5875 Return values: **CKR\_ARGUMENTS\_BAD**, **CKR\_ATTRIBUTE\_READ\_ONLY**,  
5876 **CKR\_ATTRIBUTE\_TYPE\_INVALID**, **CKR\_ATTRIBUTE\_VALUE\_INVALID**,  
5877 **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_CURVE\_NOT\_SUPPORTED**, **CKR\_DEVICE\_ERROR**,  
5878 **CKR\_DEVICE\_MEMORY**, **CKR\_DEVICE\_REMOVED**, **CKR\_FUNCTION\_CANCELED**,  
5879 **CKR\_FUNCTION\_FAILED**, **CKR\_GENERAL\_ERROR**, **CKR\_HOST\_MEMORY**,  
5880 **CKR\_MECHANISM\_INVALID**, **CKR\_MECHANISM\_PARAM\_INVALID**, **CKR\_OK**,  
5881 **CKR\_OPERATION\_ACTIVE**, **CKR\_PIN\_EXPIRED**, **CKR\_SESSION\_CLOSED**,  
5882 **CKR\_SESSION\_HANDLE\_INVALID**, **CKR\_SESSION\_READ\_ONLY**, **CKR\_TEMPLATE\_INCOMPLETE**,  
5883 **CKR\_TEMPLATE\_INCONSISTENT**, **CKR\_TOKEN\_WRITE\_PROTECTED**,  
5884 **CKR\_USER\_NOT\_LOGGED\_IN**.

5885 Example:

```
5886 CK_SESSION_HANDLE hSession;  
5887 CK_OBJECT_HANDLE hKey;  
5888 CK_MECHANISM mechanism = {  
5889     CKM_DES_KEY_GEN, NULL_PTR, 0  
5890 };  
5891 CK_RV rv;  
5892 .  
5893 .  
5894 .  
5895 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);  
5896 if (rv == CKR_OK) {  
5897     .  
5898     .  
5899 }
```

## 5.18.2 C\_GenerateKeyPair

```
5901 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair) (
5902     CK_SESSION_HANDLE hSession,
5903     CK_MECHANISM_PTR pMechanism,
5904     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
5905     CK ULONG ulPublicKeyAttributeCount,
5906     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
5907     CK ULONG ulPrivateKeyAttributeCount,
5908     CK_OBJECT_HANDLE_PTR phPublicKey,
5909     CK_OBJECT_HANDLE_PTR phPrivateKey
5910 );
```

5911 **C\_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the  
5912 session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to  
5913 the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key  
5914 template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is  
5915 the number of attributes in the private-key template; *phPublicKey* points to the location that receives the  
5916 handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new  
5917 private key.

5918 Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates  
5919 do not need to supply key types. If one of the templates does supply a key type which is inconsistent with  
5920 the key generation mechanism, **C\_GenerateKeyPair** fails and returns the error code  
5921 **CKR\_TEMPLATE\_INCONSISTENT**. The **CKA\_CLASS** attribute is treated similarly.

5922 If a call to **C\_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return  
5923 without creating any key objects.

5924 A call to **C\_GenerateKeyPair** will never create just one key and return. A call can fail, and create no  
5925 keys; or it can succeed, and create a matching public/private key pair.

5926 The key objects created by a successful call to **C\_GenerateKeyPair** will have their **CKA\_LOCAL**  
5927 attributes set to **CK\_TRUE**. In addition, the key objects created will both have values for  
5928 **CKA\_UNIQUE\_ID** generated and assigned (See Section 4.4.1).

5929 Note carefully the order of the arguments to **C\_GenerateKeyPair**. The last two arguments do not have  
5930 the same order as they did in the original Cryptoki Version 1.0 document. The order of these two  
5931 arguments has caused some unfortunate confusion.

5932 Return values: **CKR\_ARGUMENTS\_BAD**, **CKR\_ATTRIBUTE\_READ\_ONLY**,  
5933 **CKR\_ATTRIBUTE\_TYPE\_INVALID**, **CKR\_ATTRIBUTE\_VALUE\_INVALID**,  
5934 **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_CURVE\_NOT\_SUPPORTED**, **CKR\_DEVICE\_ERROR**,  
5935 **CKR\_DEVICE\_MEMORY**, **CKR\_DEVICE\_REMOVED**, **CKR\_DOMAIN\_PARAMS\_INVALID**,  
5936 **CKR\_FUNCTION\_CANCELED**, **CKR\_FUNCTION\_FAILED**, **CKR\_GENERAL\_ERROR**,  
5937 **CKR\_HOST\_MEMORY**, **CKR\_MECHANISM\_INVALID**, **CKR\_MECHANISM\_PARAM\_INVALID**,  
5938 **CKR\_OK**, **CKR\_OPERATION\_ACTIVE**, **CKR\_PIN\_EXPIRED**, **CKR\_SESSION\_CLOSED**,  
5939 **CKR\_SESSION\_HANDLE\_INVALID**, **CKR\_SESSION\_READ\_ONLY**, **CKR\_TEMPLATE\_INCOMPLETE**,  
5940 **CKR\_TEMPLATE\_INCONSISTENT**, **CKR\_TOKEN\_WRITE\_PROTECTED**,  
5941 **CKR\_USER\_NOT\_LOGGED\_IN**.

5942 Example:

```
5943 CK_SESSION_HANDLE hSession;
5944 CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
5945 CK_MECHANISM mechanism = {
5946     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
5947 };
5948 CK ULONG modulusBits = 3072;
5949 CK_BYTE publicExponent[] = { 3 };
```

```

5950 CK_BYTE subject[] = {...};
5951 CK_BYTE id[] = {123};
5952 CK_BBOOL true = CK_TRUE;
5953 CK_ATTRIBUTE publicKeyTemplate[] = {
5954     {CKA_ENCRYPT, &true, sizeof(true)},
5955     {CKA_VERIFY, &true, sizeof(true)},
5956     {CKA_WRAP, &true, sizeof(true)},
5957     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
5958     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}}
5959 };
5960 CK_ATTRIBUTE privateKeyTemplate[] = {
5961     {CKA_TOKEN, &true, sizeof(true)},
5962     {CKA_PRIVATE, &true, sizeof(true)},
5963     {CKA SUBJECT, subject, sizeof(subject)},
5964     {CKA_ID, id, sizeof(id)},
5965     {CKA_SENSITIVE, &true, sizeof(true)},
5966     {CKA_DECRYPT, &true, sizeof(true)},
5967     {CKA_SIGN, &true, sizeof(true)},
5968     {CKA_UNWRAP, &true, sizeof(true)}}
5969 };
5970 CK_RV rv;
5971
5972 rv = C_GenerateKeyPair(
5973     hSession, &mechanism,
5974     publicKeyTemplate, 5,
5975     privateKeyTemplate, 8,
5976     &hPublicKey, &hPrivateKey);
5977 if (rv == CKR_OK) {
5978 .
5979 .
5980 }

```

### 5981 5.18.3 C\_WrapKey

```

5982 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
5983     CK_SESSION_HANDLE hSession,
5984     CK_MECHANISM_PTR pMechanism,
5985     CK_OBJECT_HANDLE hWrappingKey,
5986     CK_OBJECT_HANDLE hKey,
5987     CK_BYTE_PTR pWrappedKey,
5988     CK ULONG_PTR pulWrappedKeyLen
5989 );

```

5990 **C\_WrapKey** wraps (*i.e.*, encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism* points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

5994     **C\_WrapKey** uses the convention described in Section 5.2 on producing output.  
 5995     The **CKA\_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping,  
 5996     MUST be CK\_TRUE. The **CKA\_EXTRACTABLE** attribute of the key to be wrapped MUST also be  
 5997     CK\_TRUE.  
 5998     If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its  
 5999     **CKA\_EXTRACTABLE** attribute set to CK\_TRUE, then **C\_WrapKey** fails with error code  
 6000     CKR\_KEY\_NOT\_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism  
 6001     solely because of its length, then **C\_WrapKey** fails with error code CKR\_KEY\_SIZE\_RANGE.  
 6002     **C\_WrapKey** can be used in the following situations:  
 6003         • To wrap any secret key with a public key that supports encryption and decryption.  
 6004         • To wrap any secret key with any other secret key. Consideration MUST be given to key size and  
 6005             mechanism strength or the token may not allow the operation.  
 6006         • To wrap a private key with any secret key.  
 6007     Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.  
 6008     To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute  
 6009     **CKA\_WRAP\_TEMPLATE** can be used on the wrapping key to specify an attribute set that will be  
 6010     compared against the attributes of the key to be wrapped. If all attributes match according to the  
 6011     **C\_FindObject** rules of attribute matching then the wrap will proceed. The value of this attribute is an  
 6012     attribute template and the size is the number of items in the template times the size of **CK\_ATTRIBUTE**. If  
 6013     this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be  
 6014     checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return  
 6015     CKR\_KEY\_HANDLE\_INVALID.  
 6016     Return Values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL,  
 6017     CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
 6018     CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED,  
 6019     CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID,  
 6020     CKR\_KEY\_NOT\_WRAPPABLE, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_UNEXTRACTABLE,  
 6021     CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK,  
 6022     CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED,  
 6023     CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN,  
 6024     CKR\_WRAPPING\_KEY\_HANDLE\_INVALID, CKR\_WRAPPING\_KEY\_SIZE\_RANGE,  
 6025     CKR\_WRAPPING\_KEY\_TYPE\_INCONSISTENT.  
 6026     Example:  
 6027         CK\_SESSION\_HANDLE hSession;  
 6028         CK\_OBJECT\_HANDLE hWrappingKey, hKey;  
 6029         CK\_MECHANISM mechanism = {  
 6030             CKM\_DES3\_ECB, NULL\_PTR, 0  
 6031         };  
 6032         CK\_BYTE wrappedKey[8];  
 6033         CK ULONG ulWrappedKeyLen;  
 6034         CK\_RV rv;  
 6035         .  
 6036         .  
 6037         .  
 6038         ulWrappedKeyLen = sizeof(wrappedKey);  
 6039         rv = C\_WrapKey(  
 6040             hSession, &mechanism,  
 6041             hWrappingKey, hKey,

```
6042     wrappedKey, &ulWrappedKeyLen);  
6043     if (rv == CKR_OK) {  
6044         .  
6045         .  
6046     }
```

## 6047 5.18.4 C\_UnwrapKey

```
6048 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (  
6049     CK_SESSION_HANDLE hSession,  
6050     CK_MECHANISM_PTR pMechanism,  
6051     CK_OBJECT_HANDLE hUnwrappingKey,  
6052     CK_BYTE_PTR pWrappedKey,  
6053     CK ULONG ulWrappedKeyLen,  
6054     CK_ATTRIBUTE_PTR pTemplate,  
6055     CK ULONG ulAttributeCount,  
6056     CK_OBJECT_HANDLE_PTR phKey  
6057 );
```

6058 **C\_UnwrapKey** unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or secret key object.  
6059 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is  
6060 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the  
6061 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the  
6062 number of attributes in the template; *phKey* points to the location that receives the handle of the  
6063 recovered key.

6064 The **CKA\_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports  
6065 unwrapping, MUST be CK\_TRUE.

6066 The new key will have the **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, and the  
6067 **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE. The **CKA\_EXTRACTABLE** attribute is by  
6068 default set to CK\_TRUE.

6069 Some mechanisms may modify, or attempt to modify, the contents of the *pMechanism* structure at the  
6070 same time that the key is unwrapped.

6071 If a call to **C\_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without  
6072 creating any key object.

6073 The key object created by a successful call to **C\_UnwrapKey** will have its **CKA\_LOCAL** attribute set to  
6074 CK\_FALSE. In addition, the object created will have a value for **CKA\_UNIQUE\_ID** generated and  
6075 assigned (See Section 4.4.1).

6076 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute  
6077 **CKA\_UNWRAP\_TEMPLATE** can be used on the unwrapping key to specify an attribute set that will be  
6078 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied  
6079 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute  
6080 template and the size is the number of items in the template times the size of CK\_ATTRIBUTE. If this  
6081 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute  
6082 conflict occurs on an attempt to unwrap a key then the function SHALL return  
6083 CKR\_TEMPLATE\_INCONSISTENT.

6084 Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY,  
6085 CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID,  
6086 CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
6087 CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY,  
6088 CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_CANCELED,  
6089 CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY,  
6090 CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK,  
6091 CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED,  
6092 CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE,

```
6093 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,  
6094 CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE,  
6095 CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN,  
6096 CKR_WWRAPPED_KEY_INVALID, CKR_WWRAPPED_KEY_LEN_RANGE.
```

6097 Example:

```
6098 CK_SESSION_HANDLE hSession;  
6099 CK_OBJECT_HANDLE hUnwrappingKey, hKey;  
6100 CK_MECHANISM mechanism = {  
6101     CKM_DES3_ECB, NULL_PTR, 0  
6102 };  
6103 CK_BYTE wrappedKey[8] = {...};  
6104 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
6105 CK_KEY_TYPE keyType = CKK_DES;  
6106 CK_BBOOL true = CK_TRUE;  
6107 CK_ATTRIBUTE template[] = {  
6108     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
6109     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
6110     {CKA_ENCRYPT, &true, sizeof(true)},  
6111     {CKA_DECRYPT, &true, sizeof(true)}  
6112 };  
6113 CK_RV rv;  
6114 .  
6115 .  
6116 .  
6117 rv = C_UnwrapKey(  
6118     hSession, &mechanism, hUnwrappingKey,  
6119     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);  
6120 if (rv == CKR_OK) {  
6121     .  
6122     .  
6123 }
```

## 6124 5.18.5 C\_DeriveKey

```
6125 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey) (  
6126     CK_SESSION_HANDLE hSession,  
6127     CK_MECHANISM_PTR pMechanism,  
6128     CK_OBJECT_HANDLE hBaseKey,  
6129     CK_ATTRIBUTE_PTR pTemplate,  
6130     CK_ULONG ulAttributeCount,  
6131     CK_OBJECT_HANDLE_PTR phKey  
6132 );
```

6133 **C\_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's  
6134 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the  
6135 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number  
6136 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

6137 The values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and  
 6138 **CKA\_NEVER\_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold  
 6139 for the newly-derived key. See the description of each particular key-derivation mechanism in Section  
 6140 5.21.2 for any constraints of this type.  
 6141 If a call to **C\_DeriveKey** cannot support the precise template supplied to it, it will fail and return without  
 6142 creating any key object.  
 6143 The key object created by a successful call to **C\_DeriveKey** will have its **CKA\_LOCAL** attribute set to  
 6144 **CK\_FALSE**. In addition, the object created will have a value for **CKA\_UNIQUE\_ID** generated and  
 6145 assigned (See Section 4.4.1).  
 6146 To partition the derivation keys so they can only derive a subset of keys the attribute  
 6147 **CKA\_DERIVE\_TEMPLATE** can be used on the derivation keys to specify an attribute set that will be  
 6148 added to attributes of the key to be derived. If the attributes do not conflict with the user supplied attribute  
 6149 template, in 'pTemplate', then the derivation will proceed. The value of this attribute is an attribute  
 6150 template and the size is the number of items in the template times the size of **CK\_ATTRIBUTE**. If this  
 6151 attribute is not present on the base derivation keys then no additional attributes will be added. If any  
 6152 attribute conflict occurs on an attempt to derive a key then the function SHALL return  
 6153 **CKR\_TEMPLATE\_INCONSISTENT**.  
 6154 Return values: **CKR\_ARGUMENTS\_BAD**, **CKR\_ATTRIBUTE\_READ\_ONLY**,  
 6155 **CKR\_ATTRIBUTE\_TYPE\_INVALID**, **CKR\_ATTRIBUTE\_VALUE\_INVALID**,  
 6156 **CKR\_CRYPTOKI\_NOT\_INITIALIZED**, **CKR\_CURVE\_NOT\_SUPPORTED**, **CKR\_DEVICE\_ERROR**,  
 6157 **CKR\_DEVICE\_MEMORY**, **CKR\_DEVICE\_REMOVED**, **CKR\_DOMAIN\_PARAMS\_INVALID**,  
 6158 **CKR\_FUNCTION\_CANCELED**, **CKR\_FUNCTION\_FAILED**, **CKR\_GENERAL\_ERROR**,  
 6159 **CKR\_HOST\_MEMORY**, **CKR\_KEY\_HANDLE\_INVALID**, **CKR\_KEY\_SIZE\_RANGE**,  
 6160 **CKR\_KEY\_TYPE\_INCONSISTENT**, **CKR\_MECHANISM\_INVALID**,  
 6161 **CKR\_MECHANISM\_PARAM\_INVALID**, **CKR\_OK**, **CKR\_OPERATION\_ACTIVE**, **CKR\_PIN\_EXPIRED**,  
 6162 **CKR\_SESSION\_CLOSED**, **CKR\_SESSION\_HANDLE\_INVALID**, **CKR\_SESSION\_READ\_ONLY**,  
 6163 **CKR\_TEMPLATE\_INCOMPLETE**, **CKR\_TEMPLATE\_INCONSISTENT**,  
 6164 **CKR\_TOKEN\_WRITE\_PROTECTED**, **CKR\_USER\_NOT\_LOGGED\_IN**.

6165 Example:

```

6166 CK_SESSION_HANDLE hSession;
6167 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
6168 CK_MECHANISM keyPairMechanism = {
6169   CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
6170 };
6171 CK_BYTE prime[] = {...};
6172 CK_BYTE base[] = {...};
6173 CK_BYTE publicValue[128];
6174 CK_BYTE otherPublicValue[128];
6175 CK_MECHANISM mechanism = {
6176   CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)
6177 };
6178 CK_ATTRIBUTE template[] = {
6179   {CKA_VALUE, &publicValue, sizeof(publicValue) }
6180 };
6181 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
6182 CK_KEY_TYPE keyType = CKK_DES;
6183 CK_BBOOL true = CK_TRUE;
6184 CK_ATTRIBUTE publicKeyTemplate[] = {

```

```

6185     {CKA_PRIME, prime, sizeof(prime)},
6186     {CKA_BASE, base, sizeof(base)}
6187 };
6188 CK_ATTRIBUTE privateKeyTemplate[] = {
6189     {CKA_DERIVE, &true, sizeof(true)}
6190 };
6191 CK_ATTRIBUTE derivedKeyTemplate[] = {
6192     {CKA_CLASS, &keyClass, sizeof(keyClass)},
6193     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6194     {CKA_ENCRYPT, &true, sizeof(true)},
6195     {CKA_DECRYPT, &true, sizeof(true)}
6196 };
6197 CK_RV rv;
6198 .
6199 .
6200 .
6201 rv = C_GenerateKeyPair(
6202     hSession, &keyPairMechanism,
6203     publicKeyTemplate, 2,
6204     privateKeyTemplate, 1,
6205     &hPublicKey, &hPrivateKey);
6206 if (rv == CKR_OK) {
6207     rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
6208     if (rv == CKR_OK) {
6209         /* Put other guy's public value in otherPublicKey */
6210         .
6211         .
6212         rv = C_DeriveKey(
6213             hSession, &mechanism,
6214             hPrivateKey, derivedKeyTemplate, 4, &hKey);
6215         if (rv == CKR_OK) {
6216             .
6217             .
6218         }
6219     }
6220 }

```

## 6221 5.19 Random number generation functions

6222 Cryptoki provides the following functions for generating random numbers:

### 6223 5.19.1 C\_SeedRandom

```

6224 CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
6225     CK_SESSION_HANDLE hSession,
```

```

6226     CK_BYTE_PTR pSeed,
6227     CK_ULONG ulSeedLen
6228 );

```

6229 **C\_SeedRandom** mixes additional seed material into the token's random number generator. *hSession* is  
6230 the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed  
6231 material.  
6232 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
6233 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
6234 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
6235 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE,  
6236 CKR\_RANDOM\_SEED\_NOT\_SUPPORTED, CKR\_RANDOM\_NO\_RNG, CKR\_SESSION\_CLOSED,  
6237 CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.  
6238 Example: see **C\_GenerateRandom**.

## 6239 **5.19.2 C\_GenerateRandom**

```

6240 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (
6241     CK_SESSION_HANDLE hSession,
6242     CK_BYTE_PTR pRandomData,
6243     CK_ULONG ulRandomLen
6244 );

```

6245 **C\_GenerateRandom** generates random or pseudo-random data. *hSession* is the session's handle;  
6246 *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in  
6247 bytes of the random or pseudo-random data to be generated.  
6248 Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED,  
6249 CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED,  
6250 CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,  
6251 CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_RANDOM\_NO\_RNG,  
6252 CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

6253 Example:

```

6254 CK_SESSION_HANDLE hSession;
6255 CK_BYTE seed[] = {...};
6256 CK_BYTE randomData[] = {...};
6257 CK_RV rv;
6258
6259 .
6260 .
6261 rv = C_SeedRandom(hSession, seed, sizeof(seed));
6262 if (rv != CKR_OK) {
6263     .
6264     .
6265 }
6266 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
6267 if (rv == CKR_OK) {
6268     .
6269     .
6270 }

```

## 6271 5.20 Parallel function management functions

6272 Cryptoki provides the following functions for managing parallel execution of cryptographic functions.  
6273 These functions exist only for backwards compatibility.

### 6274 5.20.1 C\_GetFunctionStatus

```
6275 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus) (
6276     CK_SESSION_HANDLE hSession
6277 );
```

6278 In previous versions of Cryptoki, **C\_GetFunctionStatus** obtained the status of a function running in  
6279 parallel with an application. Now, however, **C\_GetFunctionStatus** is a legacy function which should  
6280 simply return the value CKR\_FUNCTION\_NOT\_PARALLEL.

6281 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED,  
6282 CKR\_FUNCTION\_NOT\_PARALLEL, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY,  
6283 CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_CLOSED.

### 6284 5.20.2 C\_CancelFunction

```
6285 CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction) (
6286     CK_SESSION_HANDLE hSession
6287 );
```

6288 In previous versions of Cryptoki, **C\_CancelFunction** cancelled a function running in parallel with an  
6289 application. Now, however, **C\_CancelFunction** is a legacy function which should simply return the value  
6290 CKR\_FUNCTION\_NOT\_PARALLEL.

6291 Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED,  
6292 CKR\_FUNCTION\_NOT\_PARALLEL, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY,  
6293 CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_CLOSED.

## 6294 5.21 Callback functions

6295 Cryptoki sessions can use function pointers of type **CK\_NOTIFY** to notify the application of certain  
6296 events.

### 6297 5.21.1 Surrender callbacks

6298 Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions;  
6299 decryption functions; message digesting functions; signing and MACing functions; functions for verifying  
6300 signatures and MACs; dual-purpose cryptographic functions; key management functions; random number  
6301 generation functions) executing in Cryptoki sessions can periodically surrender control to the application  
6302 who called them if the session they are executing in had a notification callback function associated with it  
6303 when it was opened. They do this by calling the session's callback with the arguments (hSession,  
6304 CKN\_SURRENDER, pApplication), where hSession is the session's handle and pApplication was  
6305 supplied to **C\_OpenSession** when the session was opened. Surrender callbacks should return either the  
6306 value CKR\_OK (to indicate that Cryptoki should continue executing the function) or the value  
6307 CKR\_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before  
6308 returning one of these values, the callback function can perform some computation, if desired.

6309 A typical use of a surrender callback might be to give an application user feedback during a lengthy key  
6310 pair generation operation. Each time the application receives a callback, it could display an additional “.”  
6311 to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the  
6312 key pair generation operation (probably by returning the value CKR\_CANCEL) if the user hit <ESCAPE>.

6313 A Cryptoki library is not *required* to make any surrender callbacks.

6314    **5.21.2 Vendor-defined callbacks**

6315    Library vendors can also define additional types of callbacks. Because of this extension capability,  
6316    application-supplied notification callback routines should examine each callback they receive, and if they  
6317    are unfamiliar with the type of that callback, they should immediately give control back to the library by  
6318    returning with the value CKR\_OK.

---

## 6 Mechanisms

### 6.1 RSA

Table 32, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_X9_31_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓ <sup>2</sup>	✓ <sup>2</sup>	✓			✓	
CKM_RSA_PKCS_OAEP	✓ <sup>2</sup>					✓	
CKM_RSA_PKCS_PSS		✓ <sup>2</sup>					
CKM_RSA_9796		✓ <sup>2</sup>	✓				
CKM_RSA_X_509	✓ <sup>2</sup>	✓ <sup>2</sup>	✓			✓	
CKM_RSA_X9_31		✓ <sup>2</sup>					
CKM_SHA1_RSA_PKCS		✓					
CKM_SHA224_RSA_PKCS		✓					
CKM_SHA256_RSA_PKCS		✓					
CKM_SHA384_RSA_PKCS		✓					
CKM_SHA512_RSA_PKCS		✓					
CKM_SHA1_RSA_PKCS_PSS		✓					
CKM_SHA224_RSA_PKCS_PSS		✓					
CKM_SHA256_RSA_PKCS_PSS		✓					
CKM_SHA384_RSA_PKCS_PSS		✓					
CKM_SHA512_RSA_PKCS_PSS		✓					
CKM_SHA1_RSA_X9_31		✓					
CKM_RSA_PKCS TPM_1_1	✓ <sup>2</sup>					✓	
CKM_RSA_PKCS_OAEP TPM_1_1	✓ <sup>2</sup>					✓	
CKM_SHA3_224_RSA_PKCS		✓					
CKM_SHA3_256_RSA_PKCS		✓					
CKM_SHA3_384_RSA_PKCS		✓					
CKM_SHA3_512_RSA_PKCS		✓					
CKM_SHA3_224_RSA_PKCS_PSS		✓					
CKM_SHA3_256_RSA_PKCS_PSS		✓					
CKM_SHA3_384_RSA_PKCS_PSS		✓					
CKM_SHA3_512_RSA_PKCS_PSS		✓					

#### 6.1.1 Definitions

This section defines the RSA key type “CKK\_RSA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of RSA key objects.

Mechanisms:

CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN

CKM\_RSA\_PKCS

CKM\_RSA\_9796

6329 CKM\_RSA\_X\_509  
 6330 CKM\_MD2\_RSA\_PKCS  
 6331 CKM\_MD5\_RSA\_PKCS  
 6332 CKM\_SHA1\_RSA\_PKCS  
 6333 CKM\_SHA224\_RSA\_PKCS  
 6334 CKM\_SHA256\_RSA\_PKCS  
 6335 CKM\_SHA384\_RSA\_PKCS  
 6336 CKM\_SHA512\_RSA\_PKCS  
 6337 CKM\_RIPEMD128\_RSA\_PKCS  
 6338 CKM\_RIPEMD160\_RSA\_PKCS  
 6339 CKM\_RSA\_PKCS\_OAEP  
 6340 CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN  
 6341 CKM\_RSA\_X9\_31  
 6342 CKM\_SHA1\_RSA\_X9\_31  
 6343 CKM\_RSA\_PKCS\_PSS  
 6344 CKM\_SHA1\_RSA\_PKCS\_PSS  
 6345 CKM\_SHA224\_RSA\_PKCS\_PSS  
 6346 CKM\_SHA256\_RSA\_PKCS\_PSS  
 6347 CKM\_SHA512\_RSA\_PKCS\_PSS  
 6348 CKM\_SHA384\_RSA\_PKCS\_PSS  
 6349 CKM\_RSA\_PKCS TPM\_1\_1  
 6350 CKM\_RSA\_PKCS\_OAEP TPM\_1\_1  
 6351 CKM\_RSA\_AES\_KEY\_WRAP  
 6352 CKM\_SHA3\_224\_RSA\_PKCS  
 6353 CKM\_SHA3\_256\_RSA\_PKCS  
 6354 CKM\_SHA3\_384\_RSA\_PKCS  
 6355 CKM\_SHA3\_512\_RSA\_PKCS  
 6356 CKM\_SHA3\_224\_RSA\_PKCS\_PSS  
 6357 CKM\_SHA3\_256\_RSA\_PKCS\_PSS  
 6358 CKM\_SHA3\_384\_RSA\_PKCS\_PSS  
 6359 CKM\_SHA3\_512\_RSA\_PKCS\_PSS  
 6360

## 6361 6.1.2 RSA public key objects

6362 RSA public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_RSA**) hold RSA public keys.  
 6363 The following table defines the RSA public key object attributes, in addition to the common attributes  
 6364 defined for this object class:

6365 *Table 33, RSA Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_MODULUS <sup>1,4</sup>	Big integer	Modulus $n$
CKA_MODULUS_BITS <sup>2,3</sup>	CK ULONG	Length in bits of modulus $n$
CKA_PUBLIC_EXPONENT <sup>1</sup>	Big integer	Public exponent $e$

6366 - Refer to Table 11 for footnotes

6367 Depending on the token, there may be limits on the length of key components. See PKCS #1 for more  
6368 information on RSA keys.

6369 The following is a sample template for creating an RSA public key object:

```
6370     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
6371     CK_KEY_TYPE keyType = CKK_RSA;
6372     CK_UTF8CHAR label[] = "An RSA public key object";
6373     CK_BYTE modulus[] = {...};
6374     CK_BYTE exponent[] = {...};
6375     CK_BBOOL true = CK_TRUE;
6376     CK_ATTRIBUTE template[] = {
6377         {CKA_CLASS, &class, sizeof(class)},
6378         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6379         {CKA_TOKEN, &true, sizeof(true)},
6380         {CKA_LABEL, label, sizeof(label)-1},
6381         {CKA_WRAP, &true, sizeof(true)},
6382         {CKA_ENCRYPT, &true, sizeof(true)},
6383         {CKA_MODULUS, modulus, sizeof(modulus)},
6384         {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}}
6385     };
```

### 6386 6.1.3 RSA private key objects

6387 RSA private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_RSA**) hold RSA private keys.  
6388 The following table defines the RSA private key object attributes, in addition to the common attributes  
6389 defined for this object class:

6390 *Table 34, RSA Private Key Object Attributes*

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

6391 <sup>1</sup>Refer to Table 11 for footnotes

6392 Depending on the token, there may be limits on the length of the key components. See PKCS #1 for  
6393 more information on RSA keys.

6394 Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above  
6395 attributes, which can assist in performing rapid RSA computations. Other tokens might store only the  
6396 **CKA\_MODULUS** and **CKA\_PRIVATE\_EXPONENT** values. Effective with version 2.40, tokens MUST  
6397 also store **CKA\_PUBLIC\_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the  
6398 associated public key.

6399 Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an  
6400 RSA private key, it stores whichever of the fields in Table 34 it keeps track of. Later, if an application  
6401 asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose  
6402 values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request  
6403 fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various  
6404 attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores

6405 values only for the **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, and **CKA\_PRIME\_2** attributes, then  
6406 Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed  
6407 efficiently from these three values). However, a Cryptoki implementation may or may not actually do this  
6408 extra computation. The only attributes from Table 34 for which a Cryptoki implementation is *required* to  
6409 be able to return values are **CKA\_MODULUS**, **CKA\_PUBLIC\_EXPONENT** and  
6410 **CKA\_PRIVATE\_EXPONENT**. A token SHOULD also be able to return **CKA\_PUBLIC\_KEY\_INFO** for an  
6411 RSA private key.

6412 If an RSA private key object is created on a token, and more attributes from Table 34 are supplied to the  
6413 object creation call than are supported by the token, the extra attributes are likely to be thrown away. If  
6414 an attempt is made to create an RSA private key object on a token with insufficient attributes for that  
6415 particular token, then the object creation call fails and returns CKR\_TEMPLATE\_INCOMPLETE.

6416 Note that when generating an RSA private key, there is no **CKA\_MODULUS\_BITS** attribute specified.  
6417 This is because RSA private keys are only generated as part of an RSA key pair, and the  
6418 **CKA\_MODULUS\_BITS** attribute for the pair is specified in the template for the RSA public key.

6419 The following is a sample template for creating an RSA private key object:

```
6420     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
6421     CK_KEY_TYPE keyType = CKK_RSA;
6422     CK_UTF8CHAR label[] = "An RSA private key object";
6423     CK_BYTE subject[] = {...};
6424     CK_BYTE id[] = {123};
6425     CK_BYTE modulus[] = {...};
6426     CK_BYTE publicExponent[] = {...};
6427     CK_BYTE privateExponent[] = {...};
6428     CK_BYTE prime1[] = {...};
6429     CK_BYTE prime2[] = {...};
6430     CK_BYTE exponent1[] = {...};
6431     CK_BYTE exponent2[] = {...};
6432     CK_BYTE coefficient[] = {...};
6433     CK_BBOOL true = CK_TRUE;
6434     CK_ATTRIBUTE template[] = {
6435         {CKA_CLASS, &class, sizeof(class)},
6436         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6437         {CKA_TOKEN, &true, sizeof(true)},
6438         {CKA_LABEL, label, sizeof(label)-1},
6439         {CKA_SUBJECT, subject, sizeof(subject)},
6440         {CKA_ID, id, sizeof(id)},
6441         {CKA_SENSITIVE, &true, sizeof(true)},
6442         {CKA_DECRYPT, &true, sizeof(true)},
6443         {CKA_SIGN, &true, sizeof(true)},
6444         {CKA_MODULUS, modulus, sizeof(modulus)},
6445         {CKA_PUBLIC_EXPONENT, publicExponent,
6446             sizeof(publicExponent)},
6447         {CKA_PRIVATE_EXPONENT, privateExponent,
6448             sizeof(privateExponent)},
6449         {CKA_PRIME_1, prime1, sizeof(prime1)},
6450         {CKA_PRIME_2, prime2, sizeof(prime2)},
6451         {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
6452         {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
6453         {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
```

6454 } ;

#### 6455 6.1.4 PKCS #1 RSA key pair generation

6456 The PKCS #1 RSA key pair generation mechanism, denoted **CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN**, is a  
6457 key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

6458 It does not have a parameter.

6459 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public  
6460 exponent, as specified in the **CKA\_MODULUS\_BITS** and **CKA\_PUBLIC\_EXPONENT** attributes of the  
6461 template for the public key. The **CKA\_PUBLIC\_EXPONENT** may be omitted in which case the  
6462 mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).  
6463 Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by  
6464 the token.

6465 Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error  
6466 if this attribute is omitted from the template. Experience has shown that many implementations of 2.11  
6467 and prior did allow the **CKA\_PUBLIC\_EXPONENT** attribute to be omitted from the template, and  
6468 behaved as described above. The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**,  
6469 **CKA\_MODULUS**, and **CKA\_PUBLIC\_EXPONENT** attributes to the new public key.  
6470 **CKA\_PUBLIC\_EXPONENT** will be copied from the template if supplied.

6471 **CKR\_TEMPLATE\_INCONSISTENT** shall be returned if the implementation cannot use the supplied  
6472 exponent value. It contributes the **CKA\_CLASS** and **CKA\_KEY\_TYPE** attributes to the new private key; it  
6473 may also contribute some of the following attributes to the new private key: **CKA\_MODULUS**,  
6474 **CKA\_PUBLIC\_EXPONENT**, **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, **CKA\_PRIME\_2**,  
6475 **CKA\_EXPONENT\_1**, **CKA\_EXPONENT\_2**, **CKA\_COEFFICIENT**. Other attributes supported by the  
6476 RSA public and private key types (specifically, the flags indicating which functions the keys support) may  
6477 also be specified in the templates for the keys, or else are assigned default initial values.

6478 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6479 specify the supported range of RSA modulus sizes, in bits.

#### 6480 6.1.5 X9.31 RSA key pair generation

6481 The X9.31 RSA key pair generation mechanism, denoted **CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN**, is a key  
6482 pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

6483 It does not have a parameter.

6484 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public  
6485 exponent, as specified in the **CKA\_MODULUS\_BITS** and **CKA\_PUBLIC\_EXPONENT** attributes of the  
6486 template for the public key.

6487 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_MODULUS**, and  
6488 **CKA\_PUBLIC\_EXPONENT** attributes to the new public key. It contributes the **CKA\_CLASS** and  
6489 **CKA\_KEY\_TYPE** attributes to the new private key; it may also contribute some of the following attributes  
6490 to the new private key: **CKA\_MODULUS**, **CKA\_PUBLIC\_EXPONENT**, **CKA\_PRIVATE\_EXPONENT**,  
6491 **CKA\_PRIME\_1**, **CKA\_PRIME\_2**, **CKA\_EXPONENT\_1**, **CKA\_EXPONENT\_2**, **CKA\_COEFFICIENT**.

6492 Other attributes supported by the RSA public and private key types (specifically, the flags indicating which  
6493 functions the keys support) may also be specified in the templates for the keys, or else are assigned  
6494 default initial values. Unlike the **CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN** mechanism, this mechanism is  
6495 guaranteed to generate *p* and *q* values, **CKA\_PRIME\_1** and **CKA\_PRIME\_2** respectively, that meet the  
6496 strong primes requirement of X9.31.

6497 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6498 specify the supported range of RSA modulus sizes, in bits.

#### 6499 6.1.6 PKCS #1 v1.5 RSA

6500 The PKCS #1 v1.5 RSA mechanism, denoted **CKM\_RSA\_PKCS**, is a multi-purpose mechanism based  
6501 on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports

6502 single-part encryption and decryption; single-part signatures and verification with and without message  
6503 recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1  
6504 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for  
6505 the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5 .

6506 This mechanism does not have a parameter.

6507 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token  
6508 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the  
6509 "input" to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped;  
6510 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the  
6511 key, except the key length; the application must convey these separately. In particular, the mechanism  
6512 contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes  
6513 to the recovered key during unwrapping; other attributes must be specified in the template.

6514 Constraints on key types and the length of the data are summarized in the following table. For  
6515 encryption, decryption, signatures and signature verification, the input and output data may begin at the  
6516 same location in memory. In the table,  $k$  is the length in bytes of the RSA modulus.

6517 *Table 35, PKCS #1 v1.5 RSA: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt <sup>1</sup>	RSA public key	$\leq k-11$	$k$	block type 02
C_Decrypt <sup>1</sup>	RSA private key	$k$	$\leq k-11$	block type 02
C_Sign <sup>1</sup>	RSA private key	$\leq k-11$	$k$	block type 01
C_SignRecover	RSA private key	$\leq k-11$	$k$	block type 01
C_Verify <sup>1</sup>	RSA public key	$\leq k-11, k^2$	N/A	block type 01
C_VerifyRecover	RSA public key	$k$	$\leq k-11$	block type 01
C_WrapKey	RSA public key	$\leq k-11$	$k$	block type 02
C_UnwrapKey	RSA private key	$k$	$\leq k-11$	block type 02

6518 <sup>1</sup> Single-part operations only.

6519 <sup>2</sup> Data length, signature length.

6520 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6521 specify the supported range of RSA modulus sizes, in bits.

## 6522 **6.1.7 PKCS #1 RSA OAEP mechanism parameters**

### 6523 ◆ **CK\_RSA\_PKCS\_MGF\_TYPE; CK\_RSA\_PKCS\_MGF\_TYPE\_PTR**

6524 **CK\_RSA\_PKCS\_MGF\_TYPE** is used to indicate the Mask Generation Function (MGF) applied to a  
6525 message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the  
6526 PKCS #1 PSS signature scheme. It is defined as follows:

6527 `typedef CK ULONG CK_RSA_PKCS_MGF_TYPE;`

6529 The following MGFs are defined in PKCS #1. The following table lists the defined functions.

6530 *Table 36, PKCS #1 Mask Generation Functions*

Source Identifier	Value
CKG_MGF1_SHA1	0x00000001UL
CKG_MGF1_SHA224	0x00000005UL
CKG_MGF1_SHA256	0x00000002UL
CKG_MGF1_SHA384	0x00000003UL
CKG_MGF1_SHA512	0x00000004UL
CKG_MGF1_SHA3_224	0x00000006UL
CKG_MGF1_SHA3_256	0x00000007UL
CKG_MGF1_SHA3_384	0x00000008UL
CKG_MGF1_SHA3_512	0x00000009UL

6531 **CK\_RSA\_PKCS\_MGF\_TYPE\_PTR** is a pointer to a **CK\_RSA\_PKCS\_MGF\_TYPE**.

6532 ◆ **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE;**  
 6533 **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE\_PTR**

6534 **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE** is used to indicate the source of the encoding parameter  
 6535 when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

6536   **typedef CK ULONG CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE;**

6537

6538 The following encoding parameter sources are defined in PKCS #1. The following table lists the defined  
 6539 sources along with the corresponding data type for the *pSourceData* field in the  
 6540 **CK\_RSA\_PKCS\_OAEP\_PARAMS** structure defined below.

6541 *Table 37, PKCS #1 RSA OAEP: Encoding parameter sources*

Source Identifier	Value	Data Type
CKZ_DATA_SPECIFIED	0x00000001UL	Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, <i>pSourceData</i> must be NULL and <i>ulSourceDataLen</i> must be zero.

6542 **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE\_PTR** is a pointer to a

6543 **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE**.

6544 ◆ **CK\_RSA\_PKCS\_OAEP\_PARAMS; CK\_RSA\_PKCS\_OAEP\_PARAMS\_PTR**

6545 **CK\_RSA\_PKCS\_OAEP\_PARAMS** is a structure that provides the parameters to the  
 6546 **CKM\_RSA\_PKCS\_OAEP** mechanism. The structure is defined as follows:

```
6547   typedef struct CK_RSA_PKCS_OAEP_PARAMS {
6548     CK_MECHANISM_TYPE                           hashAlg;
6549     CK_RSA_PKCS_MGF_TYPE                       mgf;
6550     CK_RSA_PKCS_OAEP_SOURCE_TYPE           source;
6551     CK_VOID_PTR                               pSourceData;
6552     CK ULONG                                  ulSourceDataLen;
6553 } CK_RSA_PKCS_OAEP_PARAMS;
```

6554

6555 The fields of the structure have the following meanings:

hashAlg	mechanism ID of the message digest algorithm used to calculate the digest of the encoding parameter
mgf	mask generation function to use on the encoded block

6559 source source of the encoding parameter  
6560 pSourceData data used as the input for the encoding parameter source  
6561 ulSourceDataLen length of the encoding parameter source input  
6562 CK\_RSA\_PKCS\_OAEP\_PARAMS\_PTR is a pointer to a CK\_RSA\_PKCS\_OAEP\_PARAMS.  
6563

## 6.1.8 PKCS #1 RSA OAEP

6565 The PKCS #1 RSA OAEP mechanism, denoted **CKM\_RSA\_PKCS\_OAEP**, is a multi-purpose  
6566 mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1.  
6567 It supports single-part encryption and decryption; key wrapping; and key unwrapping.

6568 It has a parameter, a **CK\_RSA\_PKCS\_OAEP\_PARAMS** structure.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the "input" to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

6576 Constraints on key types and the length of the data are summarized in the following table. For encryption  
6577 and decryption, the input and output data may begin at the same location in memory. In the table,  $k$  is the  
6578 length in bytes of the RSA modulus, and  $hLen$  is the output length of the message digest algorithm  
6579 specified by the **hashAlg** field of the **CK RSA PKCS OAEP PARAMS** structure.

6580 Table 38, PKCS #1 RSA OAEP: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt <sup>1</sup>	RSA public key	$\leq k-2-2hLen$	$k$
C_Decrypt <sup>1</sup>	RSA private key	$k$	$\leq k-2-2hLen$
C_WrapKey	RSA public key	$\leq k-2-2hLen$	$k$
C_UnwrapKey	RSA private key	$k$	$\leq k-2-2hLen$

---

**6581**      1 Single-part operations only.

6582 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6583 specify the supported range of RSA modulus sizes, in bits.

6584 6.1.9 PKCS #1 RSA PSS mechanism parameters

6585 ◆ CK RSA PKCS PSS PARAMS; CK RSA PKCS PSS PARAMS PTR

6586 CK\_RSA\_PKCS\_PSS\_PARAMS is a structure that provides the parameters to the  
6587 CKM RSA PKCS PSS mechanism. The structure is defined as follows:

```
6588     typedef struct CK_RSA_PKCS_PSS_PARAMS {  
6589         CK_MECHANISM_TYPE      hashAlg;  
6590         CK_RSA_PKCS_MGF_TYPE   mgf;  
6591         CK_ULONG                sLen;  
6592     } CK_RSA_PKCS_PSS_PARAMS;
```

6594 The fields of the structure have the following meanings:

6595 hashAlg hash algorithm used in the PSS encoding; if the signature  
 6596 mechanism does not include message hashing, then this value must  
 6597 be the mechanism used by the application to generate the message  
 6598 hash; if the signature mechanism includes hashing, then this value  
 6599 must match the hash algorithm indicated by the signature  
 6600 mechanism  
 6601 mgf mask generation function to use on the encoded block  
 6602 sLen length, in bytes, of the salt value used in the PSS encoding; typical  
 6603 values are the length of the message hash and zero

6604 **CK\_RSA\_PKCS\_PSS\_PARAMS\_PTR** is a pointer to a **CK\_RSA\_PKCS\_PSS\_PARAMS**.

## 6.1.10 PKCS #1 RSA PSS

6606 The PKCS #1 RSA PSS mechanism, denoted **CKM\_RSA\_PKCS\_PSS**, is a mechanism based on the  
 6607 RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part  
 6608 signature generation and verification without message recovery. This mechanism corresponds only to the  
 6609 part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash  
 6610 value on the message to be signed.

6611 It has a parameter, a **CK\_RSA\_PKCS\_PSS\_PARAMS** structure. The *sLen* field must be less than or  
 6612 equal to  $k^* - 2 \cdot hLen$  and *hLen* is the length of the input to the C\_Sign or C\_Verify function.  $k^*$  is the length  
 6613 in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple  
 6614 of 8, in which case  $k^*$  is one less than the length in bytes of the RSA modulus.

6615 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*  
 6616 is the length in bytes of the RSA.

6617 *Table 39, PKCS #1 RSA PSS: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	RSA private key	<i>hLen</i>	<i>k</i>
C_Verify <sup>1</sup>	RSA public key	<i>hLen, k</i>	N/A

6618 <sup>1</sup> Single-part operations only.

6619 <sup>2</sup> Data length, signature length.

6620 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 6621 specify the supported range of RSA modulus sizes, in bits.

## 6.1.11 ISO/IEC 9796 RSA

6623 The ISO/IEC 9796 RSA mechanism, denoted **CKM\_RSA\_9796**, is a mechanism for single-part  
 6624 signatures and verification with and without message recovery based on the RSA public-key  
 6625 cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

6626 This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly,  
 6627 the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

6636 This mechanism does not have a parameter.

6637 Constraints on key types and the length of input and output data are summarized in the following table.  
6638 In the table,  $k$  is the length in bytes of the RSA modulus.

6639 *Table 40, ISO/IEC 9796 RSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	RSA private key	$\leq \lfloor k/2 \rfloor$	$k$
C_SignRecover	RSA private key	$\leq \lfloor k/2 \rfloor$	$k$
C_Verify <sup>1</sup>	RSA public key	$\leq \lfloor k/2 \rfloor, k^2$	N/A
C_VerifyRecover	RSA public key	$k$	$\leq \lfloor k/2 \rfloor$

6640 <sup>1</sup> Single-part operations only.

6641 <sup>2</sup> Data length, signature length.

6642 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6643 specify the supported range of RSA modulus sizes, in bits.

### 6644 6.1.12 X.509 (raw) RSA

6645 The X.509 (raw) RSA mechanism, denoted **CKM\_RSA\_X\_509**, is a multi-purpose mechanism based on  
6646 the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures  
6647 and verification with and without message recovery; key wrapping; and key unwrapping. All these  
6648 operations are based on so-called “raw” RSA, as assumed in X.509.

6649 “Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first,  
6650 applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first.  
6651 The input string, considered as an integer, must be less than the modulus; the output string is also less  
6652 than the modulus.

6653 This mechanism does not have a parameter.

6654 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token  
6655 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the  
6656 “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped;  
6657 similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other  
6658 information about the key; the application must convey these separately, and supply them when  
6659 unwrapping the key.

6660 Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism,  
6661 padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the  
6662 sequence of plaintext bytes  $b_1 b_2 \dots b_n$  ( $n \leq k$ ), Cryptoki forms  $P=2^{n-1}b_1+2^{n-2}b_2+\dots+b_n$ . This number must  
6663 be less than the RSA modulus. The  $k$ -byte ciphertext ( $k$  is the length in bytes of the RSA modulus) is  
6664 produced by raising  $P$  to the RSA public exponent modulo the RSA modulus. Decryption of a  $k$ -byte  
6665 ciphertext  $C$  is accomplished by raising  $C$  to the RSA private exponent modulo the RSA modulus, and  
6666 returning the resulting value as a sequence of exactly  $k$  bytes. If the resulting plaintext is to be used to  
6667 produce an unwrapped key, then however many bytes are specified in the template for the length of the  
6668 key are taken from the end of this sequence of bytes.

6669 Technically, the above procedures may differ very slightly from certain details of what is specified in  
6670 X.509.

6671 Executing cryptographic operations using this mechanism can result in the error returns  
6672 **CKR\_DATA\_INVALID** (if plaintext is supplied which has the same length as the RSA modulus and is  
6673 numerically at least as large as the modulus) and **CKR\_ENCRYPTED\_DATA\_INVALID** (if ciphertext is  
6674 supplied which has the same length as the RSA modulus and is numerically at least as large as the  
6675 modulus).

6676 Constraints on key types and the length of input and output data are summarized in the following table.  
6677 In the table,  $k$  is the length in bytes of the RSA modulus.

6678 *Table 41, X.509 (Raw) RSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt <sup>1</sup>	RSA public key	$\leq k$	$k$
C_Decrypt <sup>1</sup>	RSA private key	$k$	$k$
C_Sign <sup>1</sup>	RSA private key	$\leq k$	$k$
C_SignRecover	RSA private key	$\leq k$	$k$
C_Verify <sup>1</sup>	RSA public key	$\leq k, k^2$	N/A
C_VerifyRecover	RSA public key	$k$	$k$
C_WrapKey	RSA public key	$\leq k$	$k$
C_UnwrapKey	RSA private key	$k$	$\leq k$ (specified in template)

6679 1 Single-part operations only.

6680 2 Data length, signature length.

6681 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6682 specify the supported range of RSA modulus sizes, in bits.

6683 This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC  
6684 9796 block formats.

### 6685 6.1.13 ANSI X9.31 RSA

6686 The ANSI X9.31 RSA mechanism, denoted **CKM\_RSA\_X9\_31**, is a mechanism for single-part signatures  
6687 and verification without message recovery based on the RSA public-key cryptosystem and the block  
6688 formats defined in ANSI X9.31.

6689 This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must  
6690 be applied by the application.

6691 This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly,  
6692 the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

6701 This mechanism does not have a parameter.

6702 Constraints on key types and the length of input and output data are summarized in the following table.  
6703 In the table,  $k$  is the length in bytes of the RSA modulus. For all operations, the  $k$  value must be at least  
6704 128 and a multiple of 32 as specified in ANSI X9.31.

6705 Table 42, ANSI X9.31 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	RSA private key	$\leq k-2$	$k$
C_Verify <sup>1</sup>	RSA public key	$\leq k-2, k^2$	N/A

6706 1 Single-part operations only.

6707 2 Data length, signature length.

6708 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6709 specify the supported range of RSA modulus sizes, in bits.

## 6.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160

The PKCS #1 v1.5 RSA signature with MD2 mechanism, denoted **CKM\_MD2\_RSA\_PKCS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described initially in PKCS #1 v1.5 with the object identifier `md2WithRSAEncryption`, and as in the scheme RSASSA-PKCS1-v1\_5 in the current version of PKCS #1, where the underlying hash function is MD2.

Similarly, the PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted **CKM\_MD5\_RSA\_PKCS**, performs the same operations described in PKCS #1 with the object identifier `md5WithRSAEncryption`. The PKCS #1 v1.5 RSA signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_PKCS**, performs the same operations, except that it uses the hash function SHA-1 with object identifier `sha1WithRSAEncryption`.

Likewise, the PKCS #1 v1.5 RSA signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM\_SHA256\_RSA\_PKCS**, **CKM\_SHA384\_RSA\_PKCS**, and **CKM\_SHA512\_RSA\_PKCS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions with the object identifiers `sha256WithRSAEncryption`, `sha384WithRSAEncryption` and `sha512WithRSAEncryption` respectively.

The PKCS #1 v1.5 RSA signature with RIPEMD-128 or RIPEMD-160, denoted **CKM\_RIPEMD128\_RSA\_PKCS** and **CKM\_RIPEMD160\_RSA\_PKCS** respectively, perform the same operations using the RIPE-MD 128 and RIPE-MD 160 hash functions.

None of these mechanisms has a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table,  $k$  is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms,  $k$  must be at least 27; for the PKCS #1 v1.5 RSA signature with SHA-1 mechanism,  $k$  must be at least 31, and so on for other underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

Table 43, PKCS #1 v1.5 RSA Signatures with Various Hash Functions: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Sign	RSA private key	any	$k$	block type 01
C_Verify	RSA public key	any, $k^2$	N/A	block type 01

2 Data length, signature length.

For these mechanisms, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

## 6.1.15 PKCS #1 v1.5 RSA signature with SHA-224

The PKCS #1 v1.5 RSA signature with SHA-224 mechanism, denoted **CKM\_SHA224\_RSA\_PKCS**, performs similarly as the other **CKM\_SHAX\_RSA\_PKCS** mechanisms but uses the SHA-224 hash function.

## 6.1.16 PKCS #1 RSA PSS signature with SHA-224

The PKCS #1 RSA PSS signature with SHA-224 mechanism, denoted **CKM\_SHA224\_RSA\_PKCS\_PSS**, performs similarly as the other **CKM\_SHAX\_RSA\_PKCS\_PSS** mechanisms but uses the SHA-224 hash function.

## 6.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_PKCS\_PSS**, performs single- and multiple-part digital signatures and verification operations without message

6752 recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-  
6753 PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

6754 The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted  
6755 **CKM\_SHA256\_RSA\_PKCS\_PSS**, **CKM\_SHA384\_RSA\_PKCS\_PSS**, and  
6756 **CKM\_SHA512\_RSA\_PKCS\_PSS** respectively, perform the same operations using the SHA-256, SHA-  
6757 384 and SHA-512 hash functions.

6758 The mechanisms have a parameter, a **CK\_RSA\_PKCS\_PSS\_PARAMS** structure. The *sLen* field must  
6759 be less than or equal to  $k^* - hLen$  where *hLen* is the length in bytes of the hash value.  $k^*$  is the length in  
6760 bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of  
6761 8, in which case  $k^*$  is one less than the length in bytes of the RSA modulus.

6762 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*  
6763 is the length in bytes of the RSA modulus.

6764 *Table 44, PKCS #1 RSA PSS Signatures with Various Hash Functions: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	<i>k</i>
C_Verify	RSA public key	any, $k^2$	N/A

6765 2 Data length, signature length.

6766 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6767 specify the supported range of RSA modulus sizes, in bits.

### 6768 **6.1.18 PKCS #1 v1.5 RSA signature with SHA3**

6769 The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms,  
6770 denoted **CKM\_SHA3\_224\_RSA\_PKCS**, **CKM\_SHA3\_256\_RSA\_PKCS**, **CKM\_SHA3\_384\_RSA\_PKCS**,  
6771 and **CKM\_SHA3\_512\_RSA\_PKCS** respectively, performs similarly as the other  
6772 **CKM\_SHAX\_RSA\_PKCS** mechanisms but uses the corresponding SHA3 hash functions.

### 6773 **6.1.19 PKCS #1 RSA PSS signature with SHA3**

6774 The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms,  
6775 denoted **CKM\_SHA3\_224\_RSA\_PKCS\_PSS**, **CKM\_SHA3\_256\_RSA\_PKCS\_PSS**,  
6776 **CKM\_SHA3\_384\_RSA\_PKCS\_PSS**, and **CKM\_SHA3\_512\_RSA\_PKCS\_PSS** respectively, performs  
6777 similarly as the other **CKM\_SHAX\_RSA\_PKCS\_PSS** mechanisms but uses the corresponding SHA-3  
6778 hash functions.

### 6779 **6.1.20 ANSI X9.31 RSA signature with SHA-1**

6780 The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_X9\_31**, performs  
6781 single- and multiple-part digital signatures and verification operations without message recovery. The  
6782 operations performed are as described in ANSI X9.31.

6783 This mechanism does not have a parameter.

6784 Constraints on key types and the length of the data for these mechanisms are summarized in the  
6785 following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value  
6786 must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

6787 *Table 45, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	<i>k</i>
C_Verify	RSA public key	any, $k^2$	N/A

6788 2 Data length, signature length.

6789 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO**  
6790 structure specify the supported range of RSA modulus sizes, in bits.

### 6791 **6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA**

6792 The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted **CKM\_RSA\_PKCS TPM\_1\_1**, is a  
6793 multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in  
6794 PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b.  
6795 Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports  
6796 single-part encryption and decryption; key wrapping; and key unwrapping.

6797 This mechanism does not have a parameter. It differs from the standard PKCS#1 v1.5 RSA encryption  
6798 mechanism in that the plaintext is wrapped in a **TCPA\_BOUND\_DATA** (**TPM\_BOUND\_DATA** for TPM  
6799 1.2) structure before being submitted to the PKCS#1 v1.5 encryption process. On encryption, the version  
6800 field of the **TCPA\_BOUND\_DATA** (**TPM\_BOUND\_DATA** for TPM 1.2) structure must contain 0x01, 0x01,  
6801 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

6802 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token  
6803 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the  
6804 “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped;  
6805 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the  
6806 key, except the key length; the application must convey these separately. In particular, the mechanism  
6807 contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes  
6808 to the recovered key during unwrapping; other attributes must be specified in the template.

6809 Constraints on key types and the length of the data are summarized in the following table. For encryption  
6810 and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the  
6811 length in bytes of the RSA modulus.

6812 *Table 46, TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt <sup>1</sup>	RSA public key	$\leq k-11-5$	<i>k</i>
C_Decrypt <sup>1</sup>	RSA private key	<i>k</i>	$\leq k-11-5$
C_WrapKey	RSA public key	$\leq k-11-5$	<i>k</i>
C_UnwrapKey	RSA private key	<i>k</i>	$\leq k-11-5$

6813 <sup>1</sup> Single-part operations only.

6814

6815 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6816 specify the supported range of RSA modulus sizes, in bits.

### 6817 **6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP**

6818 The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted  
6819 **CKM\_RSA\_PKCS\_OAEP TPM\_1\_1**, is a multi-purpose mechanism based on the RSA public-key  
6820 cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA  
6821 TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM  
6822 Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key  
6823 unwrapping.

6824 This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption  
6825 mechanism in that the plaintext is wrapped in a **TCPA\_BOUND\_DATA** (**TPM\_BOUND\_DATA** for TPM  
6826 1.2) structure before being submitted to the encryption process and that all of the values of the  
6827 parameters that are passed to a standard **CKM\_RSA\_PKCS\_OAEP** operation are fixed. On encryption,  
6828 the version field of the **TCPA\_BOUND\_DATA** (**TPM\_BOUND\_DATA** for TPM 1.2) structure must contain  
6829 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be  
6830 accepted.

6831 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token  
6832 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the  
6833 “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped;  
6834 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the  
6835 key, except the key length; the application must convey these separately. In particular, the mechanism  
6836 contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes  
6837 to the recovered key during unwrapping; other attributes must be specified in the template.

6838 Constraints on key types and the length of the data are summarized in the following table. For encryption  
6839 and decryption, the input and output data may begin at the same location in memory. In the table,  $k$  is the  
6840 length in bytes of the RSA modulus.

6841 *Table 47, TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt <sup>1</sup>	RSA public key	$\leq k-2-40-5$	$k$
C_Decrypt <sup>1</sup>	RSA private key	$k$	$\leq k-2-40-5$
C_WrapKey	RSA public key	$\leq k-2-40-5$	$k$
C_UnwrapKey	RSA private key	$k$	$\leq k-2-40-5$

6842 <sup>1</sup> Single-part operations only.

6843 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
6844 specify the supported range of RSA modulus sizes, in bits.

## 6845 6.1.23 RSA AES KEY WRAP

6846 The RSA AES key wrap mechanism, denoted **CKM\_RSA\_AES\_KEY\_WRAP**, is a mechanism based on  
6847 the RSA public-key cryptosystem and the AES key wrap mechanism. It supports single-part key  
6848 wrapping; and key unwrapping.

6849 It has a parameter, a **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS** structure.

6850 The mechanism can wrap and unwrap a target asymmetric key of any length and type using an RSA  
6851 key.

- 6852 - A temporary AES key is used for wrapping the target key using  
6853      **CKM\_AES\_KEY\_WRAP\_KWP** mechanism.
- 6854 - The temporary AES key is wrapped with the wrapping RSA key using  
6855      **CKM\_RSA\_PKCS\_OAEP** mechanism.

6856 6857 For wrapping, the mechanism -

- 6858 • Generates a temporary random AES key of *ulAESKeyBits* length. This key is not accessible to the  
6859 user - no handle is returned.
- 6860 • Wraps the AES key with the wrapping RSA key using **CKM\_RSA\_PKCS\_OAEP** with parameters  
6861      of *OAEPPParams*.
- 6862 • Wraps the target key with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP**.
- 6863 • Zeroizes the temporary AES key
- 6864 • Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES  
6865      key, and the second is the wrapped target key.

6866 6867 The private target key will be encoded as defined in section 6.7.

6868 6869 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the  
6870 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown  
6871

- 6872 For unwrapping, the mechanism -
- 6873     • Splits the input into two parts. The first is the wrapped AES key, and the second is the wrapped target key. The length of the first part is equal to the length of the unwrapping RSA key.
- 6875     • Un-wraps the temporary AES key from the first part with the private RSA key using **CKM\_RSA\_PKCS\_OAEP** with parameters of *OAEPPParams*.
- 6877     • Un-wraps the target key from the second part with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP**.
- 6879     • Zeroizes the temporary AES key.
- 6880     • Returns the handle to the newly unwrapped target key.

6881 *Table 48, CKM\_RSA\_AES\_KEY\_WRAP Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_AES_KEY_WRAP						✓	

<sup>1</sup>SR = SignRecover, VR = VerifyRecover

## 6882 6.1.24 RSA AES KEY WRAP mechanism parameters

- 6883     ♦ **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS; CK\_RSA\_AES\_KEY\_WRAP\_PARAMS\_PTR**

6884 **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the  
6885 **CKM\_RSA\_AES\_KEY\_WRAP** mechanism. It is defined as follows:

```
6886     typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
6887         CK ULONG                     ulAESKeyBits;
6888         CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPPParams;
6889     } CK_RSA_AES_KEY_WRAP_PARAMS;
```

6890

6891 The fields of the structure have the following meanings:

ulAESKeyBits	length of the temporary AES key in bits. Can be only 128, 192 or 256.
--------------	-----------------------------------------------------------------------

pOAEPPParams	pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters.
--------------	---------------------------------------------------------------------------------------------------------------------------------

6897 **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS\_PTR** is a pointer to a **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS**.

## 6898 6.1.25 FIPS 186-4

6899 When **CKM\_RSA\_PKCS** is operated in FIPS mode, the length of the modulus SHALL only be 1024,  
6900 2048, or 3072 bits.

## 6901 6.2 DSA

6902 *Table 49, DSA Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA_PARAMETER_GEN					✓		
CKM_DSA_PROBABILISTIC_PARAMETER_GEN					✓		
CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN					✓		
CKM_DSA_FIPS_G_GEN					✓		
CKM_DSA		✓ <sup>2</sup>					
CKM_DSA_SHA1		✓					
CKM_DSA_SHA224		✓					
CKM_DSA_SHA256		✓					
CKM_DSA_SHA384		✓					
CKM_DSA_SHA512		✓					
CKM_DSA_SHA3_224		✓					
CKM_DSA_SHA3_256		✓					
CKM_DSA_SHA3_384		✓					
CKM_DSA_SHA3_512		✓					

## 6903    6.2.1 Definitions

6904    This section defines the key type “CKK\_DSA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE  
 6905    attribute of DSA key objects.

6906    Mechanisms:

6907       CKM\_DSA\_KEY\_PAIR\_GEN  
 6908       CKM\_DSA  
 6909       CKM\_DSA\_SHA1  
 6910       CKM\_DSA\_SHA224  
 6911       CKM\_DSA\_SHA256  
 6912       CKM\_DSA\_SHA384  
 6913       CKM\_DSA\_SHA512  
 6914       CKM\_DSA\_SHA3\_224  
 6915       CKM\_DSA\_SHA3\_256  
 6916       CKM\_DSA\_SHA3\_384  
 6917       CKM\_DSA\_SHA3\_512  
 6918       CKM\_DSA\_PARAMETER\_GEN  
 6919       CKM\_DSA\_PROBABILISTIC\_PARAMETER\_GEN  
 6920       CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN  
 6921       CKM\_DSA\_FIPS\_G\_GEN  
 6922

6923 ◆ **CK\_DSA\_PARAMETER\_GEN\_PARAM**

6924 CK\_DSA\_PARAMETER\_GEN\_PARAM is a structure which provides and returns parameters for the  
6925 NIST FIPS 186-4 parameter generating algorithms.

6926 CK\_DSA\_PARAMETER\_GEN\_PARAM\_PTR is a pointer to a CK\_DSA\_PARAMETER\_GEN\_PARAM.

6927

```
6928     typedef struct CK_DSA_PARAMETER_GEN_PARAM {  
6929         CK_MECHANISM_TYPE    hash;  
6930         CK_BYTE_PTR          pSeed;  
6931         CK ULONG              ulSeedLen;  
6932         CK ULONG              ulIndex;  
6933     } CK_DSA_PARAMETER_GEN_PARAM;
```

6934

6935 The fields of the structure have the following meanings:

6936 hash Mechanism value for the base hash used in PQG generation, Valid  
6937 values are CKM\_SHA\_1, CKM\_SHA224, CKM\_SHA256,  
6938 CKM\_SHA384, CKM\_SHA512.

6939 pSeed Seed value used to generate PQ and G. This value is returned by  
6940 CKM\_DSA\_PROBABILISTIC\_PARAMETER\_GEN,  
6941 CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN, and passed  
6942 into CKM\_DSA\_FIPS\_G\_GEN.

6943 ulSeedLen Length of seed value.

6944 ullIndex Index value for generating G. Input for CKM\_DSA\_FIPS\_G\_GEN.  
6945 Ignored by CKM\_DSA\_PROBABILISTIC\_PARAMETER\_GEN and  
6946 CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN.

6947 **6.2.2 DSA public key objects**

6948 DSA public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_DSA**) hold DSA public keys.  
6949 The following table defines the DSA public key object attributes, in addition to the common attributes  
6950 defined for this object class:

6951 *Table 50, DSA Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,3</sup>	Big integer	Prime $p$ (512 to 3072 bits, in steps of 64 bits)
CKA_SUBPRIME <sup>1,3</sup>	Big integer	Subprime $q$ (160, 224 bits, or 256 bits)
CKA_BASE <sup>1,3</sup>	Big integer	Base $g$
CKA_VALUE <sup>1,4</sup>	Big integer	Public value $y$

6952 <sup>1</sup> Refer to Table 11 for footnotes

6953 The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain  
6954 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

6955 The following is a sample template for creating a DSA public key object:

```
6956     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
6957     CK_KEY_TYPE keyType = CKK_DSA;  
6958     CK_UTF8CHAR label[] = "A DSA public key object";  
6959     CK_BYTE prime[] = {...};  
6960     CK_BYTE subprime[] = {...};  
6961     CK_BYTE base[] = {...};
```

```

6962     CK_BYTE value[] = { ... };
6963     CK_BBOOL true = CK_TRUE;
6964     CK_ATTRIBUTE template[] = {
6965         {CKA_CLASS, &class, sizeof(class)},
6966         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6967         {CKA_TOKEN, &true, sizeof(true)},
6968         {CKA_LABEL, label, sizeof(label)-1},
6969         {CKA_PRIME, prime, sizeof(prime)},
6970         {CKA_SUBPRIME, subprime, sizeof(subprime)},
6971         {CKA_BASE, base, sizeof(base)},
6972         {CKA_VALUE, value, sizeof(value)}}
6973     };
6974

```

### 6.2.3 DSA Key Restrictions

6975 FIPS PUB 186-4 specifies permitted combinations of prime and sub-prime lengths. They are:

- 6977 • Prime: 1024 bits, Subprime: 160
- 6978 • Prime: 2048 bits, Subprime: 224
- 6979 • Prime: 2048 bits, Subprime: 256
- 6980 • Prime: 3072 bits, Subprime: 256

6981 Earlier versions of FIPS 186 permitted smaller prime lengths, and those are included here for backwards  
6982 compatibility. An implementation that is compliant to FIPS 186-4 does not permit the use of primes of  
6983 any length less than 1024 bits.

### 6.2.4 DSA private key objects

6985 DSA private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_DSA**) hold DSA private keys.  
6986 The following table defines the DSA private key object attributes, in addition to the common attributes  
6987 defined for this object class:

6988 *Table 51, DSA Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4,6</sup>	Big integer	Prime $p$ (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME <sup>1,4,6</sup>	Big integer	Subprime $q$ (160 bits, 224 bits, or 256 bits)
CKA_BASE <sup>1,4,6</sup>	Big integer	Base $g$
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	Private value $x$

6989 <sup>1</sup> Refer to Table 11 for footnotes

6990 The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain  
6991 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

6992 Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key’s  
6993 template. This is because DSA private keys are only generated as part of a DSA key pair, and the DSA  
6994 domain parameters for the pair are specified in the template for the DSA public key.

6995 The following is a sample template for creating a DSA private key object:

```

6996     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
6997     CK_KEY_TYPE keyType = CKK_DSA;
6998     CK_UTF8CHAR label[] = "A DSA private key object";
6999     CK_BYTE subject[] = { ... };
7000     CK_BYTE id[] = {123};

```

```

7001 CK_BYTE prime[] = {...};
7002 CK_BYTE subprime[] = {...};
7003 CK_BYTE base[] = {...};
7004 CK_BYTE value[] = {...};
7005 CK_BBOOL true = CK_TRUE;
7006 CK_ATTRIBUTE template[] = {
7007     {CKA_CLASS, &class, sizeof(class)},
7008     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7009     {CKA_TOKEN, &true, sizeof(true)},
7010     {CKA_LABEL, label, sizeof(label)-1},
7011     {CKA SUBJECT, subject, sizeof(subject)},
7012     {CKA_ID, id, sizeof(id)},
7013     {CKA_SENSITIVE, &true, sizeof(true)},
7014     {CKA_SIGN, &true, sizeof(true)},
7015     {CKA_PRIME, prime, sizeof(prime)},
7016     {CKA_SUBPRIME, subprime, sizeof(subprime)},
7017     {CKA_BASE, base, sizeof(base)},
7018     {CKA_VALUE, value, sizeof(value)}
7019 };

```

## 7020 6.2.5 DSA domain parameter objects

7021 DSA domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type **CKK\_DSA**) hold  
7022 DSA domain parameters. The following table defines the DSA domain parameter object attributes, in  
7023 addition to the common attributes defined for this object class:

7024 *Table 52, DSA Domain Parameter Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4</sup>	Big integer	Prime $p$ (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME <sup>1,4</sup>	Big integer	Subprime $q$ (160 bits, 224 bits, or 256 bits)
CKA_BASE <sup>1,4</sup>	Big integer	Base $g$
CKA_PRIME_BITS <sup>2,3</sup>	CK ULONG	Length of the prime value.

7025 Refer to Table 11 for footnotes

7026 The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain  
7027 parameters”. See FIPS PUB 186-4 for more information on DSA domain parameters.

7028 To ensure backwards compatibility, if **CKA\_SUBPRIME\_BITS** is not specified for a call to  
7029 **C\_GenerateKey**, it takes on a default based on the value of **CKA\_PRIME\_BITS** as follows:

- If **CKA\_PRIME\_BITS** is less than or equal to 1024 then **CKA\_SUBPRIME\_BITS** shall be 160 bits
- If **CKA\_PRIME\_BITS** equals 2048 then **CKA\_SUBPRIME\_BITS** shall be 224 bits
- If **CKA\_PRIME\_BITS** equals 3072 then **CKA\_SUBPRIME\_BITS** shall be 256 bits

7033 The following is a sample template for creating a DSA domain parameter object:

```

7035 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7036 CK_KEY_TYPE keyType = CKK_DSA;
7037 CK_UTF8CHAR label[] = "A DSA domain parameter object";
7038 CK_BYTE prime[] = {...};
7039 CK_BYTE subprime[] = {...};
7040 CK_BYTE base[] = {...};

```

```

7041     CK_BBOOL true = CK_TRUE;
7042     CK_ATTRIBUTE template[] = {
7043         {CKA_CLASS, &class, sizeof(class)},
7044         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7045         {CKA_TOKEN, &true, sizeof(true)},
7046         {CKA_LABEL, label, sizeof(label)-1},
7047         {CKA_PRIME, prime, sizeof(prime)},
7048         {CKA_SUBPRIME, subprime, sizeof(subprime)},
7049         {CKA_BASE, base, sizeof(base)}},
7050     };

```

## 7051 6.2.6 DSA key pair generation

7052 The DSA key pair generation mechanism, denoted **CKM\_DSA\_KEY\_PAIR\_GEN**, is a key pair generation  
 7053 mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

7054 This mechanism does not have a parameter.

7055 The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as  
 7056 specified in the **CKA\_PRIME**, **CKA\_SUBPRIME**, and **CKA\_BASE** attributes of the template for the public  
 7057 key.

7058 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 7059 public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**, **CKA\_BASE**, and  
 7060 **CKA\_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private  
 7061 key types (specifically, the flags indicating which functions the keys support) may also be specified in the  
 7062 templates for the keys, or else are assigned default initial values.

7063 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 7064 specify the supported range of DSA prime sizes, in bits.

## 7065 6.2.7 DSA domain parameter generation

7066 The DSA domain parameter generation mechanism, denoted **CKM\_DSA\_PARAMETER\_GEN**, is a  
 7067 domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB  
 7068 186-2.

7069 This mechanism does not have a parameter.

7070 The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in  
 7071 the **CKA\_PRIME\_BITS** attribute of the template.

7072 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**,  
 7073 **CKA\_BASE** and **CKA\_PRIME\_BITS** attributes to the new object. Other attributes supported by the DSA  
 7074 domain parameter types may also be specified in the template, or else are assigned default initial values.

7075 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 7076 specify the supported range of DSA prime sizes, in bits.

## 7077 6.2.8 DSA probabilistic domain parameter generation

7078 The DSA probabilistic domain parameter generation mechanism, denoted  
 7079 **CKM\_DSA\_PROBABILISTIC\_PARAMETER\_GEN**, is a domain parameter generation mechanism based  
 7080 on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and  
 7081 Validation of Probable Primes..

7082 This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash and  
 7083 returns the seed (pSeed) and the length (ulSeedLen).

7084 The mechanism generates DSA the prime and subprime domain parameters with a particular prime  
 7085 length in bits, as specified in the **CKA\_PRIME\_BITS** attribute of the template and the subprime length as  
 7086 specified in the **CKA\_SUBPRIME\_BITS** attribute of the template.

7087 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**,  
7088 **CKA\_PRIME\_BITS**, and **CKA\_SUBPRIME\_BITS** attributes to the new object. **CKA\_BASE** is not set by  
7089 this call. Other attributes supported by the DSA domain parameter types may also be specified in the  
7090 template, or else are assigned default initial values.  
7091 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7092 specify the supported range of DSA prime sizes, in bits.

## 7093 **6.2.9 DSA Shawe-Taylor domain parameter generation**

7094 The DSA Shawe-Taylor domain parameter generation mechanism, denoted  
7095 **CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN**, is a domain parameter generation mechanism  
7096 based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2  
7097 Construction and Validation of Provable Primes p and q.  
7098 This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash and  
7099 returns the seed (pSeed) and the length (ulSeedLen).  
7100 The mechanism generates DSA the prime and subprime domain parameters with a particular prime  
7101 length in bits, as specified in the **CKA\_PRIME\_BITS** attribute of the template and the subprime length as  
7102 specified in the **CKA\_SUBPRIME\_BITS** attribute of the template.  
7103 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**,  
7104 **CKA\_PRIME\_BITS**, and **CKA\_SUBPRIME\_BITS** attributes to the new object. **CKA\_BASE** is not set by  
7105 this call. Other attributes supported by the DSA domain parameter types may also be specified in the  
7106 template, or else are assigned default initial values.  
7107 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7108 specify the supported range of DSA prime sizes, in bits.

## 7109 **6.2.10 DSA base domain parameter generation**

7110 The DSA base domain parameter generation mechanism, denoted **CKM\_DSA\_FIPS\_G\_GEN**, is a base  
7111 parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4,  
7112 section Appendix A.2 Generation of Generator G.  
7113 This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash the seed  
7114 (pSeed) and the length (ulSeedLen) and the index value.  
7115 The mechanism generates the DSA base with the domain parameter specified in the **CKA\_PRIME** and  
7116 **CKA\_SUBPRIME** attributes of the template.  
7117 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_BASE** attributes to the new  
7118 object. Other attributes supported by the DSA domain parameter types may also be specified in the  
7119 template, or else are assigned default initial values.  
7120 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7121 specify the supported range of DSA prime sizes, in bits.

## 7122 **6.2.11 DSA without hashing**

7123 The DSA without hashing mechanism, denoted **CKM\_DSA**, is a mechanism for single-part signatures and  
7124 verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism  
7125 corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash  
7126 value.)  
7127 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the  
7128 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.  
7129 It does not have a parameter.  
7130 Constraints on key types and the length of data are summarized in the following table:  
7131 *Table 53, DSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	DSA private key	20, 28, 32, 48, or 64 bytes	2*length of subprime
C_Verify <sup>1</sup>	DSA public key	(20, 28, 32, 48, or 64 bytes), (2*length of subprime) <sup>2</sup>	N/A

7132 1 Single-part operations only.

7133 2 Data length, signature length.

7134 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7135 specify the supported range of DSA prime sizes, in bits.

## 7136 6.2.12 DSA with SHA-1

7137 The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA1**, is a mechanism for single- and multiple-  
7138 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2.  
7139 This mechanism computes the entire DSA specification, including the hashing with SHA-1.

7140 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the  
7141 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

7142 This mechanism does not have a parameter.

7143 Constraints on key types and the length of data are summarized in the following table:

7144 *Table 54, DSA with SHA-1: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length <sup>2</sup>	N/A

7145 2 Data length, signature length.

7146 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7147 specify the supported range of DSA prime sizes, in bits.

## 7148 6.2.13 FIPS 186-4

7149 When CKM\_DSA is operated in FIPS mode, only the following bit lengths of *p* and *q*, represented by *L*  
7150 and *N*, SHALL be used:

7151 *L* = 1024, *N* = 160

7152 *L* = 2048, *N* = 224

7153 *L* = 2048, *N* = 256

7154 *L* = 3072, *N* = 256

7155

## 7156 6.2.14 DSA with SHA-224

7157 The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA224**, is a mechanism for single- and multiple-  
7158 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.  
7159 This mechanism computes the entire DSA specification, including the hashing with SHA-224.

7160 For the purposes of this mechanism, a DSA signature is a string of length  $2^{*}\text{subprime}$ , corresponding to  
 7161 the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.  
 7162 This mechanism does not have a parameter.  
 7163 Constraints on key types and the length of data are summarized in the following table:  
 7164 *Table 55, DSA with SHA-244: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*}\text{subprime}$ length
C_Verify	DSA public key	any, $2^{*}\text{subprime}$ length <sup>2</sup>	N/A

7165 <sup>2</sup> Data length, signature length.  
 7166 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 7167 specify the supported range of DSA prime sizes, in bits.

## 6.2.15 DSA with SHA-256

7169 The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA256**, is a mechanism for single- and multiple-  
 7170 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.  
 7171 This mechanism computes the entire DSA specification, including the hashing with SHA-256.  
 7172 For the purposes of this mechanism, a DSA signature is a string of length  $2^{*}\text{subprime}$ , corresponding to  
 7173 the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.  
 7174 This mechanism does not have a parameter.  
 7175 Constraints on key types and the length of data are summarized in the following table:  
 7176 *Table 56, DSA with SHA-256: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*}\text{subprime}$ length
C_Verify	DSA public key	any, $2^{*}\text{subprime}$ length <sup>2</sup>	N/A

7177 <sup>2</sup> Data length, signature length.

## 6.2.16 DSA with SHA-384

7179 The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA384**, is a mechanism for single- and multiple-  
 7180 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.  
 7181 This mechanism computes the entire DSA specification, including the hashing with SHA-384.  
 7182 For the purposes of this mechanism, a DSA signature is a string of length  $2^{*}\text{subprime}$ , corresponding to  
 7183 the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.  
 7184 This mechanism does not have a parameter.  
 7185 Constraints on key types and the length of data are summarized in the following table:  
 7186 *Table 57, DSA with SHA-384: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length <sup>2</sup>	N/A

7187 <sup>2</sup> Data length, signature length.

### 6.2.17 DSA with SHA-512

7189 The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.  
 7190 This mechanism computes the entire DSA specification, including the hashing with SHA-512.

7192 For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to  
 7193 the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.

7194 This mechanism does not have a parameter.

7195 Constraints on key types and the length of data are summarized in the following table:

7196 *Table 58, DSA with SHA-512: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length <sup>2</sup>	N/A

7197 <sup>2</sup> Data length, signature length.

### 6.2.18 DSA with SHA3-224

7199 The DSA with SHA3-224 mechanism, denoted **CKM\_DSA\_SHA3\_224**, is a mechanism for single- and  
 7200 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB  
 7201 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-224.

7202 For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to  
 7203 the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.

7204 This mechanism does not have a parameter.

7205 Constraints on key types and the length of data are summarized in the following table:

7206 *Table 59, DSA with SHA3-224: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length <sup>2</sup>	N/A

7207 <sup>2</sup> Data length, signature length.

7208 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 7209 specify the supported range of DSA prime sizes, in bits.

## 6.2.19 DSA with SHA3-256

The DSA with SHA3-256 mechanism, denoted **CKM\_DSA\_SHA3\_256**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-256.

For the purposes of this mechanism, a DSA signature is a string of length  $2^{*\text{subprime}}$ , corresponding to the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

*Table 60, DSA with SHA3-256: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*\text{subprime}}$ length
C_Verify	DSA public key	any, $2^{*\text{subprime}}$ length <sup>2</sup>	N/A

<sup>2</sup> Data length, signature length.

## 6.2.20 DSA with SHA3-384

The DSA with SHA3-384 mechanism, denoted **CKM\_DSA\_SHA3\_384**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-384.

For the purposes of this mechanism, a DSA signature is a string of length  $2^{*\text{subprime}}$ , corresponding to the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

*Table 61, DSA with SHA3-384: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*\text{subprime}}$ length
C_Verify	DSA public key	any, $2^{*\text{subprime}}$ length <sup>2</sup>	N/A

<sup>2</sup> Data length, signature length.

## 6.2.21 DSA with SHA3-512

The DSA with SHA3-512 mechanism, denoted **CKM\_DSA\_SHA3\_512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SH3A-512.

For the purposes of this mechanism, a DSA signature is a string of length  $2^{*\text{subprime}}$ , corresponding to the concatenation of the DSA values  $r$  and  $s$ , each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

*Table 62, DSA with SHA3-512: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length <sup>2</sup>	N/A

7239 <sup>2</sup> Data length, signature length.

7240

## 7241 6.3 Elliptic Curve

7242 The Elliptic Curve (EC) cryptosystem in this document was originally based on the one described in the  
7243 ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

7244 The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves  
7245 were always represented in their Weierstrass form. Since that time, new curves represented in Edwards  
7246 form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new  
7247 curves, the EC cryptosystem in this document has been extended from the original. Additional key  
7248 generation mechanisms have been added as well as an additional signature generation mechanism.

7249

7250 *Table 63, Elliptic Curve Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_EC_KEY_PAIR_GEN					✓		
CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS					✓		
CKM_EC_EDWARDS_KEY_PAIR_GEN					✓		
CKM_EC_MONTGOMERY_KEY_PAIR_GEN					✓		
CKM_ECDSA		✓ <sup>2</sup>					
CKM_ECDSA_SHA1		✓					
CKM_ECDSA_SHA224		✓					
CKM_ECDSA_SHA256		✓					
CKM_ECDSA_SHA384		✓					
CKM_ECDSA_SHA512		✓					
CKM_ECDSA_SHA3_224		✓					
CKM_ECDSA_SHA3_256		✓					
CKM_ECDSA_SHA3_384		✓					
CKM_ECDSA_SHA3_512		✓					
CKM_EDDSA		✓					
CKM_XEDDSA		✓					
CKM_ECDH1_DERIVE							✓
CKM_ECDH1_COFACTOR_DERIVE							✓
CKM_ECMQV_DERIVE							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ECDH_AES_KEY_WRAP						✓	

7251

7252

7253 Table 64, Mechanism Information Flags

CKF_EC_F_P	0x00100000UL	True if the mechanism can be used with EC domain parameters over $F_p$
CKF_EC_F_2M	0x00200000UL	True if the mechanism can be used with EC domain parameters over $F_{2^m}$
CKF_EC_ECPARAMETERS	0x00400000UL	True if the mechanism can be used with EC domain parameters of the choice <b>ecParameters</b>
CKF_EC_OID	0x00800000UL	True if the mechanism can be used with EC domain parameters of the choice <b>old</b>
CKF_EC_UNCOMPRESS	0x01000000UL	True if the mechanism can be used with Elliptic Curve point uncompressed
CKF_EC_COMPRESS	0x02000000UL	True if the mechanism can be used with Elliptic Curve point compressed
CKF_EC_CURVENAME	0x04000000UL	True of the mechanism can be used with EC domain parameters of the choice <b>curveName</b>

7254 Note: CKF\_EC\_NAMEDCURVE is deprecated with PKCS#11 3.00. It is replaced by CKF\_EC\_OID.

7255 In these standards, there are two different varieties of EC defined:

7256 1. EC using a field with an odd prime number of elements (i.e. the finite field  $F_p$ ).7257 2. EC using a field of characteristic two (i.e. the finite field  $F_{2^m}$ ).7258 An EC key in Cryptoki contains information about which variety of EC it is suited for. It is preferable that a Cryptoki library, which can perform EC mechanisms, be capable of performing operations with the two varieties of EC, however this is not required. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_F\_P** flag identifies a Cryptoki library supporting EC keys over  $F_p$  whereas the **CKF\_EC\_F\_2M** flag identifies a Cryptoki library supporting EC keys over  $F_{2^m}$ . A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.7264 In these specifications there are also four representation methods to define the domain parameters for an EC key. Only the **ecParameters**, the **old** and the **curveName** choices are supported in Cryptoki. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_ECPARAMETERS** flag identifies a Cryptoki library supporting the **ecParameters** choice whereas the **CKF\_EC\_OID** flag identifies a Cryptoki library supporting the **old** choice, and the **CKF\_EC\_CURVENAME** flag identifies a Cryptoki library supporting the **curveName** choice. A Cryptoki library that can perform EC mechanisms must set the appropriate flag(s) for each EC mechanism.7271 In these specifications, an EC public key (i.e. EC point Q) or the base point G when the **ecParameters** choice is used can be represented as an octet string of the uncompressed form or the compressed form. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_UNCOMPRESS** flag identifies a Cryptoki library supporting the uncompressed form whereas the **CKF\_EC\_COMPRESS** flag identifies a Cryptoki library

7275 supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or  
7276 both of these flags for each EC mechanism.

7277 Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation  
7278 of domain parameters or one form may encounter difficulties achieving interoperability with other  
7279 implementations.

7280 If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the  
7281 attempt should fail with the error code CKR\_CURVE\_NOT\_SUPPORTED. If an attempt to create,  
7282 generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain  
7283 parameters is made, that attempt should fail with the error code CKR\_DOMAIN\_PARAMS\_INVALID. If  
7284 an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that  
7285 attempt should fail with the error code CKR\_TEMPLATE\_INCONSISTENT.

### 7286 **6.3.1 EC Signatures**

7287 For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at  
7288 most two times  $nLen$  octets, where  $nLen$  is the length in octets of the base point order  $n$ . The signature  
7289 octets correspond to the concatenation of the ECDSA values  $r$  and  $s$ , both represented as an octet string  
7290 of equal length of at most  $nLen$  with the most significant byte first. If  $r$  and  $s$  have different octet length,  
7291 the shorter of both must be padded with leading zero octets such that both have the same octet length.  
7292 Loosely spoken, the first half of the signature is  $r$  and the second half is  $s$ . For signatures created by a  
7293 token, the resulting signature is always of length  $2nLen$ . For signatures passed to a token for verification,  
7294 the signature may have a shorter length but must be composed as specified before.

7295 If the length of the hash value is larger than the bit length of  $n$ , only the leftmost bits of the hash up to the  
7296 length of  $n$  will be used. Any truncation is done by the token.

7297 Note: For applications, it is recommended to encode the signature as an octet string of length two times  
7298  $nLen$  if possible. This ensures that the application works with PKCS#11 modules which have been  
7299 implemented based on an older version of this document. Older versions required all signatures to have  
7300 length two times  $nLen$ . It may be impossible to encode the signature with the maximum length of two  
7301 times  $nLen$  if the application just gets the integer values of  $r$  and  $s$  (i.e. without leading zeros), but does  
7302 not know the base point order  $n$ , because  $r$  and  $s$  can have any value between zero and the base point  
7303 order  $n$ .

7304 An EdDSA signature is an octet string of even length which is two times  $nLen$  octets, where  $nLen$  is  
7305 calculated as EdDSA parameter  $b$  divided by 8. The signature octets correspond to the concatenation of  
7306 the EdDSA values  $R$  and  $S$  as defined in [RFC 8032], both represented as an octet string of equal length  
7307 of  $nLen$  bytes in little endian order.

### 7308 **6.3.2 Definitions**

7309 This section defines the key types "CKK\_EC", "CKK\_EC\_EDWARDS" and "CKK\_EC\_MONTGOMERY"  
7310 for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

7311 Note: CKK\_ECDSA is deprecated. It is replaced by CKK\_EC.

7312 Mechanisms:

7313     CKM\_EC\_KEY\_PAIR\_GEN  
7314     CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN  
7315     CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN  
7316     CKM\_ECDSA  
7317     CKM\_ECDSA\_SHA1  
7318     CKM\_ECDSA\_SHA224  
7319     CKM\_ECDSA\_SHA256  
7320     CKM\_ECDSA\_SHA384

7322 CKM\_ECDSA\_SHA512  
7323 CKM\_ECDSA\_SHA3\_224  
7324 CKM\_ECDSA\_SHA3\_256  
7325 CKM\_ECDSA\_SHA3\_384  
7326 CKM\_ECDSA\_SHA3\_512  
7327 CKM\_EDDSA  
7328 CKM\_XEDDSA  
7329 CKM\_ECDH1\_DERIVE  
7330 CKM\_ECDH1\_COFACTOR\_DERIVE  
7331 CKM\_ECMQV\_DERIVE  
7332 CKM\_ECDH\_AES\_KEY\_WRAP  
7333  
7334 CKD\_NULL  
7335 CKD\_SHA1\_KDF  
7336 CKD\_SHA224\_KDF  
7337 CKD\_SHA256\_KDF  
7338 CKD\_SHA384\_KDF  
7339 CKD\_SHA512\_KDF  
7340 CKD\_SHA3\_224\_KDF  
7341 CKD\_SHA3\_256\_KDF  
7342 CKD\_SHA3\_384\_KDF  
7343 CKD\_SHA3\_512\_KDF  
7344 CKD\_SHA1\_KDF\_SP800  
7345 CKD\_SHA224\_KDF\_SP800  
7346 CKD\_SHA256\_KDF\_SP800  
7347 CKD\_SHA384\_KDF\_SP800  
7348 CKD\_SHA512\_KDF\_SP800  
7349 CKD\_SHA3\_224\_KDF\_SP800  
7350 CKD\_SHA3\_256\_KDF\_SP800  
7351 CKD\_SHA3\_384\_KDF\_SP800  
7352 CKD\_SHA3\_512\_KDF\_SP800  
7353 CKD\_BLAKE2B\_160\_KDF  
7354 CKD\_BLAKE2B\_256\_KDF  
7355 CKD\_BLAKE2B\_384\_KDF  
7356 CKD\_BLAKE2B\_512\_KDF

### 7357 6.3.3 Short Weierstrass Elliptic Curve public key objects

7358 Short Weierstrass EC public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_EC**) hold EC  
7359 public keys. The following table defines the EC public key object attributes, in addition to the common  
7360 attributes defined for this object class:

7361 *Table 65, Elliptic Curve Public Key Object Attributes*

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
CKA_EC_PARAMS <sup>1,3</sup>	Byte array	DER-encoding of an ANSI X9.62 Parameters value
CKA_EC_POINT <sup>1,4</sup>	Byte array	DER-encoding of ANSI X9.62 ECPoint value Q

7362 - Refer to Table 11 for footnotes

7363 Note: CKA\_ECDSA\_PARAMS is deprecated. It is replaced by CKA\_EC\_PARAMS.

7364 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI  
7365 X9.62 as a choice of three parameter representation methods with the following syntax:

```
7366     Parameters ::= CHOICE {
7367         ecParameters    ECParameters,
7368         oid             CURVES.&id({CurveNames}),
7369         implicitlyCA   NULL,
7370         curveName      PrintableString
7371     }
```

7372

7373 This allows detailed specification of all required values using choice **ecParameters**, the use of **oid** as an  
7374 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to  
7375 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve  
7376 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oid** or  
7377 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used  
7378 in Cryptoki.

7379 The following is a sample template for creating an short Weierstrass EC public key object:

```
7380     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7381     CK_KEY_TYPE keyType = CKK_EC;
7382     CK_UTF8CHAR label[] = "An EC public key object";
7383     CK_BYTE ecParams[] = {...};
7384     CK_BYTE ecPoint[] = {...};
7385     CK_BBOOL true = CK_TRUE;
7386     CK_ATTRIBUTE template[] = {
7387         {CKA_CLASS, &class, sizeof(class)},
7388         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7389         {CKA_TOKEN, &true, sizeof(true)},
7390         {CKA_LABEL, label, sizeof(label)-1},
7391         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7392         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
7393     };
```

### 7394 6.3.4 Short Weierstrass Elliptic Curve private key objects

7395 Short Weierstrass EC private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_EC**) hold  
7396 EC private keys. See Section 6.3 for more information about EC. The following table defines the EC  
7397 private key object attributes, in addition to the common attributes defined for this object class:

7398 *Table 66, Elliptic Curve Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_EC_PARAMS <sup>1,4,6</sup>	Byte array	DER-encoding of an ANSI X9.62 Parameters value
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	ANSI X9.62 private value $d$

7399 - Refer to Table 11 for footnotes

7400 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI  
7401 X9.62 as a choice of three parameter representation methods with the following syntax:

```
7402     Parameters ::= CHOICE {
7403         ecParameters    ECParameters,
7404         oId             CURVES.&id({CurveNames}),
7405         implicitlyCA   NULL,
7406         curveName       PrintableString
7407     }
```

7408  
7409 This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an  
7410 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to  
7411 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve  
7412 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or  
7413 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used  
7414 in Cryptoki. Note that when generating an EC private key, the EC domain parameters are *not* specified in  
7415 the key’s template. This is because EC private keys are only generated as part of an EC key pair, and  
7416 the EC domain parameters for the pair are specified in the template for the EC public key.

7417 The following is a sample template for creating an short Weierstrass EC private key object:

```
7418     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7419     CK_KEY_TYPE keyType = CKK_EC;
7420     CK_UTF8CHAR label[] = "An EC private key object";
7421     CK_BYTE subject[] = {...};
7422     CK_BYTE id[] = {123};
7423     CK_BYTE ecParams[] = {...};
7424     CK_BYTE value[] = {...};
7425     CK_BBOOL true = CK_TRUE;
7426     CK_ATTRIBUTE template[] = {
7427         {CKA_CLASS, &class, sizeof(class)},
7428         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7429         {CKA_TOKEN, &true, sizeof(true)},
7430         {CKA_LABEL, label, sizeof(label)-1},
7431         {CKA_SUBJECT, subject, sizeof(subject)},
7432         {CKA_ID, id, sizeof(id)},
7433         {CKA_SENSITIVE, &true, sizeof(true)},
7434         {CKA_DERIVE, &true, sizeof(true)},
7435         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7436         {CKA_VALUE, value, sizeof(value)}
7437     };
```

### 7438 6.3.5 Edwards Elliptic Curve public key objects

7439 Edwards EC public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_EC\_EDWARDS**) hold  
7440 Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in  
7441 addition to the common attributes defined for this object class:

7442 *Table 67, Edwards Elliptic Curve Public Key Object Attributes*

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
CKA_EC_PARAMS <sup>1,3</sup>	Byte array	DER-encoding of a Parameters value as defined above
CKA_EC_POINT <sup>1,4</sup>	Byte array	Public key bytes in little endian order as defined in RFC 8032

7443 Refer to Table 11 for footnotes

7444 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI  
 7445 X9.62 as a choice of three parameter representation methods. A 4<sup>th</sup> choice is added to support Edwards  
 7446 and Montgomery Elliptic Curves. The CKA\_EC\_PARAMS attribute has the following syntax:

```
7447   Parameters ::= CHOICE {
7448     ecParameters  ECParameters,
7449     oId           CURVES.&id({CurveNames}),
7450     implicitlyCA NULL,
7451     curveName    PrintableString
7452 }
```

7453 Edwards EC public keys only support the use of the **curveName** selection to specify a curve name as  
 7454 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as  
 7455 defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

7456 The following is a sample template for creating an Edwards EC public key object with Edwards25519  
 7457 being specified as curveName:

```
7458   CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7459   CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
7460   CK_UTF8CHAR label[] = "An Edwards EC public key object";
7461   CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,
7462                         0x72, 0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39};
7463   CK_BYTE ecPoint[] = {...};
7464   CK_BBOOL true = CK_TRUE;
7465   CK_ATTRIBUTE template[] = {
7466     {CKA_CLASS, &class, sizeof(class)},
7467     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7468     {CKA_TOKEN, &true, sizeof(true)},
7469     {CKA_LABEL, label, sizeof(label)-1},
7470     {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7471     {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}}
7472 };
```

### 7473 **6.3.6 Edwards Elliptic Curve private key objects**

7474 Edwards EC private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_EC\_EDWARDS**)  
 7475 hold Edwards EC private keys. See Section 6.3 for more information about EC. The following table  
 7476 defines the Edwards EC private key object attributes, in addition to the common attributes defined for this  
 7477 object class:

7478 *Table 68, Edwards Elliptic Curve Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_EC_PARAMS <sup>1,4,6</sup>	Byte array	DER-encoding of a Parameters value as defined above
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	Private key bytes in little endian order as defined in RFC 8032

7479 - Refer to Table 11 for footnotes

7480 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI  
 7481 X9.62 as a choice of three parameter representation methods. A 4<sup>th</sup> choice is added to support Edwards  
 7482 and Montgomery Elliptic Curves. The CKA\_EC\_PARAMS attribute has the following syntax:

```
7483   Parameters ::= CHOICE {
7484     ecParameters  ECParameters,
7485     oId           CURVES.&id({CurveNames}),
7486     implicitlyCA NULL,
7487     curveName    PrintableString
7488 }
```

7489 Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as  
 7490 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as  
 7491 defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

7492 Note that when generating an Edwards EC private key, the EC domain parameters are *not* specified in  
 7493 the key's template. This is because Edwards EC private keys are only generated as part of an Edwards  
 7494 EC key pair, and the EC domain parameters for the pair are specified in the template for the Edwards EC  
 7495 public key.

7496 The following is a sample template for creating an Edwards EC private key object:

```
7497   CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7498   CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
7499   CK_UTF8CHAR label[] = "An Edwards EC private key object";
7500   CK_BYTE subject[] = {...};
7501   CK_BYTE id[] = {123};
7502   CK_BYTE ecParams[] = {...};
7503   CK_BYTE value[] = {...};
7504   CK_BBOOL true = CK_TRUE;
7505   CK_ATTRIBUTE template[] = {
7506     {CKA_CLASS, &class, sizeof(class)},
7507     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7508     {CKA_TOKEN, &true, sizeof(true)},
7509     {CKA_LABEL, label, sizeof(label)-1},
7510     {CKA_SUBJECT, subject, sizeof(subject)},
7511     {CKA_ID, id, sizeof(id)},
7512     {CKA_SENSITIVE, &true, sizeof(true)},
7513     {CKA_DERIVE, &true, sizeof(true)},
7514     {CKA_VALUE, value, sizeof(value)}}
7515 };
```

### 7516 6.3.7 Montgomery Elliptic Curve public key objects

7517 Montgomery EC public key objects (object class **CKO\_PUBLIC\_KEY**, key type  
 7518 **CKK\_EC\_MONTGOMERY**) hold Montgomery EC public keys. The following table defines the  
 7519 Montgomery EC public key object attributes, in addition to the common attributes defined for this object  
 7520 class:

7521 Table 69, Montgomery Elliptic Curve Public Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS <sup>1,3</sup>	Byte array	DER-encoding of a Parameters value as defined above
CKA_EC_POINT <sup>1,4</sup>	Byte array	Public key bytes in little endian order as defined in RFC 7748

7522 Refer to Table 11 for footnotes

7523 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods. A 4<sup>th</sup> choice is added to support Edwards and Montgomery Elliptic Curves. The CKA\_EC\_PARAMS attribute has the following syntax:

```
7526     Parameters ::= CHOICE {
7527         ecParameters    ECParameters,
7528         oId             CURVES.&id({CurveNames}),
7529         implicitlyCA   NULL,
7530         curveName      PrintableString
7531     }
```

7532 Montgomery EC public keys only support the use of the **curveName** selection to specify a curve name as defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm as defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

7533 The following is a sample template for creating a Montgomery EC public key object:

```
7536     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7537     CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
7538     CK_UTF8CHAR label[] = "A Montgomery EC public key object";
7539     CK_BYTE ecParams[] = {...};
7540     CK_BYTE ecPoint[] = {...};
7541     CK_BBOOL true = CK_TRUE;
7542     CK_ATTRIBUTE template[] = {
7543         {CKA_CLASS, &class, sizeof(class)},
7544         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7545         {CKA_TOKEN, &true, sizeof(true)},
7546         {CKA_LABEL, label, sizeof(label)-1},
7547         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
7548         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}}
7549 }
```

### 7550 6.3.8 Montgomery Elliptic Curve private key objects

7551 Montgomery EC private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_EC\_MONTGOMERY**) hold Montgomery EC private keys. See Section 6.3 for more information about EC. The following table defines the Montgomery EC private key object attributes, in addition to the common attributes defined for this object class:

7555 Table 70, Montgomery Elliptic Curve Private Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS <sup>1,4,6</sup>	Byte array	DER-encoding of a Parameters value as defined above
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	Private key bytes in little endian order as defined in RFC 7748

7556 Refer to Table 11 for footnotes

7557 The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI  
7558 X9.62 as a choice of three parameter representation methods. A 4<sup>th</sup> choice is added to support Edwards  
7559 and Montgomery Elliptic Curves. The **CKA\_EC\_PARAMS** attribute has the following syntax:

```
7560     Parameters ::= CHOICE {
7561         ecParameters    ECParameters,
7562         oId             CURVES.&id({CurveNames}),
7563         implicitlyCA   NULL,
7564         curveName       PrintableString
7565     }
```

7566 Montgomery EC private keys only support the use of the **curveName** selection to specify a curve name  
7567 as defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm  
7568 as defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

7569 Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in  
7570 the key's template. This is because Montgomery EC private keys are only generated as part of a  
7571 Montgomery EC key pair, and the EC domain parameters for the pair are specified in the template for the  
7572 Montgomery EC public key.

7573 The following is a sample template for creating a Montgomery EC private key object:

```
7574     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7575     CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
7576     CK_UTF8CHAR label[] = "A Montgomery EC private key object";
7577     CK_BYTE subject[] = {...};
7578     CK_BYTE id[] = {123};
7579     CK_BYTE ecParams[] = {...};
7580     CK_BYTE value[] = {...};
7581     CK_BBOOL true = CK_TRUE;
7582     CK_ATTRIBUTE template[] = {
7583         {CKA_CLASS, &class, sizeof(class)},
7584         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7585         {CKA_TOKEN, &true, sizeof(true)},
7586         {CKA_LABEL, label, sizeof(label)-1},
7587         {CKA_SUBJECT, subject, sizeof(subject)},
7588         {CKA_ID, id, sizeof(id)},
7589         {CKA_SENSITIVE, &true, sizeof(true)},
7590         {CKA_DERIVE, &true, sizeof(true)},
7591         {CKA_VALUE, value, sizeof(value)}
7592     };
```

### 7593 6.3.9 Elliptic Curve key pair generation

7594 The short Weierstrass EC key pair generation mechanism, denoted CKM\_EC\_KEY\_PAIR\_GEN, is a key  
7595 pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

7596 The short Weierstrass EC key pair generation mechanism, denoted  
7597 CKM\_EC\_KEY\_PAIR\_GEN\_W\_EXTRA\_BITS, is a key pair generation mechanism that uses the method  
7598 defined by FIPS 186-4 Appendix B.4.1.

7599 These mechanisms do not have a parameter.

7600 These mechanisms generate EC public/private key pairs with particular EC domain parameters, as  
7601 specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key. Note that this version of  
7602 Cryptoki does not include a mechanism for generating these EC domain parameters.

7603 These mechanism contribute the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the  
7604 new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE**  
7605 attributes to the new private key. Other attributes supported by the EC public and private key types  
7606 (specifically, the flags indicating which functions the keys support) may also be specified in the templates  
7607 for the keys, or else are assigned default initial values.  
7608 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7609 specify the minimum and maximum supported number of bits in the field sizes, respectively. For  
7610 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between  
7611  $2^{200}$  and  $2^{300}$  elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary  
7612 notation, the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.  
7613 Similarly,  $2^{300}$  is a 301-bit number).

### 7614 6.3.10 Edwards Elliptic Curve key pair generation

7615 The Edwards EC key pair generation mechanism, denoted **CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN**, is a  
7616 key pair generation mechanism for EC keys over curves represented in Edwards form.

7617 This mechanism does not have a parameter.

7618 The mechanism can only generate EC public/private key pairs over the curves edwards25519 and  
7619 edwards448 as defined in RFC 8032 or the curves id-Ed25519 and id-Ed448 as defined in RFC 8410.  
7620 These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key  
7621 using the **curveName** or the **oid** methods. Attempts to generate keys over these curves using any other  
7622 EC key pair generation mechanism will fail with **CKR\_CURVE\_NOT\_SUPPORTED**.

7623 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the  
7624 new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE**  
7625 attributes to the new private key. Other attributes supported by the Edwards EC public and private key  
7626 types (specifically, the flags indicating which functions the keys support) may also be specified in the  
7627 templates for the keys, or else are assigned default initial values.

7628 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7629 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this  
7630 mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two  
7631 sizes. A Cryptoki implementation may support one or both of these curves and should set the  
7632 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### 7633 6.3.11 Montgomery Elliptic Curve key pair generation

7634 The Montgomery EC key pair generation mechanism, denoted  
7635 **CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC keys over  
7636 curves represented in Montgomery form.

7637 This mechanism does not have a parameter.

7638 The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519  
7639 and curve448 as defined in RFC 7748 or the curves id-X25519 and id-X448 as defined in RFC 8410.  
7640 These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key  
7641 using the **curveName** or **old** methods. Attempts to generate keys over these curves using any other EC  
7642 key pair generation mechanism will fail with **CKR\_CURVE\_NOT\_SUPPORTED**.

7643 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the  
7644 new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE**  
7645 attributes to the new private key. Other attributes supported by the EC public and private key types  
7646 (specifically, the flags indicating which functions the keys support) may also be specified in the templates  
7647 for the keys, or else are assigned default initial values.

7648 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7649 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this  
7650 mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two  
7651 sizes. A Cryptoki implementation may support one or both of these curves and should set the  
7652 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### 6.3.12 ECDSA without hashing

Refer section 6.3.1 for signature encoding.

The ECDSA without hashing mechanism, denoted **CKM\_ECDSA**, is a mechanism for single-part signatures and verification for ECDSA. (This mechanism corresponds only to the part of ECDSA that processes the hash value, which should not be longer than 1024 bits; it does not compute the hash value.)

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 71, ECDSA without hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	CKK_EC private key	any <sup>3</sup>	2nLen
C_Verify <sup>1</sup>	CKK_EC public key	any <sup>3</sup> , ≤2nLen <sup>2</sup>	N/A

<sup>1</sup> Single-part operations only.

<sup>2</sup> Data length, signature length.

<sup>3</sup> Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between  $2^{200}$  and  $2^{300}$  elements (inclusive), then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly,  $2^{300}$  is a 301-bit number).

### 6.3.13 ECDSA with hashing

Refer to section 6.3.1 for signature encoding.

The ECDSA with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanism, denoted **CKM\_ECDSA\_[SHA1|SHA224|SHA256|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]** respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 72, ECDSA with hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign	CKK_EC private key	any	2nLen
C_Verify	CKK_EC public key	any, ≤2nLen <sup>2</sup>	N/A

<sup>2</sup> Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between  $2^{200}$  and  $2^{300}$  elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly,  $2^{300}$  is a 301-bit number).

### 7689 6.3.14 EdDSA

7690 The EdDSA mechanism, denoted **CKM\_EDDSA**, is a mechanism for single-part and multipart signatures  
7691 and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in  
7692 RFC 8032 and RFC 8410.

7693 For curves according to RFC 8032, this mechanism has an optional parameter, a **CK\_EDDSA\_PARAMS**  
7694 structure. The absence or presence of the parameter as well as its content is used to identify which  
7695 signature scheme is to be used. The following table enumerates the five signature schemes defined in  
7696 RFC 8032 and all supported permutations of the mechanism parameter and its content.

7697 *Table 73, Mapping to RFC 8032 Signature Schemes*

Signature Scheme	Mechanism Param	phFlag	Context Data
Ed25519	Not Required	N/A	N/A
Ed25519ctx	Required	False	Optional
Ed25519ph	Required	True	Optional
Ed448	Required	False	Optional
Ed448ph	Required	True	Optional

7698 For curves according to RFC 8410, the mechanism is implicitly given by the curve, which is EdDSA in  
7699 pure mode.

7700 Constraints on key types and the length of data are summarized in the following table:

7701 *Table 74, EdDSA: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign	CKK_EC_EDWARDS private key	any	2bLen
C_Verify	CKK_EC_EDWARDS public key	any, $\leq 2bLen^2$	N/A

7702 2 Data length, signature length.

7703 Note that for EdDSA in pure mode, Ed25519 and Ed448 the data must be processed twice. Therefore, a  
7704 token might need to cache all the data, especially when used with C\_SignUpdate/C\_VerifyUpdate. If  
7705 tokens are unable to do so they can return CKR\_TOKEN\_RESOURCE\_EXCEEDED.

7706 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK\_MECHANISM\_INFO structure  
7707 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this  
7708 mechanism, the only allowed values are 255 and 448 as RFC 8032 and RFC 8410 only define curves of  
7709 these two sizes. A Cryptoki implementation may support one or both of these curves and should set the  
7710 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### 7711 6.3.15 XEdDSA

7712 The XEdDSA mechanism, denoted **CKM\_XEDDSA**, is a mechanism for single-part signatures and  
7713 verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in  
7714 [XEDDSA]. CKM\_XEDDSA operates on CKK\_EC\_MONTGOMERY type EC keys, which allows these  
7715 keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use  
7716 is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-  
7717 term keys and also contributes to the DH key-exchange.

7718 This mechanism has a parameter, a **CK\_XEDDSA\_PARAMS** structure.

7719 *Table 75, XEdDSA: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign <sup>1</sup>	CKK_EC_MONTGOMERY private key	any <sup>3</sup>	2b
C_Verify <sup>1</sup>	CKK_EC_MONTGOMERY public key	any <sup>3</sup> , ≤2b <sup>2</sup>	N/A

7720 2 Data length, signature length.

7721 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 7722 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this  
 7723 mechanism, the only allowed values are 255 and 448 as [XEDDSA] only defines curves of these two  
 7724 sizes. A Cryptoki implementation may support one or both of these curves and should set the  
 7725 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### 7726 6.3.16 EC mechanism parameters

7727 ♦ **CK\_EDDSA\_PARAMS, CK\_EDDSA\_PARAMS\_PTR**

7728 **CK\_EDDSA\_PARAMS** is a structure that provides the parameters for the **CKM\_EDDSA** signature  
 7729 mechanism. The structure is defined as follows:

```
7730     typedef struct CK_EDDSA_PARAMS {
7731         CK_BBOOL      phFlag;
7732         CK ULONG      ulContextDataLen;
7733         CK_BYTE_PTR   pContextData;
7734     } CK_EDDSA_PARAMS;
```

7735

7736 The fields of the structure have the following meanings:

7737 phFlag Boolean value which indicates if Prehashed variant of EdDSA should be used.  
 7738 ulContextDataLen the length in bytes of the context data where 0 <= ulContextDataLen <= 255.

7739 pContextData context data shared between the signer and verifier

7740 **CK\_EDDSA\_PARAMS\_PTR** is a pointer to a **CK\_EDDSA\_PARAMS**.

7741

7742 ♦ **CK\_XEDDSA\_PARAMS, CK\_XEDDSA\_PARAMS\_PTR**

7743 **CK\_XEDDSA\_PARAMS** is a structure that provides the parameters for the **CKM\_XEDDSA** signature  
 7744 mechanism. The structure is defined as follows:

```
7745     typedef struct CK_XEDDSA_PARAMS {
7746         CK_XEDDSA_HASH_TYPE hash;
7747     } CK_XEDDSA_PARAMS;
```

7748

7749 The fields of the structure have the following meanings:

7750 hash a Hash mechanism to be used by the mechanism.

7751 **CK\_XEDDSA\_PARAMS\_PTR** is a pointer to a **CK\_XEDDSA\_PARAMS**.

7752

7753 ♦ **CK\_XEDDSA\_HASH\_TYPE, CK\_XEDDSA\_HASH\_TYPE\_PTR**

7754 **CK\_XEDDSA\_HASH\_TYPE** is used to indicate the hash function used in XEDDSA. It is defined as  
 7755 follows:

```
7756     typedef CK ULONG CK_XEDDSA_HASH_TYPE;
```

7757

7758 The following table lists the defined functions.

7759 *Table 76, EC: Key Derivation Functions*

Source Identifier
CKM_BLAKE2B_256
CKM_BLAKE2B_512
CKM_SHA3_256
CKM_SHA3_512
CKM_SHA256
CKM_SHA512

7760

7761 **CK\_XEDDSA\_HASH\_TYPE\_PTR** is a pointer to a **CK\_XEDDSA\_HASH\_TYPE**.

7762

7763 ♦ **CK\_EC\_KDF\_TYPE, CK\_EC\_KDF\_TYPE\_PTR**

7764 **CK\_EC\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data  
7765 from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is  
7766 defined as follows:

7767     **typedef CK ULONG CK\_EC\_KDF\_TYPE;**

7768

7769 The following table lists the defined functions.

7770 *Table 77, EC: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_SHA1_KDF
CKD_SHA224_KDF
CKD_SHA256_KDF
CKD_SHA384_KDF
CKD_SHA512_KDF
CKD_SHA3_224_KDF
CKD_SHA3_256_KDF
CKD_SHA3_384_KDF
CKD_SHA3_512_KDF
CKD_SHA1_KDF_SP800
CKD_SHA224_KDF_SP800
CKD_SHA256_KDF_SP800
CKD_SHA384_KDF_SP800
CKD_SHA512_KDF_SP800
CKD_SHA3_224_KDF_SP800
CKD_SHA3_256_KDF_SP800
CKD_SHA3_384_KDF_SP800
CKD_SHA3_512_KDF_SP800
CKD_BLAKE2B_160_KDF

CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_384_KDF
CKD_BLAKE2B_512_KDF

7771 The key derivation function **CKD\_NULL** produces a raw shared secret value without applying any key  
 7772 derivation function.

7773 The key derivation functions  
 7774 **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF**, which are  
 7775 based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512  
 7776 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

7777 The key derivation functions  
 7778 **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**,  
 7779 which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512  
 7780 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section  
 7781 5.8.1.1.

7782 The key derivation functions **CKD\_BLAKE2B\_[160|256|384|512]\_KDF**, which are based on the Blake2b  
 7783 family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section  
 7784 5.8.1.1. **CK\_EC\_KDF\_TYPE\_PTR** is a pointer to a **CK\_EC\_KDF\_TYPE**.

7785

7786 ♦ **CK\_ECDH1\_DERIVE\_PARAMS**, **CK\_ECDH1\_DERIVE\_PARAMS\_PTR**

7787 **CK\_ECDH1\_DERIVE\_PARAMS** is a structure that provides the parameters for the  
 7788 **CKM\_ECDH1\_DERIVE** and **CKM\_ECDH1\_COFACTOR\_DERIVE** key derivation mechanisms, where  
 7789 each party contributes one key pair. The structure is defined as follows:

```
7790     typedef struct CK_ECDH1_DERIVE_PARAMS {
7791         CK_EC_KDF_TYPE    kdf;
7792         CK ULONG          ulSharedDataLen;
7793         CK BYTE PTR      pSharedData;
7794         CK ULONG          ulPublicDataLen;
7795         CK BYTE PTR      pPublicData;
7796     } CK_ECDH1_DERIVE_PARAMS;
```

7797

7798 The fields of the structure have the following meanings:

7799                           kdf	key derivation function used on the shared secret value
7800                           ulSharedDataLen	the length in bytes of the shared info
7801                           pSharedData	some data shared between the two parties
7802                           ulPublicDataLen	the length in bytes of the other party's EC public key

7803                    pPublicData<sup>1</sup>      pointer to other party's EC public key value. For short Weierstrass  
 7804                    EC keys: a token MUST be able to accept this value encoded as a  
 7805                    raw octet string (as per section A.5.2 of [ANSI X9.62]). A token  
 7806                    MAY, in addition, support accepting this value as a DER-encoded  
 7807                    ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a  
 7808                    CKA\_EC\_POINT encoding. The calling application is responsible  
 7809                    for converting the offered public key to the compressed or  
 7810                    uncompressed forms of these encodings if the token does not  
 7811                    support the offered form.  
 7812                    For Montgomery keys: the public key is provided as bytes in little  
 7813                    endian order as defined in RFC 7748.

7814                    With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation functions  
 7815                    **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF**,  
 7816                    **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**, an  
 7817                    optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending  
 7818                    to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.  
 7819

7820                    **CK\_ECDH1\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECDH1\_DERIVE\_PARAMS**.  
 7821                    ♦ **CK\_ECDH2\_DERIVE\_PARAMS**, **CK\_ECDH2\_DERIVE\_PARAMS\_PTR**

7822                    **CK\_ECDH2\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
 7823                    **CKM\_ECMQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The  
 7824                    structure is defined as follows:

```

    7825                    typedef struct CK_ECDH2_DERIVE_PARAMS {
    7826                        CK_EKDF_TYPE kdf;
    7827                        CK ULONG ulSharedDataLen;
    7828                        CK_BYTE_PTR pSharedData;
    7829                        CK ULONG ulPublicDataLen;
    7830                        CK_BYTE_PTR pPublicData;
    7831                        CK ULONG ulPrivateDataLen;
    7832                        CK_OBJECT_HANDLE hPrivateData;
    7833                        CK ULONG ulPublicDataLen2;
    7834                        CK_BYTE_PTR pPublicData2;
    7835                    } CK_ECDH2_DERIVE_PARAMS;
  
```

7836

7837                    The fields of the structure have the following meanings:

7838                    kdf	key derivation function used on the shared secret value
7839                    ulSharedDataLen	the length in bytes of the shared info
7840                    pSharedData	some data shared between the two parties
7841                    ulPublicDataLen	the length in bytes of the other party's first EC public key
7842                    pPublicData	pointer to other party's first EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS
7844                    ulPrivateDataLen	the length in bytes of the second EC private key

---

1. The encoding in V2.20 was not specified and resulted in different implementations choosing different encodings. Applications relying only on a V2.20 encoding (e.g. the DER variant) other than the one specified now (raw) may not work with all V2.30 compliant tokens.

7845                    hPrivateData      key handle for second EC private key value  
7846                    ulPublicDataLen2   the length in bytes of the other party's second EC public key  
7847                    pPublicData2      pointer to other party's second EC public key value. Encoding rules  
7848                    are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

7849 With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be  
7850 zero. With the key derivation function **CKD\_SHA1\_KDF**, an optional *pSharedData* may be supplied,  
7851 which consists of some data shared by the two parties intending to share the shared secret. Otherwise,  
7852 *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

7853 **CK\_ECDH2\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECDH2\_DERIVE\_PARAMS**.

7854

7855 ◆ **CK\_ECMQV\_DERIVE\_PARAMS, CK\_ECMQV\_DERIVE\_PARAMS\_PTR**

7856 **CK\_ECMQV\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
7857 **CKM\_ECMQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The  
7858 structure is defined as follows:

```
7859       typedef struct CK_ECMQV_DERIVE_PARAMS {  
7860           CK_EC_KDF_TYPE       kdf;  
7861           CK ULONG           ulSharedDataLen;  
7862           CK_BYTE_PTR       pSharedData;  
7863           CK ULONG           ulPublicDataLen;  
7864           CK_BYTE_PTR       pPublicData;  
7865           CK ULONG           ulPrivateDataLen;  
7866           CK_OBJECT_HANDLE hPrivateKey;  
7867           CK ULONG           ulPublicDataLen2;  
7868           CK_BYTE_PTR       pPublicData2;  
7869           CK_OBJECT_HANDLE publicKey;  
7870       } CK_ECMQV_DERIVE_PARAMS;
```

7871

7872 The fields of the structure have the following meanings:

7873                    kdf      key derivation function used on the shared secret value  
7874                    ulSharedDataLen   the length in bytes of the shared info  
7875                    pSharedData    some data shared between the two parties  
7876                    ulPublicDataLen   the length in bytes of the other party's first EC public key  
7877                    pPublicData    pointer to other party's first EC public key value. Encoding rules are  
7878                    as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS  
7879                    ulPrivateDataLen   the length in bytes of the second EC private key  
7880                    hPrivateKey    key handle for second EC private key value  
7881                    ulPublicDataLen2   the length in bytes of the other party's second EC public key  
7882                    pPublicData2    pointer to other party's second EC public key value. Encoding rules  
7883                    are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS  
7884                    publicKey     Handle to the first party's ephemeral public key

7885 With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be  
7886 zero. With the key derivation functions  
7887 **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF**,  
7888 **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**, an

7889 optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending  
7890 to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.  
7891 **CK\_ECMQV\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECMQV\_DERIVE\_PARAMS**.

### 7892 6.3.17 Elliptic Curve Diffie-Hellman key derivation

7893 The Elliptic Curve Diffie-Hellman (ECDH) key derivation mechanism, denoted **CKM\_ECDH1\_DERIVE**, is  
7894 a mechanism for key derivation based on the Diffie-Hellman version of the Elliptic Curve key agreement  
7895 scheme, as defined in ANSI X9.63 for short Weierstrass EC keys and RFC 7748 for Montgomery keys,  
7896 where each party contributes one key pair all using the same EC domain parameters.

7897 It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

7898 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
7899 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
7900 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
7901 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
7902 type must be specified in the template.

7903 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
7904 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
7905 default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
7906 will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
7907 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
7908 **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
7909 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
7910 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
7911 value from its **CKA\_EXTRACTABLE** attribute.

7915 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7916 specify the minimum and maximum supported number of bits in the field sizes, respectively. For  
7917 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between  $2^{200}$   
7918 and  $2^{300}$  elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,  
7919 the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly,  $2^{300}$   
7920 is a 301-bit number).

7921 Constraints on key types are summarized in the following table:

7922 Table 78: ECDH: Allowed Key Types

Function	Key type
C_Derive	CKK_EC or CKK_EC_MONTGOMERY

### 7923 6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation

7924 The Elliptic Curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted  
7925 **CKM\_ECDH1\_COFACTOR\_DERIVE**, is a mechanism for key derivation based on the cofactor Diffie-  
7926 Hellman version of the Elliptic Curve key agreement scheme, as defined in ANSI X9.63, where each party  
7927 contributes one key pair all using the same EC domain parameters. Cofactor multiplication is  
7928 computationally efficient and helps to prevent security problems like small group attacks.

7929 It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

7930 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
7931 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
7932 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
7933 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
7934 type must be specified in the template.

7935 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

7947 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7948 specify the minimum and maximum supported number of bits in the field sizes, respectively. For  
7949 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between  $2^{200}$   
7950 and  $2^{300}$  elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,  
7951 the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly,  $2^{300}$   
7952 is a 301-bit number).

7953 Constraints on key types are summarized in the following table:

7954 *Table 79: ECDH with cofactor: Allowed Key Types*

Function	Key type
C_Derive	CKK_EC

### 7955 6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation

7956 The Elliptic Curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted  
7957 **CKM\_ECMQV\_DERIVE**, is a mechanism for key derivation based the MQV version of the Elliptic Curve  
7958 key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using  
7959 the same EC domain parameters.

7960 It has a parameter, a **CK\_ECMQV\_DERIVE\_PARAMS** structure.

7961 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
7962 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
7963 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
7964 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
7965 type must be specified in the template.

7966 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

7978 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
7979 specify the minimum and maximum supported number of bits in the field sizes, respectively. For  
7980 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between  $2^{200}$   
7981 and  $2^{300}$  elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,

7982 the number  $2^{200}$  consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly,  $2^{300}$   
7983 is a 301-bit number).

7984 Constraints on key types are summarized in the following table:

7985 *Table 80: ECDH MQV: Allowed Key Types*

Function	Key type
C_Derive	CKK_EC

### 7986 6.3.20 ECDH AES KEY WRAP

7987 The ECDH AES KEY WRAP mechanism, denoted **CKM\_ECDH\_AES\_KEY\_WRAP**, is a mechanism  
7988 based on Elliptic Curve public-key crypto-system and the AES key wrap mechanism. It supports single-  
7989 part key wrapping; and key unwrapping.

7990 It has a parameter, a **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** structure.

7991

7992 The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC  
7993 key.

- 7994 - A temporary AES key is derived from a temporary EC key and the wrapping EC key  
7995 using the **CKM\_ECDH1\_DERIVE** mechanism.  
7996 - The derived AES key is used for wrapping the target key using the  
7997 **CKM\_AES\_KEY\_WRAP\_KWP** mechanism.

7998

7999 For wrapping, the mechanism -

- 8000 • Generates a temporary random EC key (transport key) having the same parameters as the  
8001 wrapping EC key (and domain parameters). Saves the transport key public key material.  
8002 • Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen  
8003 and pSharedData using the private key of the transport EC key and the public key of wrapping EC  
8004 key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.  
8005 • Wraps the target key with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP**.  
8006 • Zeroizes the temporary AES key and EC transport private key.  
8007 • Concatenates public key material of the transport key and output the concatenated blob. The first  
8008 part is the public key material of the transport key and the second part is the wrapped target key.  
8009

8010 The private target key will be encoded as defined in section 6.7.  
8011

8012 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the  
8013 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.  
8014

8015 For unwrapping, the mechanism -

- 8016 • Splits the input into two parts. The first part is the public key material of the transport key and the  
8017 second part is the wrapped target key. The length of the first part is equal to the length of the public  
8018 key material of the unwrapping EC key.  
8019 *Note: since the transport key and the wrapping EC key share the same domain, the length of the*  
8020 *public key material of the transport key is the same length of the public key material of the*  
8021 *unwrapping EC key.*
- 8022 • Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen  
8023 and pSharedData using the private part of unwrapping EC key and the public part of the transport  
8024 EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.

- 8025     • Un-wraps the target key from the second part with the temporary AES key using  
 8026       **CKM\_AES\_KEY\_WRAP\_KWP**.  
 8027     • Zeroizes the temporary AES key.

8028

8029 *Table 81, CKM\_ECDH\_AES\_KEY\_WRAP Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ECDH_AES_KEY_WRAP						✓	

<sup>1</sup>SR = SignRecover, VR = VerifyRecover

8030

8031 Constraints on key types are summarized in the following table:

8032 *Table 82: ECDH AES Key Wrap: Allowed Key Types*

Function	Key type
C_Wrap /	CKK_EC or CKK_EC_MONTGOMERY
C_Unwrap	

### 8033 6.3.21 ECDH AES KEY WRAP mechanism parameters

8034 ♦ **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS; CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR**

8035 **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the  
 8036 **CKM\_ECDH\_AES\_KEY\_WRAP** mechanism. It is defined as follows:

```
8037
8038     typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
8039         CK ULONG          ulAESKeyBits;
8040         CK EC KDF TYPE   kdf;
8041         CK ULONG          ulSharedDataLen;
8042         CK BYTE PTR      pSharedData;
8043     } CK_ECDH_AES_KEY_WRAP_PARAMS;
```

8044 The fields of the structure have the following meanings:

8045              ulAESKeyBits      length of the temporary AES key in bits. Can be only 128, 192 or  
 8046                                  256.

8047              kdf              key derivation function used on the shared secret value to generate  
 8048                                  AES key.

8049              ulSharedDataLen      the length in bytes of the shared info

8050              pSharedData      Some data shared between the two parties

8053

8054 **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR** is a pointer to a  
 8055 **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS**.

8056

8057 **6.3.22 FIPS 186-4**

8058 When CKM\_ECDSA is operated in FIPS mode, the curves SHALL either be NIST recommended curves  
8059 (with a fixed set of domain parameters) or curves with domain parameters generated as specified by  
8060 ANSI X9.64. The NIST recommended curves are:

8061  
8062 P-192, P-224, P-256, P-384, P-521  
8063 K-163, B-163, K-233, B-233  
8064 K-283, B-283, K-409, B-409  
8065 K-571, B-571

8066 **6.4 Diffie-Hellman**

8067 *Table 83, Diffie-Hellman Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_PARAMETER_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_X9_42_DH_KEY_PAIR_GEN					✓		
CKM_X9_42_DH_PARAMETER_GEN					✓		
CKM_X9_42_DH_DERIVE							✓
CKM_X9_42_DH_HYBRID_DERIVE							✓
CKM_X9_42_MQV_DERIVE							✓

8068 **6.4.1 Definitions**

8069 This section defines the key type "CKK\_DH" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE  
8070 attribute of [DH] key objects.

8071 Mechanisms:

8072 CKM\_DH\_PKCS\_KEY\_PAIR\_GEN  
8073 CKM\_DH\_PKCS\_PARAMETER\_GEN  
8074 CKM\_DH\_PKCS\_DERIVE  
8075 CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN  
8076 CKM\_X9\_42\_DH\_PARAMETER\_GEN  
8077 CKM\_X9\_42\_DH\_DERIVE  
8078 CKM\_X9\_42\_DH\_HYBRID\_DERIVE  
8079 CKM\_X9\_42\_MQV\_DERIVE

8080

## 8081 6.4.2 Diffie-Hellman public key objects

8082 Diffie-Hellman public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_DH**) hold Diffie-  
8083 Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in  
8084 addition to the common attributes defined for this object class:

8085 *Table 84, Diffie-Hellman Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,3</sup>	Big integer	Prime $p$
CKA_BASE <sup>1,3</sup>	Big integer	Base $g$
CKA_VALUE <sup>1,4</sup>	Big integer	Public value $y$

8086 Refer to Table 11 for footnotes

8087 The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain  
8088 parameters”. Depending on the token, there may be limits on the length of the key components. See  
8089 PKCS #3 for more information on Diffie-Hellman keys.

8090 The following is a sample template for creating a Diffie-Hellman public key object:

```
8091 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
8092 CK_KEY_TYPE keyType = CKK_DH;
8093 CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
8094 CK_BYTE prime[] = {...};
8095 CK_BYTE base[] = {...};
8096 CK_BYTE value[] = {...};
8097 CK_BBOOL true = CK_TRUE;
8098 CK_ATTRIBUTE template[] = {
8099     {CKA_CLASS, &class, sizeof(class)},
8100     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8101     {CKA_TOKEN, &true, sizeof(true)},
8102     {CKA_LABEL, label, sizeof(label)-1},
8103     {CKA_PRIME, prime, sizeof(prime)},
8104     {CKA_BASE, base, sizeof(base)},
8105     {CKA_VALUE, value, sizeof(value)}}
8106 };
```

## 8107 6.4.3 X9.42 Diffie-Hellman public key objects

8108 X9.42 Diffie-Hellman public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_X9\_42\_DH**)  
8109 hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key  
8110 object attributes, in addition to the common attributes defined for this object class:

8111 *Table 85, X9.42 Diffie-Hellman Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,3</sup>	Big integer	Prime $p$ ( $\geq 1024$ bits, in steps of 256 bits)
CKA_BASE <sup>1,3</sup>	Big integer	Base $g$
CKA_SUBPRIME <sup>1,3</sup>	Big integer	Subprime $q$ ( $\geq 160$ bits)
CKA_VALUE <sup>1,4</sup>	Big integer	Public value $y$

8112 Refer to Table 11 for footnotes

8113 The **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-  
8114 Hellman domain parameters”. See the ANSI X9.42 standard for more information on X9.42 Diffie-  
8115 Hellman keys.

8116 The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

```

8117     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
8118     CK_KEY_TYPE keyType = CKK_X9_42_DH;
8119     CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key
8120         object";
8121     CK_BYTE prime[] = {...};
8122     CK_BYTE base[] = {...};
8123     CK_BYTE subprime[] = {...};
8124     CK_BYTE value[] = {...};
8125     CK_BBOOL true = CK_TRUE;
8126     CK_ATTRIBUTE template[] = {
8127         {CKA_CLASS, &class, sizeof(class)},
8128         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8129         {CKA_TOKEN, &true, sizeof(true)},
8130         {CKA_LABEL, label, sizeof(label)-1},
8131         {CKA_PRIME, prime, sizeof(prime)},
8132         {CKA_BASE, base, sizeof(base)},
8133         {CKA_SUBPRIME, subprime, sizeof(subprime)},
8134         {CKA_VALUE, value, sizeof(value)}}
8135     };

```

#### 8136 6.4.4 Diffie-Hellman private key objects

8137 Diffie-Hellman private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_DH**) hold Diffie-
8138 Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in
8139 addition to the common attributes defined for this object class:

8140 *Table 86, Diffie-Hellman Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4,6</sup>	Big integer	Prime $p$
CKA_BASE <sup>1,4,6</sup>	Big integer	Base $g$
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	Private value $x$
CKA_VALUE_BITS <sup>2,6</sup>	CK ULONG	Length in bits of private value $x$

8141 <sup>1</sup>Refer to Table 11 for footnotes

8142 The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain
8143 parameters”. Depending on the token, there may be limits on the length of the key components. See
8144 PKCS #3 for more information on Diffie-Hellman keys.

8145 Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in
8146 the key's template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-
8147 Hellman key pair, and the Diffie-Hellman parameters for the pair are specified in the template for the
8148 Diffie-Hellman public key.

8149 The following is a sample template for creating a Diffie-Hellman private key object:

```

8150     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8151     CK_KEY_TYPE keyType = CKK_DH;
8152     CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
8153     CK_BYTE subject[] = {...};
8154     CK_BYTE id[] = {123};
8155     CK_BYTE prime[] = {...};
8156     CK_BYTE base[] = {...};

```

```

8157     CK_BYTE value[] = { ... };
8158     CK_BBOOL true = CK_TRUE;
8159     CK_ATTRIBUTE template[] = {
8160         {CKA_CLASS, &class, sizeof(class)},
8161         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8162         {CKA_TOKEN, &true, sizeof(true)},
8163         {CKA_LABEL, label, sizeof(label)-1},
8164         {CKA SUBJECT, subject, sizeof(subject)},
8165         {CKA_ID, id, sizeof(id)},
8166         {CKA_SENSITIVE, &true, sizeof(true)},
8167         {CKA_DERIVE, &true, sizeof(true)},
8168         {CKA_PRIME, prime, sizeof(prime)},
8169         {CKA_BASE, base, sizeof(base)},
8170         {CKA_VALUE, value, sizeof(value)}
8171     };

```

#### 8172 6.4.5 X9.42 Diffie-Hellman private key objects

8173 X9.42 Diffie-Hellman private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_X9\_42\_DH**)  
8174 hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key  
8175 object attributes, in addition to the common attributes defined for this object class:

8176 *Table 87, X9.42 Diffie-Hellman Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4,6</sup>	Big integer	Prime $p$ ( $\geq 1024$ bits, in steps of 256 bits)
CKA_BASE <sup>1,4,6</sup>	Big integer	Base $g$
CKA_SUBPRIME <sup>1,4,6</sup>	Big integer	Subprime $q$ ( $\geq 160$ bits)
CKA_VALUE <sup>1,4,6,7</sup>	Big integer	Private value $x$

8177 Refer to Table 11 for footnotes

8178 The **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-  
8179 Hellman domain parameters”. Depending on the token, there may be limits on the length of the key  
8180 components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

8181 Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain  
8182 parameters are *not* specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are  
8183 only generated as part of a X9.42 Diffie-Hellman key pair, and the X9.42 Diffie-Hellman domain  
8184 parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

8185 The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

```

8186     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8187     CK_KEY_TYPE keyType = CKK_X9_42_DH;
8188     CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman private key object";
8189     CK_BYTE subject[] = { ... };
8190     CK_BYTE id[] = {123};
8191     CK_BYTE prime[] = { ... };
8192     CK_BYTE base[] = { ... };
8193     CK_BYTE subprime[] = { ... };
8194     CK_BYTE value[] = { ... };
8195     CK_BBOOL true = CK_TRUE;
8196     CK_ATTRIBUTE template[] = {
8197         {CKA_CLASS, &class, sizeof(class)},

```

```

8198     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
8199     { CKA_TOKEN, &true, sizeof(true) },
8200     { CKA_LABEL, label, sizeof(label)-1 },
8201     { CKA SUBJECT, subject, sizeof(subject) },
8202     { CKA_ID, id, sizeof(id) },
8203     { CKA_SENSITIVE, &true, sizeof(true) },
8204     { CKA_DERIVE, &true, sizeof(true) },
8205     { CKA_PRIME, prime, sizeof(prime) },
8206     { CKA_BASE, base, sizeof(base) },
8207     { CKA_SUBPRIME, subprime, sizeof(subprime) },
8208     { CKA_VALUE, value, sizeof(value) }
8209 };

```

## 8210 6.4.6 Diffie-Hellman domain parameter objects

8211 Diffie-Hellman domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type  
8212 **CKK\_DH**) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain  
8213 parameter object attributes, in addition to the common attributes defined for this object class:

8214 *Table 88, Diffie-Hellman Domain Parameter Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4</sup>	Big integer	Prime $p$
CKA_BASE <sup>1,4</sup>	Big integer	Base $g$
CKA_PRIME_BITS <sup>2,3</sup>	CK ULONG	Length of the prime value.

8215 <sup>1</sup> Refer to Table 11 for footnotes

8216 The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain  
8217 parameters”. Depending on the token, there may be limits on the length of the key components. See  
8218 PKCS #3 for more information on Diffie-Hellman domain parameters.

8219 The following is a sample template for creating a Diffie-Hellman domain parameter object:

```

8220     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8221     CK_KEY_TYPE keyType = CKK_DH;
8222     CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters
8223         object";
8224     CK_BYTE prime[] = {...};
8225     CK_BYTE base[] = {...};
8226     CK_BBOOL true = CK_TRUE;
8227     CK_ATTRIBUTE template[] = {
8228         { CKA_CLASS, &class, sizeof(class) },
8229         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
8230         { CKA_TOKEN, &true, sizeof(true) },
8231         { CKA_LABEL, label, sizeof(label)-1 },
8232         { CKA_PRIME, prime, sizeof(prime) },
8233         { CKA_BASE, base, sizeof(base) },
8234     };

```

## 8235 6.4.7 X9.42 Diffie-Hellman domain parameters objects

8236 X9.42 Diffie-Hellman domain parameters objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type  
8237 **CKK\_X9\_42\_DH**) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42

8238 Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this  
8239 object class:

8240 *Table 89, X9.42 Diffie-Hellman Domain Parameters Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME <sup>1,4</sup>	Big integer	Prime $p$ ( $\geq 1024$ bits, in steps of 256 bits)
CKA_BASE <sup>1,4</sup>	Big integer	Base $g$
CKA_SUBPRIME <sup>1,4</sup>	Big integer	Subprime $q$ ( $\geq 160$ bits)
CKA_PRIME_BITS <sup>2,3</sup>	CK ULONG	Length of the prime value.
CKA_SUBPRIME_BITS <sup>2,3</sup>	CK ULONG	Length of the subprime value.

8241 Refer to Table 11 for footnotes

8242 The **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-  
8243 Hellman domain parameters”. Depending on the token, there may be limits on the length of the domain  
8244 parameters components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman  
8245 domain parameters.

8246 The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

```
8247 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8248 CK_KEY_TYPE keyType = CKK_X9_42_DH;
8249 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain
8250     parameters object";
8251 CK_BYTE prime[] = {...};
8252 CK_BYTE base[] = {...};
8253 CK_BYTE subprime[] = {...};
8254 CK_BBOOL true = CK_TRUE;
8255 CK_ATTRIBUTE template[] = {
8256     {CKA_CLASS, &class, sizeof(class)},
8257     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8258     {CKA_TOKEN, &true, sizeof(true)},
8259     {CKA_LABEL, label, sizeof(label)-1},
8260     {CKA_PRIME, prime, sizeof(prime)},
8261     {CKA_BASE, base, sizeof(base)},
8262     {CKA_SUBPRIME, subprime, sizeof(subprime)},
8263 };
```

#### 8264 **6.4.8 PKCS #3 Diffie-Hellman key pair generation**

8265 The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted  
8266 **CKM\_DH\_PKCS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on Diffie-Hellman key  
8267 agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase I”. It does not have a parameter.

8268 The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as  
8269 specified in the **CKA\_PRIME** and **CKA\_BASE** attributes of the template for the public key. If the  
8270 **CKA\_VALUE\_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the  
8271 private value, as described in PKCS #3.

8272 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
8273 public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**, and **CKA\_VALUE** (and  
8274 the **CKA\_VALUE\_BITS** attribute, if it is not already provided in the template) attributes to the new private  
8275 key; other attributes required by the Diffie-Hellman public and private key types must be specified in the  
8276 templates.

8277 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8278 specify the supported range of Diffie-Hellman prime sizes, in bits.

## 8279 6.4.9 PKCS #3 Diffie-Hellman domain parameter generation

8280 The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted  
8281 **CKM\_DH\_PKCS\_PARAMETER\_GEN**, is a domain parameter generation mechanism based on Diffie-  
8282 Hellman key agreement, as defined in PKCS #3.

8283 It does not have a parameter.

8284 The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as  
8285 specified in the **CKA\_PRIME\_BITS** attribute of the template.

8286 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**, and  
8287 **CKA\_PRIME\_BITS** attributes to the new object. Other attributes supported by the Diffie-Hellman domain  
8288 parameter types may also be specified in the template, or else are assigned default initial values.

8289 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8290 specify the supported range of Diffie-Hellman prime sizes, in bits.

## 8291 6.4.10 PKCS #3 Diffie-Hellman key derivation

8292 The PKCS #3 Diffie-Hellman key derivation mechanism, denoted **CKM\_DH\_PKCS\_DERIVE**, is a  
8293 mechanism for key derivation based on Diffie-Hellman key agreement, as defined in PKCS #3. This is  
8294 what PKCS #3 calls “phase II”.

8295 It has a parameter, which is the public value of the other party in the key agreement protocol, represented  
8296 as a Cryptoki “Big integer” (*i.e.*, a sequence of bytes, most-significant byte first).

8297 This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other  
8298 party. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS  
8299 #3, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one  
8300 and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes  
8301 bytes from the leading end of the secret value.) The mechanism contributes the result as the  
8302 **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the  
8303 template.

8304 This mechanism has the following rules about key sensitivity and extractability<sup>2</sup>:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
8306 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
8307 default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
8309 will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
8310 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
8311 **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
8313 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
8314 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
8315 value from its **CKA\_EXTRACTABLE** attribute.

8316 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8317 specify the supported range of Diffie-Hellman prime sizes, in bits.

---

2 Note that the rules regarding the **CKA\_SENSITIVE**, **CKA\_EXTRACTABLE**, **CKA\_ALWAYS\_SENSITIVE**, and **CKA\_NEVER\_EXTRACTABLE** attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as **CKM\_SSL3\_MASTER\_KEY\_DERIVE**.

8318 **6.4.11 X9.42 Diffie-Hellman mechanism parameters**

- 8319 ◆ **CK\_X9\_42\_DH\_KDF\_TYPE, CK\_X9\_42\_DH\_KDF\_TYPE\_PTR**

8320 **CK\_X9\_42\_DH\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive  
8321 keying data from a shared secret. The key derivation function will be used by the X9.42 Diffie-Hellman  
8322 key agreement schemes. It is defined as follows:

8323     **typedef CK ULONG CK\_X9\_42\_DH\_KDF\_TYPE;**

8325 The following table lists the defined functions.

8326 *Table 90, X9.42 Diffie-Hellman Key Derivation Functions*

Source Identifier
<b>CKD_NULL</b>
<b>CKD_SHA1_KDF ASN1</b>
<b>CKD_SHA1_KDF CONCATENATE</b>

8327 The key derivation function **CKD\_NULL** produces a raw shared secret value without applying any key  
8328 derivation function whereas the key derivation functions **CKD\_SHA1\_KDF ASN1** and  
8329 **CKD\_SHA1\_KDF CONCATENATE**, which are both based on SHA-1, derive keying data from the  
8330 shared secret value as defined in the ANSI X9.42 standard.

8331 **CK\_X9\_42\_DH\_KDF\_TYPE\_PTR** is a pointer to a **CK\_X9\_42\_DH\_KDF\_TYPE**.

- 8332 ◆ **CK\_X9\_42\_DH1\_DERIVE\_PARAMS, CK\_X9\_42\_DH1\_DERIVE\_PARAMS\_PTR**

8333 **CK\_X9\_42\_DH1\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
8334 **CKM\_X9\_42\_DH\_DERIVE** key derivation mechanism, where each party contributes one key pair. The  
8335 structure is defined as follows:

8336     **typedef struct CK\_X9\_42\_DH1\_DERIVE\_PARAMS {**  
8337         **CK\_X9\_42\_DH\_KDF\_TYPE kdf;**  
8338         **CK ULONG ulOtherInfoLen;**  
8339         **CK\_BYTE\_PTR pOtherInfo;**  
8340         **CK ULONG ulPublicDataLen;**  
8341         **CK\_BYTE\_PTR pPublicData;**  
8342     **} CK\_X9\_42\_DH1\_DERIVE\_PARAMS;**

8343 The fields of the structure have the following meanings:

<b>kdf</b>	key derivation function used on the shared secret value
<b>ulOtherInfoLen</b>	the length in bytes of the other info
<b>pOtherInfo</b>	some data shared between the two parties
<b>ulPublicDataLen</b>	the length in bytes of the other party's X9.42 Diffie-Hellman public key
<b>pPublicData</b>	pointer to other party's X9.42 Diffie-Hellman public key value

8351 With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.  
8352 With the key derivation function **CKD\_SHA1\_KDF ASN1**, *pOtherInfo* must be supplied, which contains  
8353 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by  
8354 the two parties intending to share the shared secret. With the key derivation function  
8355 **CKD\_SHA1\_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some

8356 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be  
8357 NULL and *ulOtherInfoLen* must be zero.  
8358 **CK\_X9\_42\_DH1\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_DH1\_DERIVE\_PARAMS**.

8359 • **CK\_X9\_42\_DH2\_DERIVE\_PARAMS, CK\_X9\_42\_DH2\_DERIVE\_PARAMS\_PTR**

8360 **CK\_X9\_42\_DH2\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
8361 **CKM\_X9\_42\_DH\_HYBRID\_DERIVE** and **CKM\_X9\_42\_MQV\_DERIVE** key derivation mechanisms,  
8362 where each party contributes two key pairs. The structure is defined as follows:

```
8363     typedef struct CK_X9_42_DH2_DERIVE_PARAMS {  
8364         CK_X9_42_DH_KDF_TYPE kdf;  
8365         CK ULONG ulOtherInfoLen;  
8366         CK BYTE PTR pOtherInfo;  
8367         CK ULONG ulPublicDataLen;  
8368         CK BYTE PTR pPublicData;  
8369         CK ULONG ulPrivateDataLen;  
8370         CK OBJECT_HANDLE hPrivateKey;  
8371         CK ULONG ulPublicDataLen2;  
8372         CK BYTE PTR pPublicData2;  
8373     } CK_X9_42_DH2_DERIVE_PARAMS;
```

8374

8375 The fields of the structure have the following meanings:

8376	kdf	key derivation function used on the shared secret value
8377	ulOtherInfoLen	the length in bytes of the other info
8378	pOtherInfo	some data shared between the two parties
8379	ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman 8380 public key
8381	pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
8382	ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
8383	hPrivateKey	key handle for second X9.42 Diffie-Hellman private key value
8384	ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman 8385 public key
8386	pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key 8387 value

8388 With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.  
8389 With the key derivation function **CKD\_SHA1\_KDF ASN1**, *pOtherInfo* must be supplied, which contains  
8390 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by  
8391 the two parties intending to share the shared secret. With the key derivation function  
8392 **CKD\_SHA1\_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some  
8393 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be  
8394 NULL and *ulOtherInfoLen* must be zero.

8395 **CK\_X9\_42\_DH2\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_DH2\_DERIVE\_PARAMS**.

8396 • **CK\_X9\_42\_MQV\_DERIVE\_PARAMS, CK\_X9\_42\_MQV\_DERIVE\_PARAMS\_PTR**

8397 **CK\_X9\_42\_MQV\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
8398 **CKM\_X9\_42\_MQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The  
8399 structure is defined as follows:

```
8400     typedef struct CK_X9_42_MQV_DERIVE_PARAMS {  
8401         CK_X9_42_DH_KDF_TYPE    kdf;  
8402         CK ULONG                ulOtherInfoLen;  
8403         CK BYTE PTR             pOtherInfo;  
8404         CK ULONG                ulPublicDataLen;  
8405         CK BYTE PTR             pPublicData;  
8406         CK ULONG                ulPrivateDataLen;  
8407         CK OBJECT_HANDLE        hPrivateKey;  
8408         CK ULONG                ulPublicDataLen2;  
8409         CK BYTE PTR             pPublicData2;  
8410         CK OBJECT_HANDLE        publicKey;  
8411     } CK_X9_42_MQV_DERIVE_PARAMS;
```

8412 The fields of the structure have the following meanings:

8414	kdf	key derivation function used on the shared secret value
8415	ulOtherInfoLen	the length in bytes of the other info
8416	pOtherInfo	some data shared between the two parties
8417	ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman public key
8418	pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
8419	ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
8420	hPrivateKey	key handle for second X9.42 Diffie-Hellman private key value
8421	ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman public key
8422	pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key value
8423	publicKey	Handle to the first party's ephemeral public key

8427 With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.  
8428 With the key derivation function **CKD\_SHA1\_KDF ASN1**, *pOtherInfo* must be supplied, which contains  
8429 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by  
8430 the two parties intending to share the shared secret. With the key derivation function  
8431 **CKD\_SHA1\_KDF\_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some  
8432 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be  
8433 NULL and *ulOtherInfoLen* must be zero.

8434 **CK\_X9\_42\_MQV\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_MQV\_DERIVE\_PARAMS**.

8435 **6.4.12 X9.42 Diffie-Hellman key pair generation**

8436 The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN**,  
8437 is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the ANSI  
8438 X9.42 standard.

8439 It does not have a parameter.

8440 The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and  
8441 subprime, as specified in the **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attributes of the template  
8442 for the public key.  
8443 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
8444 public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**, **CKA\_SUBPRIME**, and  
8445 **CKA\_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman  
8446 public and private key types must be specified in the templates.  
8447 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8448 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

#### 8449 **6.4.13 X9.42 Diffie-Hellman domain parameter generation**

8450 The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted  
8451 **CKM\_X9\_42\_DH\_PARAMETER\_GEN**, is a domain parameters generation mechanism based on X9.42  
8452 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.  
8453 It does not have a parameter.  
8454 The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime  
8455 length in bits, as specified in the **CKA\_PRIME\_BITS** and **CKA\_SUBPRIME\_BITS** attributes of the  
8456 template for the domain parameters.  
8457 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**,  
8458 **CKA\_SUBPRIME**, **CKA\_PRIME\_BITS** and **CKA\_SUBPRIME\_BITS** attributes to the new object. Other  
8459 attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the  
8460 template for the domain parameters, or else are assigned default initial values.  
8461 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8462 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

#### 8463 **6.4.14 X9.42 Diffie-Hellman key derivation**

8464 The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM\_X9\_42\_DH\_DERIVE**, is a  
8465 mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the  
8466 ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman  
8467 domain parameters.  
8468 It has a parameter, a **CK\_X9\_42\_DH1\_DERIVE\_PARAMS** structure.  
8469 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
8470 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
8471 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
8472 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
8473 type must be specified in the template. Note that in order to validate this mechanism it may be required to  
8474 use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g.  
8475 **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

8476 This mechanism has the following rules about key sensitivity and extractability:

- 8477 • The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
8478 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
8479 default value.
- 8480 • If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
8481 will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
8482 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
8483 **CKA\_SENSITIVE** attribute.
- 8484 • Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
8485 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
8486 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
8487 value from its **CKA\_EXTRACTABLE** attribute.

8488 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8489 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

#### 8490 6.4.15 X9.42 Diffie-Hellman hybrid key derivation

8491 The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted  
8492 **CKM\_X9\_42\_DH\_HYBRID\_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman  
8493 hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two  
8494 key pair, all using the same X9.42 Diffie-Hellman domain parameters.

8495 It has a parameter, a **CK\_X9\_42\_DH2\_DERIVE\_PARAMS** structure.

8496 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
8497 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
8498 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
8499 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
8500 type must be specified in the template. Note that in order to validate this mechanism it may be required to  
8501 use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g.  
8502 **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

8503 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
**CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
value from its **CKA\_EXTRACTABLE** attribute.

8511 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8512 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

#### 8517 6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

8518 The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted  
8519 **CKM\_X9\_42\_MQV\_DERIVE**, is a mechanism for key derivation based the MQV scheme, as defined in  
8520 the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-  
8521 Hellman domain parameters.

8522 It has a parameter, a **CK\_X9\_42\_MQV\_DERIVE\_PARAMS** structure.

8523 This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE**  
8524 attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of  
8525 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism  
8526 contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key  
8527 type must be specified in the template. Note that in order to validate this mechanism it may be required to  
8528 use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g.  
8529 **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

8530 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the

8536 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
8537 **CKA\_SENSITIVE** attribute.

- 8538 • Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
8539 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
8540 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
8541 value from its **CKA\_EXTRACTABLE** attribute.

8542 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8543 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

## 8544 6.5 Extended Triple Diffie-Hellman (x3dh)

8545 The Extended Triple Diffie-Hellman mechanism described here is the one described in  
8546 [SIGNAL].

8547 8548 *Table 91, Extended Triple Diffie-Hellman Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_X3DH_INITIALIZE							✓
CKM_X3DH RESPOND							✓

### 8549 6.5.1 Definitions

8550 Mechanisms:

8551 CKM\_X3DH\_INITIALIZE

8552 CKM\_X3DH\_RESPOND

### 8553 6.5.2 Extended Triple Diffie-Hellman key objects

8554 Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation  
8555 (**CKK\_EC\_MONTGOMERY**). Three different kinds of keys are used, they differ in their lifespan:

- 8556 • identity keys are long-term keys, which identify the peer,  
8557 • prekeys are short-term keys, which should be rotated often (weekly to hourly)  
8558 • onetime prekeys are keys, which should be used only once.

8559 Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of  
8560 their:

- 8561 • public Identity key,  
8562 • current prekey, signed using XEDDSA with their identity key  
8563 • optionally a batch of One-time public keys.

### 8564 6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

8565 Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public  
8566 keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and  
8567 optionally one One-time public key.

8568 When the necessary key material is available, the initiating party calls CKM\_X3DH\_INITIALIZE, also  
8569 providing the following additional parameters:

- 8570 • the initiators identity key

- 8571       • the initiators ephemeral key (a fresh, one-time **CKK\_EC\_MONTGOMERY** type key)

8572  
 8573 **CK\_X3DH\_INITIATE\_PARAMS** is a structure that provides the parameters to the  
 8574 **CKM\_X3DH\_INITIALIZE** key exchange mechanism. The structure is defined as follows:

```
8575     typedef struct CK_X3DH_INITIATE_PARAMS {
8576         CK_X3DH_KDF_TYPE    kdf;
8577         CK_OBJECT_HANDLE   pPeer_identity;
8578         CK_OBJECT_HANDLE   pPeer_prekey;
8579         CK_BYTE_PTR        pPrekey_signature;
8580         CK_BYTE_PTR        pOnetime_key;
8581         CK_OBJECT_HANDLE   pOwn_identity;
8582         CK_OBJECT_HANDLE   pOwn_ephemeral;
8583     } CK_X3DH_INITIATE_PARAMS;
```

8584 *Table 92, Extended Triple Diffie-Hellman Initiate Message parameters:*

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	<i>Key derivation function</i>
pPeer_identity	Key handle	<i>Peers public Identity key (from the prekey-bundle)</i>
pPeer_prekey	Key Handle	Peers public prekey (from the prekey-bundle)
pPrekey_signature	Byte array	<i>XEDDSA signature of PEER_PREKEY (from prekey-bundle)</i>
pOnetime_key	Byte array	Optional one-time public prekey of peer (from the prekey-bundle)
pOwn_identity	Key Handle	Initiators Identity key
pOwn_ephemeral	Key Handle	Initiators ephemeral key

#### 8586 6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange

8587 Responding an Extended Triple Diffie-Hellman key exchange is done by executing a  
 8588 **CKM\_X3DH RESPOND** mechanism. **CK\_X3DH RESPOND\_PARAMS** is a structure that provides the  
 8589 parameters to the **CKM\_X3DH RESPOND** key exchange mechanism. All these parameter should be  
 8590 supplied by the Initiator in a message to the responder. The structure is defined as follows:

```
8591     typedef struct CK_X3DH_RESPOND_PARAMS {
8592         CK_X3DH_KDF_TYPE    kdf;
8593         CK_BYTE_PTR        pIdentity_id;
8594         CK_BYTE_PTR        pPrekey_id;
8595         CK_BYTE_PTR        pOnetime_id;
8596         CK_OBJECT_HANDLE   pInitiator_identity;
8597         CK_BYTE_PTR        pInitiator_ephemeral;
8598     } CK_X3DH_RESPOND_PARAMS;
```

8599  
 8600 *Table 93, Extended Triple Diffie-Hellman 1st Message parameters:*

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	<i>Key derivation function</i>
pIdentity_id	Byte array	<i>Peers public identity key identifier (from the prekey-bundle)</i>
pPrekey_id	Byte array	Peers public prekey identifier (from the prekey-bundle)
pOnetime_id	Byte array	Optional one-time public prekey of peer (from the prekey-bundle)
pInitiator_identity	Key handle	Initiators Identity key
pInitiator_ephemeral	Byte array	Initiators ephemeral key

8601

8602 Where the \*\_id fields are identifiers marking which key has been used from the prekey-bundle, these  
 8603 identifiers could be the keys themselves.

8604

8605 This mechanism has the following rules about key sensitivity and extractability<sup>3</sup>:

- 1 The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- 2 If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- 3 Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

## 8617 6.5.5 Extended Triple Diffie-Hellman parameters

- **CK\_X3DH\_KDF\_TYPE, CK\_X3DH\_KDF\_TYPE\_PTR**

8619 **CK\_X3DH\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying  
 8620 data from a shared secret. The key derivation function will be used by the X3DH key agreement  
 8621 schemes. It is defined as follows:

```
8622     typedef CK ULONG CK_X3DH_KDF_TYPE;
```

8623

8624 The following table lists the defined functions.

8625 *Table 94, X3DH: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF

---

3 Note that the rules regarding the CKA\_SENSITIVE, CKA\_EXTRACTABLE, CKA\_ALWAYS\_SENSITIVE, and CKA\_NEVER\_EXTRACTABLE attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as CKM\_SSL3\_MASTER\_KEY\_DERIVE.

CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

## 8626 6.6 Double Ratchet

8627 The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of  
 8628 short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations.  
 8629 The algorithm is described in [DoubleRatchet]. The Signal protocol uses X3DH to exchange a shared  
 8630 secret in the first step, which is then used to derive a Double Ratchet secret key.

8631 *Table 95, Double Ratchet Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_X2RATCHET_INITIALIZE							✓
CKM_X2RATCHET RESPOND							✓
CKM_X2RATCHET_ENCRYPT	✓					✓	
CKM_X2RATCHET_DECRYPT	✓					✓	

8632

### 8633 6.6.1 Definitions

8634 This section defines the key type "CKK\_X2RATCHET" for type CK\_KEY\_TYPE as used in the  
 8635 CKA\_KEY\_TYPE attribute of key objects.

8636 Mechanisms:

8637 CKM\_X2RATCHET\_INITIALIZE  
 8638 CKM\_X2RATCHET\_RESPOND  
 8639 CKM\_X2RATCHET\_ENCRYPT  
 8640 CKM\_X2RATCHET\_DECRYPT

### 8641 6.6.2 Double Ratchet secret key objects

8642 Double Ratchet secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_X2RATCHET) hold  
 8643 Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the  
 8644 mechanism CKM\_X2RATCHET\_INITIALIZE or CKM\_X2RATCHET\_RESPOND. In the Signal protocol  
 8645 these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-  
 8646 exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the  
 8647 common attributes defined for this object class:

8648 *Table 96, Double Ratchet Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_X2RATCHET_RK	Byte array	Root key
CKA_X2RATCHET_HKS	Byte array	Sender Header key
CKA_X2RATCHET_HKR	Byte array	Receiver Header key
CKA_X2RATCHET_NHKS	Byte array	Next Sender Header Key
CKA_X2RATCHET_NHKR	Byte array	Next Receiver Header Key
CKA_X2RATCHET_CKS	Byte array	Sender Chain key

Attribute	Data type	Meaning
CKA_X2RATCHET_CKR	Byte array	Receiver Chain key
CKA_X2RATCHET_DHS	Byte array	Sender DH secret key
CKA_X2RATCHET_DHP	Byte array	Sender DH public key
CKA_X2RATCHET_DHR	Byte array	Receiver DH public key
CKA_X2RATCHET_NS	ULONG	Message number send
CKA_X2RATCHET_NR	ULONG	Message number receive
CKA_X2RATCHET_PNS	ULONG	Previous message number send
CKA_X2RATCHET_BOBS1STMSG	BOOL	Is this bob and has he ever sent a message?
CKA_X2RATCHET_ISALICE	BOOL	Is this Alice?
CKA_X2RATCHET_BAGSIZE	ULONG	How many out-of-order keys do we store
CKA_X2RATCHET_BAG	Byte array	Out-of-order keys

### 8649 6.6.3 Double Ratchet key derivation

8650 The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the  
 8651 receiving, denoted **CKM\_X2RATCHET\_INITIALIZE** and **CKM\_X2RATCHET RESPOND**, are the key  
 8652 derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by  
 8653 executing a X3DH key exchange.

8654 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 8655 key. Additionally the attribute flags indicating which functions the key supports are also contributed by the  
 8656 mechanism.

8657 For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these  
 8658 two sizes. A Cryptoki implementation may support one or both of these curves and should set the  
 8659 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

- **CK\_X2RATCHET\_INITIALIZE\_PARAMS;**  
**CK\_X2RATCHET\_INITIALIZE\_PARAMS\_PTR**

8662 **CK\_X2RATCHET\_INITIALIZE\_PARAMS** provides the parameters to the  
 8663 **CKM\_X2RATCHET\_INITIALIZE** mechanism. It is defined as follows:

```
8664     typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
8665         CK_BYTE_PTR                      sk;
8666         CK_OBJECT_HANDLE                  peer_public_prekey;
8667         CK_OBJECT_HANDLE                  peer_public_identity;
8668         CK_OBJECT_HANDLE                  own_public_identity;
8669         CK_BBOOL                          bEncryptedHeader;
8670         CK ULONG                         eCurve;
8671         CK_MECHANISM_TYPE                aeadMechanism;
8672         CK_X2RATCHET_KDF_TYPE            kdfMechanism;
8673     } CK_X2RATCHET_INITIALIZE_PARAMS;
```

8674  
 8675 The fields of the structure have the following meanings:

8676 <i>sk</i>	<i>the shared secret with peer (derived using X3DH)</i>
8677 <i>peers_public_prekey</i>	<i>Peers public prekey which the Initiator used in the X3DH</i>
8678 <i>peers_public_identity</i>	<i>Peers public identity which the Initiator used in the X3DH</i>

8679        *own\_public\_identity*     *Initiators public identity as used in the X3DH*  
 8680        *bEncryptedHeader*     *whether the headers are encrypted*  
 8681              *eCurve*     *255 for curve 25519 or 448 for curve 448*  
 8682        *aeadMechanism*     *a mechanism supporting AEAD encryption*  
 8683        *kdfMechanism*     *a Key Derivation Mechanism, such as*  
 8684              *CKD\_BLAKE2B\_512\_KDF*

- 8685 • **CK\_X2RATCHET\_RESPOND\_PARAMS;**  
 8686        **CK\_X2RATCHET\_RESPOND\_PARAMS\_PTR**

8687 **CK\_X2RATCHET\_RESPOND\_PARAMS** provides the parameters to the  
 8688 **CKM\_X2RATCHET\_RESPOND** mechanism. It is defined as follows:

```

8689     typedef struct CK_X2RATCHET_RESPOND_PARAMS {
8690         CK_BYTE_PTR           sk;
8691         CK_OBJECT_HANDLE      own_prekey;
8692         CK_OBJECT_HANDLE      initiator_identity;
8693         CK_OBJECT_HANDLE      own_public_identity;
8694         CK_BBOOL                bEncryptedHeader;
8695         CK ULONG                 eCurve;
8696         CK_MECHANISM_TYPE      aeadMechanism;
8697         CK_X2RATCHET_KDF_TYPE   kdfMechanism;
8698     } CK_X2RATCHET_RESPOND_PARAMS;
  
```

8699 The fields of the structure have the following meanings:

8700 <i>sk</i> 8701 <i>own_prekey</i> 8702 <i>initiator_identity</i> 8703 <i>own_public_identity</i> 8704 <i>bEncryptedHeader</i> 8705 <i>eCurve</i> 8706 <i>aeadMechanism</i> 8707 <i>kdfMechanism</i>	<i>shared secret with the Initiator</i> <i>Own Prekey pair that the Initiator used</i> <i>Initiators public identity key used</i> <i>as used in the prekey bundle by the initiator in the X3DH</i> <i>whether the headers are encrypted</i> <i>255 for curve 25519 or 448 for curve 448</i> <i>a mechanism supporting AEAD encryption</i> <i>a Key Derivation Mechanism, such as</i> <i>CKD_BLAKE2B_512_KDF</i>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### 8710 **6.6.4 Double Ratchet Encryption mechanism**

8711 The Double Ratchet encryption mechanism, denoted **CKM\_X2RATCHET\_ENCRYPT** and  
 8712 **CKM\_X2RATCHET\_DECRYPT**, are a mechanisms for single part encryption and decryption based on  
 8713 the Double Ratchet and its underlying AEAD cipher.

8714 **6.6.5 Double Ratchet parameters**

8715 • **CK\_X2RATCHET\_KDF\_TYPE, CK\_X2RATCHET\_KDF\_TYPE\_PTR**

8716 **CK\_X2RATCHET\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive  
8717 keying data from a shared secret. The key derivation function will be used by the X key derivation  
8718 scheme. It is defined as follows:

8719     **typedef CK ULONG CK\_X2RATCHET\_KDF\_TYPE;**

8720

8721 The following table lists the defined functions.

8722 *Table 97, X2RATCHET: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF
CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

8723

8724 **6.7 Wrapping/unwrapping private keys**

8725 Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private  
8726 keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, short Weierstrass EC private keys  
8727 and DSA private keys. For wrapping, a private key is BER-encoded according to PKCS #8's  
8728 PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the private key. The  
8729 object identifiers for the required algorithm identifiers are as follows:

```
8730     rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
8731
8732     dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }
8733
8734     dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)
8735                                 us(840) ansi-x942(10046) number-type(2) 1 }
8736
8737     id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
8738                                 us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
8739
8740     id-dsa OBJECT IDENTIFIER ::= {
8741         iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }
8742
8743     where
8744     pkcs-1 OBJECT IDENTIFIER ::= {
8745         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }
8746
8747     pkcs-3 OBJECT IDENTIFIER ::= {
8748         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }
```

```

8750      These parameters for the algorithm identifiers have the
8751          following types, respectively:
8752          NULL
8753
8754          DHParameter ::= SEQUENCE {
8755              prime                  INTEGER,    -- p
8756              base                   INTEGER,    -- g
8757              privateValueLength     INTEGER OPTIONAL
8758          }
8759
8760          DomainParameters ::= SEQUENCE {
8761              prime                  INTEGER,    -- p
8762              base                   INTEGER,    -- g
8763              subprime                INTEGER,    -- q
8764              cofactor                INTEGER OPTIONAL, -- j
8765              validationParms         ValidationParms OPTIONAL
8766          }
8767
8768          ValidationParms ::= SEQUENCE {
8769              Seed                   BIT STRING, -- seed
8770              PGenCounter            INTEGER      -- parameter verification
8771          }
8772
8773          Parameters ::= CHOICE {
8774              ecParameters           ECParameters,
8775              namedCurve             CURVES.&id({CurveNames}),
8776              implicitlyCA          NULL
8777          }
8778
8779          Dss_Parms ::= SEQUENCE {
8780              p INTEGER,
8781              q INTEGER,
8782              g INTEGER
8783          }
8784
8785      For the X9.42 Diffie-Hellman domain parameters, the cofactor and the validationParms optional fields
8786      should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values
8787      are not stored within the token.
8788      For the EC domain parameters, the use of namedCurve is recommended over the choice
8789      ecParameters. The choice implicitlyCA must not be used in Cryptoki.
8790      Within the PrivateKeyInfo type:
8791      • RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type
8792          requires values to be present for all the attributes specific to Cryptoki's RSA private key objects. In
8793          other words, if a Cryptoki library does not have values for an RSA private key's CKA_MODULUS,
8794          CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2,
8795          CKA_EXPONENT_1, CKA_EXPONENT_2, and CKA_COEFFICIENT values, it must not create an
8796          RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
8797      • Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.

```

- 8798 • X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.  
 8799 • Short Weierstrass EC private keys are BER-encoded according to SEC G SEC 1 ECPrivateKey  
 8800 ASN.1 type:

```
8801   ECPrivateKey ::= SEQUENCE {
 8802     Version      INTEGER { ecPrivkeyVer1(1) }
 8803     (ecPrivkeyVer1),
 8804     privateKey   OCTET STRING,
 8805     parameters   [0] Parameters OPTIONAL,
 8806     publicKey    [1] BIT STRING OPTIONAL
 8807   }
```

8808  
 8809 Since the EC domain parameters are placed in the PKCS #8's privateKeyAlgorithm field, the optional  
 8810 **parameters** field in an ECPrivateKey must be omitted. A Cryptoki application must be able to  
 8811 unwrap an ECPrivateKey that contains the optional **publicKey** field; however, what is done with this  
 8812 **publicKey** field is outside the scope of Cryptoki.

- 8813 • DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

8814 Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is  
 8815 encrypted with the secret key. This encryption is defined in the section for the respective key wrapping  
 8816 mechanism.

8817 Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined  
 8818 for the respective key unwrapping mechanism. The data thereby obtained are parsed as a  
 8819 PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the  
 8820 decrypted data does not parse properly, or its type does not match the key type specified in the template  
 8821 for the new key. The unwrapping mechanism contributes only those attributes specified in the  
 8822 PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will  
 8823 take their default values.

8824 Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

```
8825   DSA OBJECT IDENTIFIER ::= { algorithm 12 }
 8826   algorithm OBJECT IDENTIFIER ::= {
 8827     iso(1) identifier-organization(3) oiw(14) secsig(3)
 8828       algorithm(2) }
```

8829 with associated parameters

```
8831   DSAParameters ::= SEQUENCE {
 8832     prime1 INTEGER, -- modulus p
 8833     prime2 INTEGER, -- modulus q
 8834     base INTEGER -- base g
 8835   }
```

8836  
 8837 for wrapping DSA private keys. Note that although the two structures for holding DSA domain  
 8838 parameters appear identical when instances of them are encoded, the two corresponding object  
 8839 identifiers are different.

## 8840 6.8 Generic secret key

8841 *Table 98, Generic Secret Key Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GENERIC_SECRET_KEY_GEN					✓		

## 6.8.1 Definitions

This section defines the key type "CKK\_GENERIC\_SECRET" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_GENERIC\_SECRET\_KEY\_GEN

## 6.8.2 Generic secret key objects

Generic secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_GENERIC\_SECRET) hold generic secret keys. These keys do not support encryption or decryption; however, other keys can be derived from them and they can be used in HMAC operations. The following table defines the generic secret key object attributes, in addition to the common attributes defined for this object class:

These key types are used in several of the mechanisms described in this section.

*Table 99, Generic Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (arbitrary length)
CKA_VALUE_LEN <sup>2,3</sup>	CK ULONG	Length in bytes of key value

Refer to Table 11 for footnotes

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

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
CK_UTF8CHAR label[] = "A generic secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}}
};
```

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the generic secret key object's CKA\_VALUE attribute.

### 8872 6.8.3 Generic secret key generation

8873 The generic secret key generation mechanism, denoted **CKM\_GENERIC\_SECRET\_KEY\_GEN**, is used  
8874 to generate generic secret keys. The generated keys take on any attributes provided in the template  
8875 passed to the **C\_GenerateKey** call, and the **CKA\_VALUE\_LEN** attribute specifies the length of the key  
8876 to be generated.

8877 It does not have a parameter.

8878 The template supplied must specify a value for the **CKA\_VALUE\_LEN** attribute. If the template specifies  
8879 an object type and a class, they must have the following values:

```
8880     CK_OBJECT_CLASS = CKO_SECRET_KEY;  
8881     CK_KEY_TYPE = CKK_GENERIC_SECRET;
```

8882 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8883 specify the supported range of key sizes, in bits.

## 8884 6.9 HMAC mechanisms

8885 Refer to **RFC2104** and **FIPS 198** for HMAC algorithm description. The HMAC secret key shall correspond  
8886 to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition).  
8887 Such keys, for use with HMAC operations can be created using **C\_CreateObject** or **C\_GenerateKey**.

8888 The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in  
8889 the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

### 8890 6.9.1 General block cipher mechanism parameters

#### 8891 • **CK\_MAC\_GENERAL\_PARAMS; CK\_MAC\_GENERAL\_PARAMS\_PTR**

8892 **CK\_MAC\_GENERAL\_PARAMS** provides the parameters to the general-length MACing mechanisms of  
8893 the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers. It also provides the parameters to  
8894 the general-length HMACing mechanisms (i.e., SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T  
8895 family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1). It holds the length of the MAC  
8896 that these mechanisms produce. It is defined as follows:

```
8897     typedef CK ULONG CK_MAC_GENERAL_PARAMS;
```

8899 **CK\_MAC\_GENERAL\_PARAMS\_PTR** is a pointer to a **CK\_MAC\_GENERAL\_PARAMS**.

## 8900 6.10 AES

8901 For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

8902 *Table 100, AES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_KEY_GEN					✓		
CKM_AES_ECB	✓					✓	
CKM_AES_CBC	✓					✓	
CKM_AES_CBC_PAD	✓					✓	
CKM_AES_MAC_GENERAL		✓					

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_MAC		✓					
CKM_AES_OFB	✓					✓	
CKM_AES_CFB64	✓					✓	
CKM_AES_CFB8	✓					✓	
CKM_AES_CFB128	✓					✓	
CKM_AES_CFB1	✓					✓	
CKM_AES_XCBC_MAC		✓					
CKM_AES_XCBC_MAC_96		✓					

## 8903 6.10.1 Definitions

8904 This section defines the key type “CKK\_AES” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE  
 8905 attribute of key objects.

8906 Mechanisms:

8907 CKM\_AES\_KEY\_GEN  
 8908 CKM\_AES\_ECB  
 8909 CKM\_AES\_CBC  
 8910 CKM\_AES\_MAC  
 8911 CKM\_AES\_MAC\_GENERAL  
 8912 CKM\_AES\_CBC\_PAD  
 8913 CKM\_AES\_OFB  
 8914 CKM\_AES\_CFB64  
 8915 CKM\_AES\_CFB8  
 8916 CKM\_AES\_CFB128  
 8917 CKM\_AES\_CFB1  
 8918 CKM\_AES\_XCBC\_MAC  
 8919 CKM\_AES\_XCBC\_MAC\_96

## 8920 6.10.2 AES secret key objects

8921 AES secret key objects (object class **CKO\_SECRET\_KEY**, key type **CKK\_AES**) hold AES keys. The  
 8922 following table defines the AES secret key object attributes, in addition to the common attributes defined  
 8923 for this object class:

8924 *Table 101, AES Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN <sup>2,3,6</sup>	CK ULONG	Length in bytes of key value

8925 - Refer to Table 11 for footnotes

8926 The following is a sample template for creating an AES secret key object:

```

8927     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
8928     CK_KEY_TYPE keyType = CKK_AES;
8929     CK_UTF8CHAR label[] = "An AES secret key object";
8930     CK_BYTE value[] = { ... };
8931     CK_BBOOL true = CK_TRUE;
8932     CK_ATTRIBUTE template[] = {
8933         {CKA_CLASS, &class, sizeof(class)},
8934         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8935         {CKA_TOKEN, &true, sizeof(true)},
8936         {CKA_LABEL, label, sizeof(label)-1},
8937         {CKA_ENCRYPT, &true, sizeof(true)},
8938         {CKA_VALUE, value, sizeof(value)}}
8939     } ;
8940
8941 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
8942 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
8943 the key type of the secret key object.

```

### 8944 6.10.3 AES key generation

8945 The AES key generation mechanism, denoted **CKM\_AES\_KEY\_GEN**, is a key generation mechanism for  
8946 NIST's Advanced Encryption Standard.

8947 It does not have a parameter.

8948 The mechanism generates AES keys with a particular length in bytes, as specified in the  
8949 **CKA\_VALUE\_LEN** attribute of the template for the key.

8950 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
8951 key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the  
8952 key supports) may be specified in the template for the key, or else are assigned default initial values.

8953 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
8954 specify the supported range of AES key sizes, in bytes.

### 8955 6.10.4 AES-ECB

8956 AES-ECB, denoted **CKM\_AES\_ECB**, is a mechanism for single- and multiple-part encryption and  
8957 decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and  
8958 electronic codebook mode.

8959 It does not have a parameter.

8960 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
8961 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the  
8962 **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus  
8963 one null bytes so that the resulting length is a multiple of the block size. The output data is the same  
8964 length as the padded input data. It does not wrap the key type, key length, or any other information about  
8965 the key; the application must convey these separately.

8966 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the  
8967 **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the  
8968 **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE**  
8969 attribute of the new key; other attributes required by the key type must be specified in the template.

8970 Constraints on key types and the length of data are summarized in the following table:

8971 *Table 102, AES-ECB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	multiple of block size	same as input length	no final part
C_Decrypt	AES	multiple of block size	same as input length	no final part
C_WrapKey	AES	any	input length rounded up to multiple of block size	
C_UnwrapKey	AES	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

8972 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 8973 specify the supported range of AES key sizes, in bytes.

## 8974 6.10.5 AES-CBC

8975 AES-CBC, denoted **CKM\_AES\_CBC**, is a mechanism for single- and multiple-part encryption and  
 8976 decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and  
 8977 cipher-block chaining mode.

8978 It has a parameter, a 16-byte initialization vector.

8979 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
 8980 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the  
 8981 **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus  
 8982 one null bytes so that the resulting length is a multiple of the block size. The output data is the same  
 8983 length as the padded input data. It does not wrap the key type, key length, or any other information about  
 8984 the key; the application must convey these separately.

8985 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the  
 8986 **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the  
 8987 **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE**  
 8988 attribute of the new key; other attributes required by the key type must be specified in the template.

8989 Constraints on key types and the length of data are summarized in the following table:

8990 *Table 103, AES-CBC: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	multiple of block size	same as input length	no final part
C_Decrypt	AES	multiple of block size	same as input length	no final part
C_WrapKey	AES	any	input length rounded up to multiple of the block size	
C_UnwrapKey	AES	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

8991 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 8992 specify the supported range of AES key sizes, in bytes.

## 8993 6.10.6 AES-CBC with PKCS padding

8994 AES-CBC with PKCS padding, denoted **CKM\_AES\_CBC\_PAD**, is a mechanism for single- and multiple-  
 8995 part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced

8996 Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in PKCS  
8997 #7.

8998 It has a parameter, a 16-byte initialization vector.

8999 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the  
9000 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified  
9001 for the **CKA\_VALUE\_LEN** attribute.

9002 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,  
9003 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for  
9004 details). The entries in the table below for data length constraints when wrapping and unwrapping keys  
9005 do not apply to wrapping and unwrapping private keys.

9006 Constraints on key types and the length of data are summarized in the following table:

9007 *Table 104, AES-CBC with PKCS Padding: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt	AES	any	input length rounded up to multiple of the block size
C_Decrypt	AES	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	AES	any	input length rounded up to multiple of the block size
C_UnwrapKey	AES	multiple of block size	between 1 and block length bytes shorter than input length

9008 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
9009 specify the supported range of AES key sizes, in bytes.

## 9010 **6.10.7 AES-OFB**

9011 AES-OFB, denoted CKM\_AES\_OFB. It is a mechanism for single and multiple-part encryption and  
9012 decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

9013 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as  
9014 the block size.

9015 Constraints on key types and the length of data are summarized in the following table:

9016

9017 *Table 105, AES-OFB: Key And Data Length*

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

9018 For this mechanism the CK\_MECHANISM\_INFO structure is as specified for CBC mode.

## 9020 **6.10.8 AES-CFB**

9021 Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM\_AES\_CFB8, CKM\_AES\_CFB64, and  
9022 CKM\_AES\_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES.  
9023 AES-OFB mode is described [NIST sp800-38a].

9024 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as  
9025 the block size.

9026

9027 Constraints on key types and the length of data are summarized in the following table:

9028

9029 *Table 106, AES-CFB: Key And Data Length*

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

9030 For this mechanism the CK\_MECHANISM\_INFO structure is as specified for CBC mode.

### 9031 **6.10.9 General-length AES-MAC**

9032 General-length AES-MAC, denoted **CKM\_AES\_MAC\_GENERAL**, is a mechanism for single- and  
9033 multiple-part signatures and verification, based on NIST Advanced Encryption Standard as defined in  
9034 FIPS PUB 197 and data authentication as defined in FIPS PUB 113.

9035 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length  
9036 desired from the mechanism.

9037 The output bytes from this mechanism are taken from the start of the final AES cipher block produced in  
9038 the MACing process.

9039 Constraints on key types and the length of data are summarized in the following table:

9040 *Table 107, General-length AES-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	any	1-block size, as specified in parameters
C_Verify	AES	any	1-block size, as specified in parameters

9041 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
9042 specify the supported range of AES key sizes, in bytes.

### 9043 **6.10.10 AES-MAC**

9044 AES-MAC, denoted by **CKM\_AES\_MAC**, is a special case of the general-length AES-MAC mechanism.  
9045 AES-MAC always produces and verifies MACs that are half the block size in length.

9046 It does not have a parameter.

9047 Constraints on key types and the length of data are summarized in the following table:

9048 *Table 108, AES-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	½ block size (8 bytes)
C_Verify	AES	Any	½ block size (8 bytes)

9049 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
9050 specify the supported range of AES key sizes, in bytes.

### 9051 **6.10.11 AES-XCBC-MAC**

9052 AES-XCBC-MAC, denoted **CKM\_AES\_XCBC\_MAC**, is a mechanism for single and multiple part  
9053 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

9054 It does not have a parameter.

9055 Constraints on key types and the length of data are summarized in the following table:

9056 *Table 109, AES-XCBC-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	16 bytes
C_Verify	AES	Any	16 bytes

9057 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 9058 specify the supported range of AES key sizes, in bytes.

### 9059 **6.10.12 AES-XCBC-MAC-96**

9060 AES-XCBC-MAC-96, denoted **CKM\_AES\_XCBC\_MAC\_96**, is a mechanism for single and multiple part  
 9061 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

9062 It does not have a parameter.

9063 Constraints on key types and the length of data are summarized in the following table:

9064 *Table 110, AES-XCBC-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	12 bytes
C_Verify	AES	Any	12 bytes

9065 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 9066 specify the supported range of AES key sizes, in bytes.

## 9067 **6.11 AES with Counter**

9068 *Table 111, AES with Counter Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CTR	✓					✓	

### 9069 **6.11.1 Definitions**

9070 Mechanisms:

9071 CKM\_AES\_CTR

### 9072 **6.11.2 AES with Counter mechanism parameters**

#### 9073 ♦ **CK\_AES\_CTR\_PARAMS; CK\_AES\_CTR\_PARAMS\_PTR**

9074 **CK\_AES\_CTR\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CTR** mechanism.  
 9075 It is defined as follows:

```
9076     typedef struct CK_AES_CTR_PARAMS {
9077         CK ULONG ulCounterBits;
9078         CK BYTE cb[16];
9079     } CK_AES_CTR_PARAMS;
```

9081 ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This  
 9082 number shall be such that  $0 < ulCounterBits \leq 128$ . For any values outside this range the mechanism  
 9083 shall return **CKR\_MECHANISM\_PARAM\_INVALID**.

9084 It's up to the caller to initialize all of the bits in the counter block including the counter  
9085 bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting  
9086 with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

9087 E.g. as defined in [RFC 3686]:

```
9088    0          1          2          3
9089    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
9090    +-----+-----+-----+-----+-----+-----+-----+
9091    |           Nonce           |
9092    +-----+-----+-----+-----+-----+-----+-----+
9093    |           Initialization Vector (IV)           |
9094    |
9095    +-----+-----+-----+-----+-----+-----+-----+
9096    |           Block Counter           |
9097    +-----+-----+-----+-----+-----+-----+-----+
```

9098  
9099 This construction permits each packet to consist of up to  $2^{32}-1$  blocks = 4,294,967,295 blocks =
9100 68,719,476,720 octets.

9101 **CK\_AES\_CTR\_PARAMS\_PTR** is a pointer to a **CK\_AES\_CTR\_PARAMS**.

### 9102 6.11.3 AES with Counter Encryption / Decryption

9103 Generic AES counter mode is described in NIST Special Publication 800-38A and in RFC 3686. These  
9104 describe encryption using a counter block which may include a nonce to guarantee uniqueness of the  
9105 counter block. Since the nonce is not incremented, the mechanism parameter must specify the number of  
9106 counter bits in the counter block.

9107 The block counter is incremented by 1 after each block of plaintext is processed. There is no support for  
9108 any other increment functions in this mechanism.

9109 If an attempt to encrypt/decrypt is made which will cause an overflow of the counter block's counter bits,  
9110 then the mechanism shall return **CKR\_DATA\_LEN\_RANGE**. Note that the mechanism should allow the  
9111 final post increment of the counter to overflow (if it implements it this way) but not allow any further  
9112 processing after this point. E.g. if ulCounterBits = 2 and the counter bits start as 1 then only 3 blocks of  
9113 data can be processed.

9114

### 9115 6.12 AES CBC with Cipher Text Stealing CTS

9116 Ref [NIST AES CTS]

9117 This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to  
9118 the same length of cipher text.

9119 *Table 112, AES CBC with Cipher Text Stealing CTS Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CTS	✓					✓	

#### 9120 6.12.1 Definitions

9121 Mechanisms:

9122      CKM\_AES\_CTS

9123 **6.12.2 AES CTS mechanism parameters**

9124 It has a parameter, a 16-byte initialization vector.

9125 *Table 113, AES-CTS: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	Any, $\geq$ block size (16 bytes)	same as input length	no final part
C_Decrypt	AES	any, $\geq$ block size (16 bytes)	same as input length	no final part

9126

9127 **6.13 Additional AES Mechanisms**

9128 *Table 114, Additional AES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_AES_GCM	✓					✓	
CKM_AES_CCM	✓					✓	
CKM_AES_GMAC		✓					

9129

9130 **6.13.1 Definitions**

9131 Mechanisms:

9132 CKM\_AES\_GCM

9133 CKM\_AES\_CCM

9134 CKM\_AES\_GMAC

9135 Generator Functions:

9136 CKG\_NO\_GENERATE

9137 CKG\_GENERATE

9138 CKG\_GENERATE\_COUNTER

9139 CKG\_GENERATE\_RANDOM

9140 CKG\_GENERATE\_COUNTER\_XOR

9141 **6.13.2 AES-GCM Authenticated Encryption / Decryption**

9142 Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where  $K$  (key) and  $AAD$  (additional authenticated data) are as described in [GCM]. AES-GCM uses CK\_GCM\_PARAMS for Encrypt, Decrypt and CK\_GCM\_MESSAGE\_PARAMS for MessageEncrypt and MessageDecrypt.

9146 Encrypt:

- 9147 • Set the IV length  $ullvLen$  in the parameter block.

- 9148 • Set the IV data  $pIV$  in the parameter block.

- 9149     • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if  
9150     *ulAADLen* is 0.

- 9151     • Set the tag length *ulTagBits* in the parameter block.  
9152     • Call C\_EncryptInit() for **CKM\_AES\_GCM** mechanism with parameters and key *K*.  
9153     • Call C\_Encrypt(), or C\_EncryptUpdate()<sup>4</sup> C\_EncryptFinal(), for the plaintext obtaining ciphertext  
9154     and authentication tag output.

9155 Decrypt:

- 9156     • Set the IV length *ullvLen* in the parameter block.  
9157     • Set the IV data *p/v* in the parameter block.  
9158     • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if  
9159     *ulAADLen* is 0.  
9160     • Set the tag length *ulTagBits* in the parameter block.  
9161     • Call C\_DecryptInit() for **CKM\_AES\_GCM** mechanism with parameters and key *K*.  
9162     • Call C\_Decrypt(), or C\_DecryptUpdate()<sup>5</sup> C\_DecryptFinal(), for the ciphertext, including the  
9163     appended tag, obtaining plaintext output. Note: since **CKM\_AES\_GCM** is an AEAD cipher, no data  
9164     should be returned until C\_Decrypt() or C\_DecryptFinal().

9165 MessageEncrypt:

- 9166     • Set the IV length *ullvLen* in the parameter block.  
9167     • Set *p/v* to hold the IV data returned from C\_EncryptMessage() and C\_EncryptMessageBegin(). If  
9168     *ullvFixedBits* is not zero, then the most significant bits of *p/V* contain the fixed IV. If *ivGenerator* is  
9169     set to CKG\_NO\_GENERATE, *p/v* is an input parameter with the full IV.  
9170     • Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.  
9171     • Set the tag length *ulTagBits* in the parameter block.  
9172     • Set *pTag* to hold the tag data returned from C\_EncryptMessage() or the final  
9173     C\_EncryptMessageNext().  
9174     • Call C\_MessageEncryptInit() for **CKM\_AES\_GCM** mechanism key *K*.  
9175     • Call C\_EncryptMessage(), or C\_EncryptMessageBegin() followed by C\_EncryptMessageNext()<sup>5</sup>.  
9176     The mechanism parameter is passed to all three of these functions.  
9177     • Call C\_MessageEncryptFinal() to close the message decryption.

9178 MessageDecrypt:

- 9179     • Set the IV length *ullvLen* in the parameter block.  
9180     • Set the IV data *p/v* in the parameter block.  
9181     • The *ullvFixedBits* and *ivGenerator* fields are ignored.  
9182     • Set the tag length *ulTagBits* in the parameter block.  
9183     • Set the tag data *pTag* in the parameter block before C\_DecryptMessage() or the final  
9184     C\_DecryptMessageNext().

---

4 "\*" indicates 0 or more calls may be made as required

5 "\*" indicates 0 or more calls may be made as required

- Call C\_MessageDecryptInit() for **CKM\_AES\_GCM** mechanism key  $K$ .
- Call C\_DecryptMessage(), or C\_DecryptMessageBegin followed by C\_DecryptMessageNext()<sup>6</sup>. The mechanism parameter is passed to all three of these functions.
- Call C\_MessageDecryptFinal() to close the message decryption.

In  $pIV$  the least significant bit of the initialization vector is the rightmost bit.  $ullvLen$  is the length of the initialization vector in bytes.

On MessageEncrypt, the meaning of  $ivGenerator$  is as follows: CKG\_NO\_GENERATE means the IV is passed in on MessageEncrypt and no internal IV generation is done. CKG\_GENERATE means that the non-fixed portion of the IV is generated by the module internally. The generation method is not defined.

CKG\_GENERATE\_COUNTER means that the non-fixed portion of the IV is generated by the module internally by use of an incrementing counter, the initial IV counter is zero.

CKG\_GENERATE\_COUNTER\_XOR means that the non-fixed portion of the IV is xored with a counter. The value of the non-fixed portion passed must not vary from call to call. Like CKG\_GENERATE\_COUNTER, the counter starts at zero.

CKG\_GENERATE\_RANDOM means that the non-fixed portion of the IV is generated by the module internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in  $pIV$ .

Modules must implement CKG\_GENERATE. Modules may also reject  $ullvFixedBits$  values which are too large. Zero is always an acceptable value for  $ullvFixedBits$ .

In Encrypt and Decrypt the tag is appended to the cipher text and the least significant bit of the tag is the rightmost bit and the tag bits are the rightmost  $ulTagBits$  bits. In MessageEncrypt the tag is returned in the  $pTag$  field of CK\_GCM\_MESSAGE\_PARAMS. In MessageDecrypt the tag is provided by the  $pTag$  field of CK\_GCM\_MESSAGE\_PARAMS.

The key type for  $K$  must be compatible with **CKM\_AES\_ECB** and the C\_EncryptInit()/C\_DecryptInit()/C\_MessageEncryptInit()/C\_MessageDecryptInit() calls shall behave, with respect to  $K$ , as if they were called directly with **CKM\_AES\_ECB**,  $K$  and NULL parameters.

### 6.13.3 AES-CCM authenticated Encryption / Decryption

For IPsec (RFC 4309) and also for use in ZFS encryption. Generic CCM mode is described in [RFC 3610].

To set up for AES-CCM use the following process, where  $K$  (key), nonce and additional authenticated data are as described in [RFC 3610]. AES-CCM uses CK\_CCM\_PARAMS for Encrypt and Decrypt, and CK\_CCM\_MESSAGE\_PARAMS for MessageEncrypt and MessageDecrypt.

Encrypt:

- Set the message/data length  $ulDataLen$  in the parameter block.
- Set the nonce length  $ulNonceLen$  and the nonce data  $pNonce$  in the parameter block.
- Set the AAD data  $pAAD$  and size  $ulAADLen$  in the parameter block.  $pAAD$  may be NULL if  $ulAADLen$  is 0.
- Set the MAC length  $ulMACLen$  in the parameter block.
- Call C\_EncryptInit() for **CKM\_AES\_CCM** mechanism with parameters and key  $K$ .
- Call C\_Encrypt(), C\_EncryptUpdate(), or C\_EncryptFinal(), for the plaintext obtaining the final ciphertext output and the MAC. The total length of data processed must be  $ulDataLen$ . The output length will be  $ulDataLen + ulMACLen$ .

Decrypt:

---

<sup>6</sup> “\*” indicates 0 or more calls may be made as required

- Set the message/data length *ulDataLen* in the parameter block. This length must not include the length of the MAC that is appended to the cipher text.
- Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- Set the MAC length *ulMACLen* in the parameter block.
- Call *C\_DecryptInit()* for **CKM\_AES\_CCM** mechanism with parameters and key *K*.
- Call *C\_Decrypt()*, *C\_DecryptUpdate()*, or *C\_DecryptFinal()*, for the ciphertext, including the appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen* + *ulMACLen*. Note: since **CKM\_AES\_CCM** is an AEAD cipher, no data should be returned until *C\_Decrypt()* or *C\_DecryptFinal()*.

9238 MessageEncrypt:

- Set the message/data length *ulDataLen* in the parameter block.
- Set the nonce length *ulNonceLen*.
- Set *pNonce* to hold the nonce data returned from *C\_EncryptMessage()* and *C\_EncryptMessageBegin()*. If *ulNonceFixedBits* is not zero, then the most significant bits of *pNonce* contain the fixed nonce. If *nonceGenerator* is set to *CKG\_NO\_GENERATE*, *pNonce* is an input parameter with the full nonce.
- Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
- Set the MAC length *ulMACLen* in the parameter block.
- Set *pMAC* to hold the MAC data returned from *C\_EncryptMessage()* or the final *C\_EncryptMessageNext()*.
- Call *C\_MessageEncryptInit()* for **CKM\_AES\_CCM** mechanism key *K*.
- Call *C\_EncryptMessage()*, or *C\_EncryptMessageBegin()* followed by *C\_EncryptMessageNext()*<sup>7</sup>. The mechanism parameter is passed to all three functions.
- Call *C\_MessageEncryptFinal()* to close the message encryption.
- The MAC is returned in *pMac* of the *CK\_CCM\_MESSAGE\_PARAMS* structure.

9249 MessageDecrypt:

- Set the message/data length *ulDataLen* in the parameter block.
- Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block
- The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.
- Set the MAC length *ulMACLen* in the parameter block.
- Set the MAC data *pMAC* in the parameter block before *C\_DecryptMessage()* or the final *C\_DecryptMessageNext()*.
- Call *C\_MessageDecryptInit()* for **CKM\_AES\_CCM** mechanism key *K*.
- Call *C\_DecryptMessage()*, or *C\_DecryptMessageBegin()* followed by *C\_DecryptMessageNext()*<sup>8</sup>. The mechanism parameter is passed to all three functions.

7 <sup>\*\*</sup> indicates 0 or more calls may be made as required

8 <sup>\*\*</sup> indicates 0 or more calls may be made as required

- 9264     • Call C\_MessageDecryptFinal() to close the message decryption.

9265 In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce  
9266 in bytes.

9267 On MessageEncrypt, the meaning of *nonceGenerator* is as follows: CKG\_NO\_GENERATE means the  
9268 nonce is passed in on MessageEncrypt and no internal MAC generation is done. CKG\_GENERATE  
9269 means that the non-fixed portion of the nonce is generated by the module internally. The generation  
9270 method is not defined.

9271 CKG\_GENERATE\_COUNTER means that the non-fixed portion of the nonce is generated by the module  
9272 internally by use of an incrementing counter, the initial IV counter is zero.

9273 CKG\_GENERATE\_COUNTER\_XOR means that the non-fixed portion of the IV is xored with a counter.  
9274 The value of the non-fixed portion passed must not vary from call to call. Like  
9275 CKG\_GENERATE\_COUNTER, the counter starts at zero.

9276 CKG\_GENERATE\_RANDOM means that the non-fixed portion of the nonce is generated by the module  
9277 internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

9278 Modules must implement CKG\_GENERATE. Modules may also reject *ulNonceFixedBits* values which are  
9279 too large. Zero is always an acceptable value for *ulNonceFixedBits*.

9280 In Encrypt and Decrypt the MAC is appended to the cipher text and the least significant byte of the MAC  
9281 is the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In MessageEncrypt the MAC  
9282 is returned in the *pMAC* field of CK\_CCM\_MESSAGE\_PARAMS. In MessageDecrypt the MAC is  
9283 provided by the *pMAC* field of CK\_CCM\_MESSAGE\_PARAMS.

9284 The key type for K must be compatible with CKM\_AES\_ECB and the  
9285 C\_EncryptInit()/C\_DecryptInit()/C\_MessageEncryptInit()/C\_MessageDecryptInit() calls shall behave, with  
9286 respect to K, as if they were called directly with CKM\_AES\_ECB, K and NULL parameters.

## 9287 6.13.4 AES-GMAC

9288 AES-GMAC, denoted CKM\_AES\_GMAC, is a mechanism for single and multiple-part signatures and  
9289 verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of  
9290 GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism  
9291 parameters. When GMAC is used with C\_Sign or C\_Verify, pData points to the AAD. GMAC does not  
9292 use plaintext or ciphertext.

9293 The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the  
9294 CK\_GCM\_PARAMS field *ulTagBits*.

9295 The IV length is determined by the CK\_GCM\_PARAMS field *ullvLen*.

9296 Constraints on key types and the length of data are summarized in the following table:

9297 *Table 115, AES-GMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	< 2^64	Depends on param's ulTagBits
C_Verify	CKK_AES	< 2^64	Depends on param's ulTagBits

9298 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK\_MECHANISM\_INFO structure  
9299 specify the supported range of AES key sizes, in bytes.

## 9300 6.13.5 AES GCM and CCM Mechanism parameters

### 9301 ♦ CK\_GENERATOR\_FUNCTION

9302 Functions to generate unique IVs and nonces.

```
9303     typedef CK_ULONG CK_GENERATOR_FUNCTION;
```

#### 9304 ◆ **CK\_GCM\_PARAMS; CK\_GCM\_PARAMS\_PTR**

9305 CK\_GCM\_PARAMS is a structure that provides the parameters to the CKM\_AES\_GCM mechanism  
9306 when used for Encrypt or Decrypt. It is defined as follows:

```
9307     typedef struct CK_GCM_PARAMS {
9308         CK_BYTE_PTR    pIv;
9309         CK ULONG       ulIvLen;
9310         CK ULONG       ulIvBits;
9311         CK_BYTE_PTR    pAAD;
9312         CK ULONG       ulAADLen;
9313         CK ULONG       ulTagBits;
9314     } CK_GCM_PARAMS;
```

9315

9316 The fields of the structure have the following meanings:

9317	plv	pointer to initialization vector
9318	ullvLen	length of initialization vector in bytes. The length of the initialization vector can be any number between 1 and $(2^{32}) - 1$ . 96-bit (12 byte) IV values can be processed more efficiently, so that length is recommended for situations in which efficiency is critical.
9322	ullvBits	length of initialization vector in bits. Do not use ullvBits to specify the length of the initialization vector, but ullvLen instead.
9324	pAAD	pointer to additional authentication data. This data is authenticated but not encrypted.
9326	ulAADLen	length of pAAD in bytes. The length of the AAD can be any number between 0 and $(2^{32}) - 1$ .
9328	ulTagBits	length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128.

9330 CK\_GCM\_PARAMS\_PTR is a pointer to a CK\_GCM\_PARAMS.

#### 9331 ◆ **CK\_GCM\_MESSAGE\_PARAMS; CK\_GCM\_MESSAGE\_PARAMS\_PTR**

9332 CK\_GCM\_MESSAGE\_PARAMS is a structure that provides the parameters to the CKM\_AES\_GCM mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```
9334     typedef struct CK_GCM_MESSAGE_PARAMS {
9335         CK_BYTE_PTR    pIv;
9336         CK ULONG       ulIvLen;
9337         CK ULONG       ulIvFixedBits;
9338         CK_GENERATOR_FUNCTION ivGenerator;
9339         CK_BYTE_PTR    pTag;
9340         CK ULONG       ulTagBits;
9341     } CK_GCM_MESSAGE_PARAMS;
```

9342

9343 The fields of the structure have the following meanings:

9344 plv pointer to initialization vector

9345                   
 9346                   vector can be any number between 1 and (2<sup>32</sup>) - 1. 96-bit (12 byte)  
 9347                   IV values can be processed more efficiently, so that length is  
 9348                   recommended for situations in which efficiency is critical.  
 9349                   
 9350                   new IV. These bits are counted from the Most significant bits (to the  
 9351                   right).  
 9352                   ivGenerator   Function used to generate a new IV. Each IV must be unique for a  
 9353                   given session.  
 9354                   pTag          location of the authentication tag which is returned on  
 9355                   MessageEncrypt, and provided on MessageDecrypt.  
 9356                   ulTagBits   length of authentication tag in bits. Can be any value between 0 and  
 9357                   128.

9358 **CK\_GCM\_MESSAGE\_PARAMS\_PTR** is a pointer to a **CK\_GCM\_MESSAGE\_PARAMS**.

9359

## 9360 ◆ **CK\_CCM\_PARAMS; CK\_CCM\_PARAMS\_PTR**

9361 **CK\_CCM\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CCM** mechanism  
 9362 when used for Encrypt or Decrypt. It is defined as follows:

```

 9363        typedef struct CK_CCM_PARAMS {
 9364            CK ULONG       ulDataLen; /*plaintext or ciphertext*/
 9365            CK BYTE PTR  pNonce;
 9366            CK ULONG       ulNonceLen;
 9367            CK BYTE PTR  pAAD;
 9368            CK ULONG       ulAADLen;
 9369            CK ULONG       ulMACLen;
 9370        } CK_CCM_PARAMS;
  
```

9371 The fields of the structure have the following meanings, where L is the size in bytes of the data length's  
 9372 length (2 <= L <= 8):

9373                   ulDataLen	length of the data where 0 <= ulDataLen < 2^(8L).
9374                   pNonce	the nonce.
9375                   ulNonceLen	length of pNonce in bytes where 7 <= ulNonceLen <= 13.
9376                   pAAD	Additional authentication data. This data is authenticated but not 9377                    encrypted.
9378                   ulAADLen	length of pAAD in bytes where 0 <= ulAADLen <= (2 <sup>32</sup> ) - 1.
9379                   ulMACLen	length of the MAC (output following cipher text) in bytes. Valid 9380                    values are 4, 6, 8, 10, 12, 14, and 16.

9381 **CK\_CCM\_PARAMS\_PTR** is a pointer to a **CK\_CCM\_PARAMS**.

## 9382 ◆ **CK\_CCM\_MESSAGE\_PARAMS; CK\_CCM\_MESSAGE\_PARAMS\_PTR**

9383 **CK\_CCM\_MESSAGE\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CCM**  
 9384 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

 9385        typedef struct CK_CCM_MESSAGE_PARAMS {
 9386            CK ULONG       ulDataLen; /*plaintext or ciphertext*/
 9387            CK BYTE PTR  pNonce;
  
```

```

9388     CK ULONG      ulNonceLen;
9389     CK ULONG      ulNonceFixedBits;
9390     CK GENERATOR FUNCTION  nonceGenerator;
9391     CK BYTE PTR   pMAC;
9392     CK ULONG      ulMACLen;
9393 } CK_CCM_MESSAGE_PARAMS;

9394

9395 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
9396 length (2 <= L <= 8):
9397     ulDataLen    length of the data where 0 <= ulDataLen < 2^(8L).
9398     pNonce       the nonce.
9399     ulNonceLen   length of pNonce in bytes where 7 <= ulNonceLen <= 13.
9400     ulNonceFixedBits number of bits of the original nonce to preserve when generating a
9401                               new nonce. These bits are counted from the Most significant bits (to
9402                               the right).
9403     nonceGenerator Function used to generate a new nonce. Each nonce must be
9404                               unique for a given session.
9405     pMAC          location of the CCM MAC returned on MessageEncrypt, provided on
9406                               MessageDecrypt
9407     ulMACLen     length of the MAC (output following cipher text) in bytes. Valid
9408                               values are 4, 6, 8, 10, 12, 14, and 16.

9409 CK_CCM_MESSAGE_PARAMS_PTR is a pointer to a CK_CCM_MESSAGE_PARAMS.
9410

```

## 6.14 AES CMAC

Table 116, Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_AES_CMAC_GENERAL		✓					
CKM_AES_CMAC		✓					

<sup>1</sup> SR = SignRecover, VR = VerifyRecover.

### 6.14.1 Definitions

Mechanisms:

```

CKM_AES_CMAC_GENERAL
CKM_AES_CMAC

```

### 6.14.2 Mechanism parameters

```

CKM_AES_CMAC_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure.
CKM_AES_CMAC does not use a mechanism parameter.

```

### 6.14.3 General-length AES-CMAC

General-length AES-CMAC, denoted **CKM\_AES\_CMAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on [NIST SP800-38B] and [RFC 4493].

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final AES cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

*Table 117, General-length AES-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	any	1-block size, as specified in parameters
C_Verify	CKK_AES	any	1-block size, as specified in parameters

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### 6.14.4 AES-CMAC

AES-CMAC, denoted **CKM\_AES\_CMAC**, is a special case of the general-length AES-CMAC mechanism. AES-CMAC always produces and verifies MACs that are a full block size in length, the default output length specified by [RFC 4493].

Constraints on key types and the length of data are summarized in the following table:

*Table 118, AES-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	any	Block size (16 bytes)
C_Verify	CKK_AES	any	Block size (16 bytes)

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

## 6.15 AES XTS

*Table 119, Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_AES_XTS	✓					✓	
CKM_AES_XTS_KEY_GEN					✓		

## 9448 6.15.1 Definitions

9449 This section defines the key type “CKK\_AES\_XTS” for type CK\_KEY\_TYPE as used in the  
9450 CKA\_KEY\_TYPE attribute of key objects.

9451 Mechanisms:

9452 CKM\_AES\_XTS

9453 CKM\_AES\_XTS\_KEY\_GEN

## 9454 6.15.2 AES-XTS secret key objects

9455 *Table 120, AES-XTS Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (32 or 64 bytes)
CKA_VALUE_LEN <sup>2,3,6</sup>	CK_ULONG	Length in bytes of key value

9456 Refer to Table 11 for footnotes

## 9457 6.15.3 AES-XTS key generation

9458 The double-length AES-XTS key generation mechanism, denoted **CKM\_AES\_XTS\_KEY\_GEN**, is a key  
9459 generation mechanism for double-length AES-XTS keys.

9460 The mechanism generates AES-XTS keys with a particular length in bytes as specified in the  
9461 CKA\_VALUE\_LEN attributes of the template for the key.

9462 This mechanism contributes the CKA\_CLASS, CKA\_KEY\_TYPE, and CKA\_VALUE attributes to the new  
9463 key. Other attributes supported by the double-length AES-XTS key type (specifically, the flags indicating  
9464 which functions the key supports) may be specified in the template for the key, or else are assigned  
9465 default initial values.

9466 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK\_MECHANISM\_INFO structure  
9467 specify the supported range of AES-XTS key sizes, in bytes.

## 9468 6.15.4 AES-XTS

9469 AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted **CKM\_AES\_XTS**,  
9470 is a mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

9471 Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and  
9472 64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraints on  
9473 key types and the length of data are summarized in the following table:

9474 *Table 121, AES-XTS: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_AES_XTS	Any, ≥ block size (16 bytes)	Same as input length	No final part
C_Decrypt	CKK_AES_XTS	Any, ≥ block size (16 bytes)	Same as input length	No final part

9475

## 9476 6.16 AES Key Wrap

9477 *Table 122, AES Key Wrap Mechanisms vs. Functions*

9478

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_KEY_WRAP	✓					✓	
CKM_AES_KEY_WRAP_PAD	✓					✓	
CKM_AES_KEY_WRAP_KWP	✓					✓	
CKM_AES_KEY_WRAP_PKCS7	✓					✓	

<sup>1</sup>SR = SignRecover, VR = VerifyRecover

## 9479 6.16.1 Definitions

9480 Mechanisms:

9481 CKM\_AES\_KEY\_WRAP  
 9482 CKM\_AES\_KEY\_WRAP\_PAD  
 9483 CKM\_AES\_KEY\_WRAP\_KWP  
 9484 CKM\_AES\_KEY\_WRAP\_PKCS7

## 9485 6.16.2 AES Key Wrap Mechanism parameters

9486 The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if  
 9487 present, must be a fixed size array of 8 bytes for CKM\_AES\_KEY\_WRAP and  
 9488 CKM\_AES\_KEY\_WRAP\_PKCS7, resp. 4 bytes for CKM\_AES\_KEY\_WRAP\_KWP; and, if NULL, will use  
 9489 the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

9490 The type of this parameter is CK\_BYTE\_PTR and the pointer points to the array of bytes to be used as  
 9491 the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM\_AES\_KEY\_WRAP and  
 9492 CKM\_AES\_KEY\_WRAP\_PKCS7, resp. 4 for CKM\_AES\_KEY\_WRAP\_KWP, and the pointer non-NULL.

## 9493 6.16.3 AES Key Wrap

9494 The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and  
 9495 single-part encryption and decryption.

### 9496 ◆ CKM\_AES\_KEY\_WRAP

9497 The CKM\_AES\_KEY\_WRAP mechanism can wrap a key of any length. A secret key whose length is not  
 9498 a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is  
 9499 defined in Section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the  
 9500 encoded private key will be zero padded to fit if necessary.

9501

9502 The CKM\_AES\_KEY\_WRAP mechanism can only encrypt a block of data whose size is an exact multiple  
 9503 of the AES Key Wrap algorithm semiblock size.

9504

9505 For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it truncates the result  
 9506 according to the CKA\_KEY\_TYPE attribute of the template and, if it has one and the key type supports it,  
 9507 the CKA\_VALUE\_LEN attribute of the template. The length specified in the template must not be less  
 9508 than n-7 bytes, where n is the length of the wrapped key. In case of a private key, the mechanism parses  
 9509 the encoding as defined in section 6.7 and ignores trailing zero bytes.

9510 ◆ **CKM\_AES\_KEY\_WRAP\_PAD**

9511 The CKM\_AES\_KEY\_WRAP\_PAD mechanism is deprecated. CKM\_AES\_KEY\_WRAP\_KWP resp.  
9512 CKM\_AES\_KEY\_WRAP\_PKCS7 shall be used instead.

9513 ◆ **CKM\_AES\_KEY\_WRAP\_KWP**

9514 The CKM\_AES\_KEY\_WRAP\_KWP mechanism can wrap a key or encrypt block of data of any length.  
9515 The input is zero-padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which  
9516 produces same results as RFC 5649.

9517 ◆ **CKM\_AES\_KEY\_WRAP\_PKCS7**

9518 The CKM\_AES\_KEY\_WRAP\_PKCS7 mechanism can wrap a key or encrypt a block of data of any  
9519 length. It does the padding detailed in PKCS #7 of inputs (keys or data blocks) up to a semiblock size to  
9520 make it an exact multiple of AES Key Wrap algorithm semiblock size (8bytes), always producing  
9521 wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token  
9522 before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of  
9523 data as defined in Section 6.2 of [AES KEYWRAP].

9524 **6.17 Key derivation by data encryption – DES & AES**

9525 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.  
9526 They are for use with the C\_DeriveKey function.

9527 *Table 123, Key derivation by data encryption Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES_ECB_ENCRYPT_DATA							✓
CKM_DES_CBC_ENCRYPT_DATA							✓
CKM_DES3_ECB_ENCRYPT_DATA							✓
CKM_DES3_CBC_ENCRYPT_DATA							✓
CKM_AES_ECB_ENCRYPT_DATA							✓
CKM_AES_CBC_ENCRYPT_DATA							✓

9528 **6.17.1 Definitions**

9529 Mechanisms:

9530 CKM\_DES\_ECB\_ENCRYPT\_DATA

9531 CKM\_DES\_CBC\_ENCRYPT\_DATA

9532 CKM\_DES3\_ECB\_ENCRYPT\_DATA

9533 CKM\_DES3\_CBC\_ENCRYPT\_DATA

9534 CKM\_AES\_ECB\_ENCRYPT\_DATA

9535 CKM\_AES\_CBC\_ENCRYPT\_DATA

9536  
9537      **typedef struct CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS {**  
9538            **CK\_BYTE            iv[8];**

```

9539     CK_BYTE_PTR    pData;
9540     CK ULONG        length;
9541 } CK DES CBC ENCRYPT DATA PARAMS;
9542
9543     typedef CK DES CBC ENCRYPT DATA PARAMS CK PTR
9544             CK DES CBC ENCRYPT DATA PARAMS PTR;
9545
9546     typedef struct CK AES CBC ENCRYPT DATA PARAMS {
9547         CK BYTE          iv[16];
9548         CK BYTE_PTR    pData;
9549         CK ULONG        length;
9550     } CK AES CBC ENCRYPT DATA PARAMS;
9551
9552     typedef CK AES CBC ENCRYPT DATA PARAMS CK PTR
9553             CK AES CBC ENCRYPT DATA PARAMS PTR;

```

## 6.17.2 Mechanism Parameters

Uses CK\_KEY\_DERIVATION\_STRING\_DATA as defined in section 6.43.2

*Table 124, Mechanism Parameters*

CKM DES ECB ENCRYPT DATA CKM DES3 ECB ENCRYPT DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long.
CKM AES ECB ENCRYPT DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM DES CBC ENCRYPT DATA CKM DES3 CBC ENCRYPT DATA	Uses CK DES CBC ENCRYPT DATA PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 8 bytes long.
CKM AES CBC ENCRYPT DATA	Uses CK AES CBC ENCRYPT DATA PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

## 6.17.3 Mechanism Description

The mechanisms will function by performing the encryption over the data provided using the base key. The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data. The derived key shall be defined by the attribute template supplied but constrained by the length of cipher text available for the key value and other normal PKCS11 derivation constraints.

Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1 Key Derivation mechanism in section 6.20.5.

If the data is too short to make the requested key then the mechanism returns CKR DATA LEN RANGE.

## 6.18 Double and Triple-length DES

*Table 125, Double and Triple-Length DES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES2_KEY_GEN					✓		
CKM_DES3_KEY_GEN					✓		
CKM_DES3_ECB	✓					✓	
CKM_DES3_CBC	✓					✓	
CKM_DES3_CBC_PAD	✓					✓	
CKM_DES3_MAC_GENERAL		✓					
CKM_DES3_MAC		✓					

## 9569 6.18.1 Definitions

9570 This section defines the key type "CKK\_DES2" and "CKK\_DES3" for type CK\_KEY\_TYPE as used in the  
 9571 CKA\_KEY\_TYPE attribute of key objects.

9572 Mechanisms:

9573 CKM\_DES2\_KEY\_GEN  
 9574 CKM\_DES3\_KEY\_GEN  
 9575 CKM\_DES3\_ECB  
 9576 CKM\_DES3\_CBC  
 9577 CKM\_DES3\_MAC  
 9578 CKM\_DES3\_MAC\_GENERAL  
 9579 CKM\_DES3\_CBC\_PAD

## 9580 6.18.2 DES2 secret key objects

9581 DES2 secret key objects (object class **CKO\_SECRET\_KEY**, key type **CKK\_DES2**) hold double-length  
 9582 DES keys. The following table defines the DES2 secret key object attributes, in addition to the common  
 9583 attributes defined for this object class:

9584 *Table 126, DES2 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (always 16 bytes long)

9585 Refer to Table 11 for footnotes

9586 DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of  
 9587 the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or  
 9588 unwrap a DES2 key with incorrect parity will return an error.

9589 The following is a sample template for creating a double-length DES secret key object:

```
9590 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
9591 CK_KEY_TYPE keyType = CKK_DES2;
9592 CK_UTF8CHAR label[] = "A DES2 secret key object";
9593 CK_BYTE value[16] = {...};
9594 CK_BBOOL true = CK_TRUE;
9595 CK_ATTRIBUTE template[] = {
9596     {CKA_CLASS, &class, sizeof(class)},
9597     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
```

```

9598     { CKA_TOKEN, &true, sizeof(true) },
9599     { CKA_LABEL, label, sizeof(label)-1 },
9600     { CKA_ENCRYPT, &true, sizeof(true) },
9601     { CKA_VALUE, value, sizeof(value) }
9602 } ;
9603
9604 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
9605 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
9606 the key type of the secret key object.

```

### 6.18.3 DES3 secret key objects

DES3 secret key objects (object class **CKO\_SECRET\_KEY**, key type **CKK\_DES3**) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes defined for this object class:

*Table 127, DES3 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (always 24 bytes long)

Refer to Table 11 for footnotes

DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or unwrap a DES3 key with incorrect parity will return an error.

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

```

9617 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
9618 CK_KEY_TYPE keyType = CKK_DES3;
9619 CK_UTF8CHAR label[] = "A DES3 secret key object";
9620 CK_BYTE value[24] = {...};
9621 CK_BBOOL true = CK_TRUE;
9622 CK_ATTRIBUTE template[] = {
9623     { CKA_CLASS, &class, sizeof(class) },
9624     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
9625     { CKA_TOKEN, &true, sizeof(true) },
9626     { CKA_LABEL, label, sizeof(label)-1 },
9627     { CKA_ENCRYPT, &true, sizeof(true) },
9628     { CKA_VALUE, value, sizeof(value) }
9629 } ;
9630

```

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

### 6.18.4 Double-length DES key generation

The double-length DES key generation mechanism, denoted **CKM\_DES2\_KEY\_GEN**, is a key generation mechanism for double-length DES keys. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in FIPS PUB 46-3.

It does not have a parameter.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which

functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys:

**CKM\_DES3\_ECB**, **CKM\_DES3\_CBC**, **CKM\_DES3\_CBC\_PAD**, **CKM\_DES3\_MAC\_GENERAL**, and **CKM\_DES3\_MAC**. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be “weak” or “semi-weak” keys.

## 6.18.5 Triple-length DES Order of Operations

Triple-length DES encryptions are carried out as specified in FIPS PUB 46-3: encrypt, decrypt, encrypt. Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical representations of the encrypt and decrypt operations are as follows:

$$\text{DES3-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P)))$$

$$\text{DES3-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P)))$$

## 6.18.6 Triple-length DES in CBC Mode

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

$$\text{DES3-CBC-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P + I)))$$

$$\text{DES3-CBC-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P))) + I$$

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

## 6.18.7 DES and Triple length DES in OFB Mode

Table 128, DES and Triple Length DES in OFB Mode Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_DES_OFB64	✓						
CKM_DES_OFB8	✓						
CKM_DES_CFB64	✓						
CKM_DES_CFB8	✓						

9665

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM\_DES\_OFB8** and **CKM\_DES\_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.

Constraints on key types and the length of data are summarized in the following table:

Table 129, OFB: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

9672 For this mechanism the **CK\_MECHANISM\_INFO** structure is as specified for CBC mode.

## 6.18.8 DES and Triple length DES in CFB Mode

9674 Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM\_DES\_CFB8** and **CKM\_DES\_CFB64**.  
9675 It is a mechanism for single and multiple-part encryption and decryption with DES.

9676 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as  
9677 the block size.

9678 Constraints on key types and the length of data are summarized in the following table:

9679 *Table 130, CFB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

9680 For this mechanism the **CK\_MECHANISM\_INFO** structure is as specified for CBC mode.

## 6.19 Double and Triple-length DES CMAC

9682 *Table 131, Double and Triple-length DES CMAC Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES3_CMAC_GENERAL		✓					
CKM_DES3_CMAC		✓					

9683 <sup>1</sup> SR = SignRecover, VR = VerifyRecover.

### 6.19.1 Definitions

9685 Mechanisms:

9686       **CKM\_DES3\_CMAC\_GENERAL**

9687       **CKM\_DES3\_CMAC**

### 6.19.2 Mechanism parameters

9689 **CKM\_DES3\_CMAC\_GENERAL** uses the existing **CK\_MAC\_GENERAL\_PARAMS** structure.  
9690 **CKM\_DES3\_CMAC** does not use a mechanism parameter.

### 6.19.3 General-length DES3-MAC

General-length DES3-CMAC, denoted **CKM\_DES3\_CMAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

*Table 132, General-length DES3-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	1-block size, as specified in parameters
C_Verify	CKK_DES3 CKK_DES2	any	1-block size, as specified in parameters

Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits (which means using the entire block for DES). The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used

### 6.19.4 DES3-CMAC

DES3-CMAC, denoted **CKM\_DES3\_CMAC**, is a special case of the general-length DES3-CMAC mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the DES3 block length is the minimum output length recommended by [NIST sp800-38b].

Constraints on key types and the length of data are summarized in the following table:

*Table 133, DES3-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	Block size (8 bytes)
C_Verify	CKK_DES3 CKK_DES2	any	Block size (8 bytes)

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

## 6.20 SHA-1

*Table 134, SHA-1 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA_1				✓			
CKM_SHA_1_HMAC_GENERAL		✓					
CKM_SHA_1_HMAC		✓					
CKM_SHA1_KEY_DERIVATION							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA_1_KEY_GEN					✓		

## 9716 6.20.1 Definitions

9717 This section defines the key type "CKK\_SHA\_1\_HMAC" for type CK\_KEY\_TYPE as used in the  
 9718 CKA\_KEY\_TYPE attribute of key objects.

9719 Mechanisms:

9720       CKM\_SHA\_1  
 9721       CKM\_SHA\_1\_HMAC  
 9722       CKM\_SHA\_1\_HMAC\_GENERAL  
 9723       CKM\_SHA1\_KEY\_DERIVATION  
 9724       CKM\_SHA\_1\_KEY\_GEN  
 9725

## 9726 6.20.2 SHA-1 digest

9727 The SHA-1 mechanism, denoted **CKM\_SHA\_1**, is a mechanism for message digesting, following the  
 9728 Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

9729 It does not have a parameter.

9730 Constraints on the length of input and output data are summarized in the following table. For single-part  
 9731 digesting, the data and the digest may begin at the same location in memory.

9732 *Table 135, SHA-1: Data Length*

Function	Input length	Digest length
C_Digest	any	20

## 9733 6.20.3 General-length SHA-1-HMAC

9734 The general-length SHA-1-HMAC mechanism, denoted **CKM\_SHA\_1\_HMAC\_GENERAL**, is a  
 9735 mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash  
 9736 function. The keys it uses are generic secret keys and CKK\_SHA\_1\_HMAC.

9737 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 9738 output. This length should be in the range 1-20 (the output size of SHA-1 is 20 bytes). Signatures  
 9739 (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

9740 *Table 136, General-length SHA-1-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret CKK_SHA_1_HMAC	any	1-20, depending on parameters
C_Verify	generic secret CKK_SHA_1_HMAC	any	1-20, depending on parameters

## 6.20.4 SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted **CKM\_SHA\_1\_HMAC**, is a special case of the general-length SHA-1-HMAC mechanism in Section 6.20.3.

It has no parameter, and always produces an output of length 20.

## 6.20.5 SHA-1 key derivation

SHA-1 key derivation, denoted **CKM\_SHA1\_KEY\_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the opposite value from its **CKA\_EXTRACTABLE** attribute.

## 6.20.6 SHA-1 HMAC key generation

The SHA-1-HMAC key generation mechanism, denoted **CKM\_SHA\_1\_KEY\_GEN**, is a key generation mechanism for NIST's SHA-1-HMAC.

It does not have a parameter.

9778 The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the  
 9779 **CKA\_VALUE\_LEN** attribute of the template for the key.

9780 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 9781 key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which  
 9782 functions the key supports) may be specified in the template for the key, or else are assigned default  
 9783 initial values.

9784 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 9785 specify the supported range of **CKM\_SHA\_1\_HMAC** key sizes, in bytes.

## 9786 **6.21 SHA-224**

9787 *Table 137, SHA-224 Mechanisms vs. Functions*

<b>Mechanism</b>	<b>Functions</b>						
	<b>Encrypt &amp; Decrypt</b>	<b>Sign &amp; Verify</b>	<b>SR &amp; VR<sup>1</sup></b>	<b>Digest</b>	<b>Gen. Key/ Key Pair</b>	<b>Wrap &amp; Unwrap</b>	<b>Derive</b>
CKM_SHA224				✓			
CKM_SHA224_HMAC		✓					
CKM_SHA224_HMAC_GENERAL		✓					
CKM_SHA224_KEY_DERIVATION							✓
CKM_SHA224_KEY_GEN					✓		

### 9788 **6.21.1 Definitions**

9789 This section defines the key type "CKK\_SHA224\_HMAC" for type CK\_KEY\_TYPE as used in the  
 9790 CKA\_KEY\_TYPE attribute of key objects.

9791 Mechanisms:

9792     CKM\_SHA224  
 9793     CKM\_SHA224\_HMAC  
 9794     CKM\_SHA224\_HMAC\_GENERAL  
 9795     CKM\_SHA224\_KEY\_DERIVATION  
 9796     CKM\_SHA224\_KEY\_GEN

### 9797 **6.21.2 SHA-224 digest**

9798 The SHA-224 mechanism, denoted **CKM\_SHA224**, is a mechanism for message digesting, following the  
 9799 Secure Hash Algorithm with a 224-bit message digest defined in FIPS PUB 180-4.

9800 It does not have a parameter.

9801 Constraints on the length of input and output data are summarized in the following table. For single-part  
 9802 digesting, the data and the digest may begin at the same location in memory.

9803 *Table 138, SHA-224: Data Length*

<b>Function</b>	<b>Input length</b>	<b>Digest length</b>
C_Digest	any	28

### 9804 **6.21.3 General-length SHA-224-HMAC**

9805 The general-length SHA-224-HMAC mechanism, denoted **CKM\_SHA224\_HMAC\_GENERAL**, is the  
 9806 same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based

9807 on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are  
9808 generic secret keys and CKK\_SHA224\_HMAC. FIPS-198 compliant tokens may require the key length to  
9809 be at least 14 bytes; that is, half the size of the SHA-224 hash output.

9810 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
9811 output. This length should be in the range 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198  
9812 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).  
9813 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC  
9814 output.

9815 *Table 139, General-length SHA-224-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret CKK_SHA224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret CKK_SHA224_HMAC	Any	1-28, depending on parameters

## 9816 **6.21.4 SHA-224-HMAC**

9817 The SHA-224-HMAC mechanism, denoted **CKM\_SHA224\_HMAC**, is a special case of the general-length  
9818 SHA-224-HMAC mechanism.

9819 It has no parameter, and always produces an output of length 28.

## 9820 **6.21.5 SHA-224 key derivation**

9821 SHA-224 key derivation, denoted **CKM\_SHA224\_KEY\_DERIVATION**, is the same as the SHA-1 key  
9822 derivation mechanism in Section 6.20.5 except that it uses the SHA-224 hash function and the relevant  
9823 length is 28 bytes.

## 9824 **6.21.6 SHA-224 HMAC key generation**

9825 The SHA-224-HMAC key generation mechanism, denoted **CKM\_SHA224\_KEY\_GEN**, is a key  
9826 generation mechanism for NIST's SHA224-HMAC.

9827 It does not have a parameter.

9828 The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the  
9829 **CKA\_VALUE\_LEN** attribute of the template for the key.

9830 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
9831 key. Other attributes supported by the SHA224-HMAC key type (specifically, the flags indicating which  
9832 functions the key supports) may be specified in the template for the key, or else are assigned default  
9833 initial values.

9834 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
9835 specify the supported range of **CKM\_SHA224\_HMAC** key sizes, in bytes.

## 9836 **6.22 SHA-256**

9837 *Table 140, SHA-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA256				✓			
CKM_SHA256_HMAC_GENERAL		✓					
CKM_SHA256_HMAC		✓					
CKM_SHA256_KEY_DERIVATION							✓
CKM_SHA256_KEY_GEN					✓		

## 6.22.1 Definitions

This section defines the key type "CKK\_SHA256\_HMAC" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

- CKM\_SHA256
- CKM\_SHA256\_HMAC
- CKM\_SHA256\_HMAC\_GENERAL
- CKM\_SHA256\_KEY\_DERIVATION
- CKM\_SHA256\_KEY\_GEN

## 6.22.2 SHA-256 digest

The SHA-256 mechanism, denoted **CKM\_SHA256**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 256-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

*Table 141, SHA-256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

## 6.22.3 General-length SHA-256-HMAC

The general-length SHA-256-HMAC mechanism, denoted **CKM\_SHA256\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_SHA256\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

*Table 142, General-length SHA-256-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret, CKK_SHA256_HMAC	Any	1-32, depending on parameters

## 6.22.4 SHA-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM\_SHA256\_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 6.22.3.

It has no parameter, and always produces an output of length 32.

## 6.22.5 SHA-256 key derivation

SHA-256 key derivation, denoted CKM\_SHA256\_KEY\_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

## 6.22.6 SHA-256 HMAC key generation

The SHA-256-HMAC key generation mechanism, denoted **CKM\_SHA256\_KEY\_GEN**, is a key generation mechanism for NIST's SHA256-HMAC.

It does not have a parameter.

The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA256\_HMAC** key sizes, in bytes.

## 6.23 SHA-384

Table 143, *SHA-384 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA384				✓			
CKM_SHA384_HMAC_GENERAL		✓					
CKM_SHA384_HMAC		✓					
CKM_SHA384_KEY_DERIVATION							✓
CKM_SHA384_KEY_GEN					✓		

## 6.23.1 Definitions

This section defines the key type "CKK\_SHA384\_HMAC" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

9891 CKM\_SHA384  
9892 CKM\_SHA384\_HMAC  
9893 CKM\_SHA384\_HMAC\_GENERAL  
9894 CKM\_SHA384\_KEY\_DERIVATION  
9895 CKM\_SHA384\_KEY\_GEN

### 9896 6.23.2 SHA-384 digest

9897 The SHA-384 mechanism, denoted **CKM\_SHA384**, is a mechanism for message digesting, following the  
9898 Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

9899 It does not have a parameter.

9900 Constraints on the length of input and output data are summarized in the following table. For single-part  
9901 digesting, the data and the digest may begin at the same location in memory.

9902 *Table 144, SHA-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

### 9903 6.23.3 General-length SHA-384-HMAC

9904 The general-length SHA-384-HMAC mechanism, denoted **CKM\_SHA384\_HMAC\_GENERAL**, is the  
9905 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC  
9906 construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

9907 The keys it uses are generic secret keys and CKK\_SHA384\_HMAC. FIPS-198 compliant tokens may  
9908 require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

9909 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
9910 output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198  
9911 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).  
9912 Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC  
9913 output.

9914 *Table 145, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret, CKK_SHA384_HMAC	Any	1-48, depending on parameters

9915

### 9916 6.23.4 SHA-384-HMAC

9917 The SHA-384-HMAC mechanism, denoted **CKM\_SHA384\_HMAC**, is a special case of the general-length  
9918 SHA-384-HMAC mechanism.

9919 It has no parameter, and always produces an output of length 48.

### 9920 6.23.5 SHA-384 key derivation

9921 SHA-384 key derivation, denoted **CKM\_SHA384\_KEY\_DERIVATION**, is the same as the SHA-1 key  
9922 derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant  
9923 length is 48 bytes.

## 6.23.6 SHA-384 HMAC key generation

The SHA-384-HMAC key generation mechanism, denoted **CKM\_SHA384\_KEY\_GEN**, is a key generation mechanism for NIST's SHA384-HMAC.

It does not have a parameter.

The mechanism generates SHA384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA384\_HMAC** key sizes, in bytes.

## 6.24 SHA-512

Table 146, SHA-512 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512				✓			
CKM_SHA512_HMAC_GENERAL		✓					
CKM_SHA512_HMAC		✓					
CKM_SHA512_KEY_DERIVATION							✓
CKM_SHA512_KEY_GEN					✓		

### 6.24.1 Definitions

This section defines the key type "CKK\_SHA512\_HMAC" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA512

CKM\_SHA512\_HMAC

CKM\_SHA512\_HMAC\_GENERAL

CKM\_SHA512\_KEY\_DERIVATION

CKM\_SHA512\_KEY\_GEN

### 6.24.2 SHA-512 digest

The SHA-512 mechanism, denoted **CKM\_SHA512**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 147, SHA-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

### 9954 6.24.3 General-length SHA-512-HMAC

9955 The general-length SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_HMAC\_GENERAL**, is the  
 9956 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC  
 9957 construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

9958 The keys it uses are generic secret keys and CKK\_SHA512\_HMAC. FIPS-198 compliant tokens may  
 9959 require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

9960 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 9961 output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198  
 9962 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).  
 9963 Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC  
 9964 output.

9965 *Table 148, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA512_HMAC	Any	1-64, depending on parameters
C_Verify	generic secret, CKK_SHA512_HMAC	Any	1-64, depending on parameters

9966

### 9967 6.24.4 SHA-512-HMAC

9968 The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_HMAC**, is a special case of the general-length  
 9969 SHA-512-HMAC mechanism.

9970 It has no parameter, and always produces an output of length 64.

### 9971 6.24.5 SHA-512 key derivation

9972 SHA-512 key derivation, denoted **CKM\_SHA512\_KEY\_DERIVATION**, is the same as the SHA-1 key  
 9973 derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant  
 9974 length is 64 bytes.

### 9975 6.24.6 SHA-512 HMAC key generation

9976 The SHA-512-HMAC key generation mechanism, denoted **CKM\_SHA512\_KEY\_GEN**, is a key  
 9977 generation mechanism for NIST's SHA512-HMAC.

9978 It does not have a parameter.

9979 The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the  
 9980 **CKA\_VALUE\_LEN** attribute of the template for the key.

9981 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 9982 key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which  
 9983 functions the key supports) may be specified in the template for the key, or else are assigned default  
 9984 initial values.

9985 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 9986 specify the supported range of **CKM\_SHA512\_HMAC** key sizes, in bytes.

## 9987 6.25 SHA-512/224

9988 *Table 149, SHA-512/224 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_224				✓			
CKM_SHA512_224_HMAC_GENERAL		✓					
CKM_SHA512_224_HMAC		✓					
CKM_SHA512_224_KEY_DERIVATION							✓
CKM_SHA512_224_KEY_GEN					✓		

### 9989 6.25.1 Definitions

9990 This section defines the key type “CKK\_SHA512\_224\_HMAC” for type CK\_KEY\_TYPE as used in the  
 9991 CKA\_KEY\_TYPE attribute of key objects.

9992 Mechanisms:

9993       CKM\_SHA512\_224  
 9994       CKM\_SHA512\_224\_HMAC  
 9995       CKM\_SHA512\_224\_HMAC\_GENERAL  
 9996       CKM\_SHA512\_224\_KEY\_DERIVATION  
 9997       CKM\_SHA512\_224\_KEY\_GEN

### 9998 6.25.2 SHA-512/224 digest

9999 The SHA-512/224 mechanism, denoted **CKM\_SHA512\_224**, is a mechanism for message digesting,  
 10000 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit  
 10001 message digest with a distinct initial hash value and truncated to 224 bits. **CKM\_SHA512\_224** is the  
 10002 same as **CKM\_SHA512\_T** with a parameter value of 224.

10003 It does not have a parameter.

10004 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10005 digesting, the data and the digest may begin at the same location in memory.

10006 *Table 150, SHA-512/224: Data Length*

Function	Input length	Digest length
C_Digest	any	28

### 10007 6.25.3 General-length SHA-512/224-HMAC

10008 The general-length SHA-512/224-HMAC mechanism, denoted **CKM\_SHA512\_224\_HMAC\_GENERAL**,  
 10009 is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the  
 10010 HMAC construction based on the SHA-512/224 hash function and length of the output should be in the  
 10011 range 1-28. The keys it uses are generic secret keys and CKK\_SHA512\_224\_HMAC. FIPS-198  
 10012 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-  
 10013 512/224 hash output.

10014 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 10015 output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198  
 10016 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).  
 10017 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC  
 10018 output.

10019 *Table 151, General-length SHA-384-HMAC: Key And Data Length*

<b>Function</b>	<b>Key type</b>	<b>Data length</b>	<b>Signature length</b>
C_Sign	generic secret, CKK_SHA512_224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret, CKK_SHA512_224_HMAC	Any	1-28, depending on parameters

10020

## 6.25.4 SHA-512/224-HMAC

10022 The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_224\_HMAC**, is a special case of the general-  
 10023 length SHA-512/224-HMAC mechanism.  
 10024 It has no parameter, and always produces an output of length 28.

## 6.25.5 SHA-512/224 key derivation

10026 The SHA-512/224 key derivation, denoted **CKM\_SHA512\_224\_KEY\_DERIVATION**, is the same as the  
 10027 SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/224 hash function  
 10028 and the relevant length is 28 bytes.

## 6.25.6 SHA-512/224 HMAC key generation

10030 The SHA-512/224-HMAC key generation mechanism, denoted **CKM\_SHA512\_224\_KEY\_GEN**, is a key  
 10031 generation mechanism for NIST's SHA512/224-HMAC.

10032 It does not have a parameter.

10033 The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the  
 10034 **CKA\_VALUE\_LEN** attribute of the template for the key.

10035 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 10036 key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating  
 10037 which functions the key supports) may be specified in the template for the key, or else are assigned  
 10038 default initial values.

10039 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 10040 specify the supported range of **CKM\_SHA512\_224\_HMAC** key sizes, in bytes.

## 6.26 SHA-512/256

10042 *Table 152, SHA-512/256 Mechanisms vs. Functions*

<b>Mechanism</b>	<b>Functions</b>						
	<b>Encrypt &amp; Decrypt</b>	<b>Sign &amp; Verify</b>	<b>SR &amp; VR<sup>1</sup></b>	<b>Digest</b>	<b>Gen. Key/Key Pair</b>	<b>Wrap &amp; Unwrap</b>	<b>Derive</b>
CKM_SHA512_256				✓			

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_256_HMAC_GENERAL		✓					
CKM_SHA512_256_HMAC		✓					
CKM_SHA512_256_KEY_DERIVATION							✓
CKM_SHA512_256_KEY_GEN					✓		

## 10043 6.26.1 Definitions

10044 This section defines the key type “CKK\_SHA512\_256\_HMAC” for type CK\_KEY\_TYPE as used in the  
 10045 CKA\_KEY\_TYPE attribute of key objects.

10046 Mechanisms:

10047 CKM\_SHA512\_256  
 10048 CKM\_SHA512\_256\_HMAC  
 10049 CKM\_SHA512\_256\_HMAC\_GENERAL  
 10050 CKM\_SHA512\_256\_KEY\_DERIVATION  
 10051 CKM\_SHA512\_256\_KEY\_GEN

## 10052 6.26.2 SHA-512/256 digest

10053 The SHA-512/256 mechanism, denoted **CKM\_SHA512\_256**, is a mechanism for message digesting,  
 10054 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit  
 10055 message digest with a distinct initial hash value and truncated to 256 bits. **CKM\_SHA512\_256** is the  
 10056 same as **CKM\_SHA512\_T** with a parameter value of 256.

10057 It does not have a parameter.

10058 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10059 digesting, the data and the digest may begin at the same location in memory.

10060 *Table 153, SHA-512/256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

## 10061 6.26.3 General-length SHA-512/256-HMAC

10062 The general-length SHA-512/256-HMAC mechanism, denoted **CKM\_SHA512\_256\_HMAC\_GENERAL**,  
 10063 is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the  
 10064 HMAC construction based on the SHA-512/256 hash function and length of the output should be in the  
 10065 range 1-32. The keys it uses are generic secret keys and CKK\_SHA512\_256\_HMAC. FIPS-198  
 10066 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-  
 10067 512/256 hash output.

10068 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 10069 output. This length should be in the range 1-32 (the output size of SHA-512/256 is 32 bytes). FIPS-198  
 10070 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).  
 10071 Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC  
 10072 output.

10073 *Table 154, General-length SHA-384-HMAC: Key And Data Length*

<b>Function</b>	<b>Key type</b>	<b>Data length</b>	<b>Signature length</b>
C_Sign	generic secret, CKK_SHA512_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret, CKK_SHA512_256_HMAC	Any	1-32, depending on parameters

10074

## 6.26.4 SHA-512/256-HMAC

10076 The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_256\_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

10077

10078 It has no parameter, and always produces an output of length 32.

## 6.26.5 SHA-512/256 key derivation

10080 The SHA-512/256 key derivation, denoted **CKM\_SHA512\_256\_KEY\_DERIVATION**, is the same as the  
10081 SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/256 hash function  
10082 and the relevant length is 32 bytes.

## 6.26.6 SHA-512/256 HMAC key generation

10084 The SHA-512/256-HMAC key generation mechanism, denoted **CKM\_SHA512\_256\_KEY\_GEN**, is a key  
10085 generation mechanism for NIST's SHA512/256-HMAC.

10086 It does not have a parameter.

10087 The mechanism generates SHA512/256-HMAC keys with a particular length in bytes, as specified in the  
10088 **CKA\_VALUE\_LEN** attribute of the template for the key.

10089 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10090 key. Other attributes supported by the SHA512/256-HMAC key type (specifically, the flags indicating  
10091 which functions the key supports) may be specified in the template for the key, or else are assigned  
10092 default initial values.

10093 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10094 specify the supported range of **CKM\_SHA512\_256\_HMAC** key sizes, in bytes.

## 6.27 SHA-512/t

10096 *Table 155, SHA-512 / t Mechanisms vs. Functions*

<b>Mechanism</b>	<b>Functions</b>						
	<b>Encrypt &amp; Decrypt</b>	<b>Sign &amp; Verify</b>	<b>SR &amp; VR<sup>1</sup></b>	<b>Digest</b>	<b>Gen . Key/ Key Pair</b>	<b>Wrap &amp; Unwrap</b>	<b>Derive</b>
CKM_SHA512_T				✓			
CKM_SHA512_T_HMAC_GENERAL		✓					
CKM_SHA512_T_HMAC		✓					
CKM_SHA512_T_KEY_DERIVATION							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sub>1</sub>	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_T_KEY_GEN					✓		

10097 **6.27.1 Definitions**

10098 This section defines the key type “CKK\_SHA512\_T\_HMAC” for type CK\_KEY\_TYPE as used in the  
 10099 CKA\_KEY\_TYPE attribute of key objects.

10100 Mechanisms:

10101 CKM\_SHA512\_T  
 10102 CKM\_SHA512\_T\_HMAC  
 10103 CKM\_SHA512\_T\_HMAC\_GENERAL  
 10104 CKM\_SHA512\_T\_KEY\_DERIVATION  
 10105 CKM\_SHA512\_T\_KEY\_GEN

10106 **6.27.2 SHA-512/t digest**

10107 The SHA-512/t mechanism, denoted **CKM\_SHA512\_T**, is a mechanism for message digesting, following  
 10108 the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message  
 10109 digest with a distinct initial hash value and truncated to t bits.

10110 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the value of t in bits. The length in  
 10111 bytes of the desired output should be in the range of 0– $\lceil t/8 \rceil$ , where 0 < t < 512, and t > 384.

10112 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10113 digesting, the data and the digest may begin at the same location in memory.

10114 *Table 156, SHA-512/256: Data Length*

Function	Input length	Digest length
C_Digest	any	$\lceil t/8 \rceil$ , where 0 < t < 512, and t > 384

10115 **6.27.3 General-length SHA-512/t-HMAC**

10116 The general-length SHA-512/t-HMAC mechanism, denoted **CKM\_SHA512\_T\_HMAC\_GENERAL**, is the  
 10117 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC  
 10118 construction based on the SHA-512/t hash function and length of the output should be in the range 0 –  
 10119  $\lceil t/8 \rceil$ , where 0 < t < 512, and t > 384.

10120 **6.27.4 SHA-512/t-HMAC**

10121 The SHA-512/t-HMAC mechanism, denoted **CKM\_SHA512\_T\_HMAC**, is a special case of the general-  
 10122 length SHA-512/t-HMAC mechanism.

10123 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the value of t in bits. The length in  
 10124 bytes of the desired output should be in the range of 0– $\lceil t/8 \rceil$ , where 0 < t < 512, and t > 384.

10125 **6.27.5 SHA-512/t key derivation**

10126 The SHA-512/t key derivation, denoted **CKM\_SHA512\_T\_KEY\_DERIVATION**, is the same as the SHA-  
10127 512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/t hash function and the  
10128 relevant length is  $\lceil t/8 \rceil$  bytes, where  $0 < t < 512$ , and  $t \geq 384$ .

10129 **6.27.6 SHA-512/t HMAC key generation**

10130 The SHA-512/t-HMAC key generation mechanism, denoted **CKM\_SHA512\_T\_KEY\_GEN**, is a key  
10131 generation mechanism for NIST's SHA512/t-HMAC.

10132 It does not have a parameter.

10133 The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the  
10134 **CKA\_VALUE\_LEN** attribute of the template for the key.

10135 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10136 key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which  
10137 functions the key supports) may be specified in the template for the key, or else are assigned default  
10138 initial values.

10139 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10140 specify the supported range of **CKM\_SHA512\_T\_HMAC** key sizes, in bytes.

10141

10142 **6.28 SHA3-224**

10143 *Table 157, SHA3-224 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_224				✓			
CKM_SHA3_224_HMAC		✓					
CKM_SHA3_224_HMAC_GENERAL		✓					
CKM_SHA3_224_KEY_DERIVATION							✓
CKM_SHA3_224_KEY_GEN					✓		

10144 **6.28.1 Definitions**

10145 Mechanisms:

10146 CKM\_SHA3\_224

10147 CKM\_SHA3\_224\_HMAC

10148 CKM\_SHA3\_224\_HMAC\_GENERAL

10149 CKM\_SHA3\_224\_KEY\_DERIVATION

10150 CKM\_SHA3\_224\_KEY\_GEN

10151

10152 CKK\_SHA3\_224\_HMAC

## 6.28.2 SHA3-224 digest

The SHA3-224 mechanism, denoted **CKM\_SHA3\_224**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 224-bit message digest defined in FIPS Pub 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

*Table 158, SHA3-224: Data Length*

Function	Input length	Digest length
C_Digest	any	28

## 6.28.3 General-length SHA3-224-HMAC

The general-length SHA3-224-HMAC mechanism, denoted **CKM\_SHA3\_224\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in section 6.20.4 except that it uses the HMAC construction based on the SHA3-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK\_SHA3\_224\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA3-224 hash output.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-28 (the output size of SHA3-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 28-byte HMAC output.

*Table 159, General-length SHA3-224-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret or CKK_SHA3_224_HMAC	Any	1-28, depending on parameters

## 6.28.4 SHA3-224-HMAC

The SHA3-224-HMAC mechanism, denoted **CKM\_SHA3\_224\_HMAC**, is a special case of the general-length SHA3-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

## 6.28.5 SHA3-224 key derivation

SHA-224 key derivation, denoted **CKM\_SHA3\_224\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the SHA3-224 hash function and the relevant length is 28 bytes.

## 6.28.6 SHA3-224 HMAC key generation

The SHA3-224-HMAC key generation mechanism, denoted **CKM\_SHA3\_224\_KEY\_GEN**, is a key generation mechanism for NIST's SHA3-224-HMAC.

It does not have a parameter.

The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which

10188 functions the key supports) may be specified in the template for the key, or else are assigned default  
 10189 initial values.  
 10190 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 10191 specify the supported range of **CKM\_SHA3\_224\_HMAC** key sizes, in bytes.

## 10192 **6.29 SHA3-256**

10193 *Table 160, SHA3-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_256				✓			
CKM_SHA3_256_HMAC_GENERAL		✓					
CKM_SHA3_256_HMAC		✓					
CKM_SHA3_256_KEY_DERIVATION							✓
CKM_SHA3_256_KEY_GEN					✓		

### 10194 **6.29.1 Definitions**

10195 Mechanisms:  
 10196 CKM\_SHA3\_256  
 10197 CKM\_SHA3\_256\_HMAC  
 10198 CKM\_SHA3\_256\_HMAC\_GENERAL  
 10199 CKM\_SHA3\_256\_KEY\_DERIVATION  
 10200 CKM\_SHA3\_256\_KEY\_GEN  
 10201  
 10202 CKK\_SHA3\_256\_HMAC

### 10203 **6.29.2 SHA3-256 digest**

10204 The SHA3-256 mechanism, denoted **CKM\_SHA3\_256**, is a mechanism for message digesting, following  
 10205 the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.  
 10206 It does not have a parameter.  
 10207 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10208 digesting, the data and the digest may begin at the same location in memory.  
 10209 *Table 161, SHA3-256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

### 10210 **6.29.3 General-length SHA3-256-HMAC**

10211 The general-length SHA3-256-HMAC mechanism, denoted **CKM\_SHA3\_256\_HMAC\_GENERAL**, is the  
 10212 same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC  
 10213 construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.  
 10214 The keys it uses are generic secret keys and CKK\_SHA3\_256\_HMAC. FIPS-198 compliant tokens may  
 10215 require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.

10216 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 10217 output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198  
 10218 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).  
 10219 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC  
 10220 output.

10221 *Table 162, General-length SHA3-256-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret or CKK_SHA3_256_HMAC	Any	1-32, depending on parameters

## 10222 **6.29.4 SHA3-256-HMAC**

10223 The SHA-256-HMAC mechanism, denoted **CKM\_SHA3\_256\_HMAC**, is a special case of the general-  
 10224 length SHA-256-HMAC mechanism.

10225 It has no parameter, and always produces an output of length 32.

## 10226 **6.29.5 SHA3-256 key derivation**

10227 SHA-256 key derivation, denoted **CKM\_SHA3\_256\_KEY\_DERIVATION**, is the same as the SHA-1 key  
 10228 derivation mechanism in Section 6.20.5, except that it uses the SHA3-256 hash function and the relevant  
 10229 length is 32 bytes.

## 10230 **6.29.6 SHA3-256 HMAC key generation**

10231 The SHA3-256-HMAC key generation mechanism, denoted **CKM\_SHA3\_256\_KEY\_GEN**, is a key  
 10232 generation mechanism for NIST's SHA3-256-HMAC.

10233 It does not have a parameter.

10234 The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the  
 10235 **CKA\_VALUE\_LEN** attribute of the template for the key.

10236 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 10237 key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which  
 10238 functions the key supports) may be specified in the template for the key, or else are assigned default  
 10239 initial values.

10240 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 10241 specify the supported range of **CKM\_SHA3\_256\_HMAC** key sizes, in bytes.

10242

## 10243 **6.30 SHA3-384**

10244 *Table 163, SHA3-384 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384				✓			
CKM_SHA3_384_HMAC_GENERAL		✓					
CKM_SHA3_384_HMAC		✓					
CKM_SHA3_384_KEY_DERIVATION							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384_KEY_GEN				✓			

### 6.30.1 Definitions

CKM\_SHA3\_384  
 CKM\_SHA3\_384\_HMAC  
 CKM\_SHA3\_384\_HMAC\_GENERAL  
 CKM\_SHA3\_384\_KEY\_DERIVATION  
 CKM\_SHA3\_384\_KEY\_GEN  
 CKK\_SHA3\_384\_HMAC

### 6.30.2 SHA3-384 digest

The SHA3-384 mechanism, denoted **CKM\_SHA3\_384**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 164, SHA3-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

### 6.30.3 General-length SHA3-384-HMAC

The general-length SHA3-384-HMAC mechanism, denoted **CKM\_SHA3\_384\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48. The keys it uses are generic secret keys and CKK\_SHA3\_384\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

10266

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

Table 165, General-length SHA3-384-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret or CKK_SHA3_384_HMAC	Any	1-48, depending on parameters

10273

## 6.30.4 SHA3-384-HMAC

The SHA3-384-HMAC mechanism, denoted **CKM\_SHA3\_384\_HMAC**, is a special case of the general-length SHA3-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

## 6.30.5 SHA3-384 key derivation

SHA3-384 key derivation, denoted **CKM\_SHA3\_384\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

## 6.30.6 SHA3-384 HMAC key generation

The SHA3-384-HMAC key generation mechanism, denoted **CKM\_SHA3\_384\_KEY\_GEN**, is a key generation mechanism for NIST's SHA3-384-HMAC.

It does not have a parameter.

The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA3\_384\_HMAC** key sizes, in bytes.

## 6.31 SHA3-512

Table 166, SHA-512 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_512				✓			
CKM_SHA3_512_HMAC_GENERAL		✓					
CKM_SHA3_512_HMAC		✓					
CKM_SHA3_512_KEY_DERIVATION							✓
CKM_SHA3_512_KEY_GEN				✓			

## 6.31.1 Definitions

CKM\_SHA3\_512

CKM\_SHA3\_512\_HMAC

CKM\_SHA3\_512\_HMAC\_GENERAL

CKM\_SHA3\_512\_KEY\_DERIVATION

CKM\_SHA3\_512\_KEY\_GEN

CKK\_SHA3\_512\_HMAC

### 6.31.2 SHA3-512 digest

The SHA3-512 mechanism, denoted **CKM\_SHA3\_512**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 512-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

*Table 167, SHA3-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

### 6.31.3 General-length SHA3-512-HMAC

The general-length SHA3-512-HMAC mechanism, denoted **CKM\_SHA3\_512\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC construction based on the SHA3-512 hash function and length of the output should be in the range 1-64. The keys it uses are generic secret keys and CKK\_SHA3\_512\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

*Table 168, General-length SHA3-512-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_512_HMAC	Any	1-64, depending on parameters
C_Verify	generic secret or CKK_SHA3_512_HMAC	Any	1-64, depending on parameters

### 6.31.4 SHA3-512-HMAC

The SHA3-512-HMAC mechanism, denoted **CKM\_SHA3\_512\_HMAC**, is a special case of the general-length SHA3-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### 6.31.5 SHA3-512 key derivation

SHA3-512 key derivation, denoted **CKM\_SHA3\_512\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

### 6.31.6 SHA3-512 HMAC key generation

The SHA3-512-HMAC key generation mechanism, denoted **CKM\_SHA3\_512\_KEY\_GEN**, is a key generation mechanism for NIST's SHA3-512-HMAC.

It does not have a parameter.

The mechanism generates SHA3-512-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

10339 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10340 key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which  
10341 functions the key supports) may be specified in the template for the key, or else are assigned default  
10342 initial values.

10343 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10344 specify the supported range of **CKM\_SHA3\_512\_HMAC** key sizes, in bytes.

## 10345 6.32 SHAKE

10346 *Table 169, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHAKE_128_KEY_DERIVATION							✓
CKM_SHAKE_256_KEY_DERIVATION							✓

### 10347 6.32.1 Definitions

10348 CKM\_SHAKE\_128\_KEY\_DERIVATION  
10349 CKM\_SHAKE\_256\_KEY\_DERIVATION

### 10350 6.32.2 SHAKE Key Derivation

10351 SHAKE-128 and SHAKE-256 key derivation, denoted **CKM\_SHAKE\_128\_KEY\_DERIVATION** and  
10352 **CKM\_SHAKE\_256\_KEY\_DERIVATION**, implements the SHAKE expansion function defined in FIPS 202  
10353 on the input key.

- 10354 • If no length or key type is provided in the template a **CKR\_TEMPLATE\_INCOMPLETE** error is  
10355 generated.
- 10356 • If no key type is provided in the template, but a length is, then the key produced by this mechanism  
10357 shall be a generic secret key of the specified length.
- 10358 • If no length was provided in the template, but a key type is, then that key type must have a well-  
10359 defined length. If it does, then the key produced by this mechanism shall be of the type specified in  
10360 the template. If it doesn't, an error shall be returned.
- 10361 • If both a key type and a length are provided in the template, the length must be compatible with that  
10362 key type. The key produced by this mechanism shall be of the specified type and length.

10363 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set  
10364 properly.

10365 This mechanism has the following rules about key sensitivity and extractability:

- 10366 • The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
10367 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
10368 default value.
- 10369 • If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
10370 shall as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
10371 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
10372 **CKA\_SENSITIVE** attribute.
  - 10373 • Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then  
10374 the derived key shall, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
10375 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
10376 value from its **CKA\_EXTRACTABLE** attribute.

10377 

## 6.33 BLAKE2B-160

10378 *Table 170, BLAKE2B-160 Mechanisms vs. Functions*

<b>Mechanism</b>	<b>Functions</b>						
	<b>Encrypt &amp; Decrypt</b>	<b>Sign &amp; Verify</b>	<b>SR &amp; VR<sup>1</sup></b>	<b>Digest</b>	<b>Gen. Key/Key Pair</b>	<b>Wrap &amp; Unwrap</b>	<b>Derive</b>
CKM_BLAKE2B_160				✓			
CKM_BLAKE2B_160_HMAC		✓					
CKM_BLAKE2B_160_HMAC_GENERAL		✓					
CKM_BLAKE2B_160_KEY_DERIVE							✓
CKM_BLAKE2B_160_KEY_GEN					✓		

10379 

### 6.33.1 Definitions

10380 Mechanisms:

10381 CKM\_BLAKE2B\_160  
 10382 CKM\_BLAKE2B\_160\_HMAC  
 10383 CKM\_BLAKE2B\_160\_HMAC\_GENERAL  
 10384 CKM\_BLAKE2B\_160\_KEY\_DERIVE  
 10385 CKM\_BLAKE2B\_160\_KEY\_GEN  
 10386 CKK\_BLAKE2B\_160\_HMAC

10387 

### 6.33.2 BLAKE2B-160 digest

10388 The BLAKE2B-160 mechanism, denoted **CKM\_BLAKE2B\_160**, is a mechanism for message digesting,  
 10389 following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [RFC 7693](#).

10390 It does not have a parameter.

10391 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10392 digesting, the data and the digest may begin at the same location in memory.

10393 *Table 171, BLAKE2B-160: Data Length*

<b>Function</b>	<b>Input length</b>	<b>Digest length</b>
C_Digest	any	20

10394 

### 6.33.3 General-length BLAKE2B-160-HMAC

10395 The general-length BLAKE2B-160-HMAC mechanism, denoted  
 10396 **CKM\_BLAKE2B\_160\_HMAC\_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output  
 10397 should be in the range 1-20. The keys it uses are generic secret keys and CKK\_BLAKE2B\_160\_HMAC.

10398 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 10399 output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures  
 10400 (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

10401 *Table 172, General-length BLAKE2B-160-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_160_H MAC	Any	1-20, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_160_H MAC	Any	1-20, depending on parameters

#### 10402 **6.33.4 BLAKE2B-160-HMAC**

10403 The BLAKE2B-160-HMAC mechanism, denoted **CKM\_BLAKE2B\_160\_HMAC**, is a special case of the  
10404 general-length BLAKE2B-160-HMAC mechanism.

10405 It has no parameter, and always produces an output of length 20.

#### 10406 **6.33.5 BLAKE2B-160 key derivation**

10407 BLAKE2B-160 key derivation, denoted **CKM\_BLAKE2B\_160\_KEY\_DERIVE**, is the same as the SHA-1  
10408 key derivation mechanism in Section 6.20.5 except that it uses the BLAKE2B-160 hash function and the  
10409 relevant length is 20 bytes.

#### 10410 **6.33.6 BLAKE2B-160 HMAC key generation**

10411 The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_160\_KEY\_GEN**, is a  
10412 key generation mechanism for BLAKE2B-160-HMAC.

10413 It does not have a parameter.

10414 The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the  
10415 **CKA\_VALUE\_LEN** attribute of the template for the key.

10416 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10417 key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating  
10418 which functions the key supports) may be specified in the template for the key, or else are assigned  
10419 default initial values.

10420 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10421 specify the supported range of **CKM\_BLAKE2B\_160\_HMAC** key sizes, in bytes.

#### 10422 **6.34 BLAKE2B-256**

10423 *Table 173, BLAKE2B-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_256				✓			
CKM_BLAKE2B_256_HMAC_GENERAL		✓					
CKM_BLAKE2B_256_HMAC		✓					
CKM_BLAKE2B_256_KEY_DERIVE							✓
CKM_BLAKE2B_256_KEY_GEN					✓		

### 6.34.1 Definitions

Mechanisms:

CKM\_BLAKE2B\_256  
CKM\_BLAKE2B\_256\_HMAC  
CKM\_BLAKE2B\_256\_HMAC\_GENERAL  
CKM\_BLAKE2B\_256\_KEY\_DERIVE  
CKM\_BLAKE2B\_256\_KEY\_GEN  
CKK\_BLAKE2B\_256\_HMAC

### 6.34.2 BLAKE2B-256 digest

The BLAKE2B-256 mechanism, denoted **CKM\_BLAKE2B\_256**, is a mechanism for message digesting, following the Blake2b Algorithm with a 256-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 174, BLAKE2B-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

### 6.34.3 General-length BLAKE2B-256-HMAC

The general-length BLAKE2B-256-HMAC mechanism, denoted **CKM\_BLAKE2B\_256\_HMAC\_GENERAL**, is the keyed variant of Blake2b-256 and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_BLAKE2B\_256\_HMAC.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

Table 175, General-length BLAKE2B-256-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_256_HMAC	Any	1-32, depending on parameters

### 6.34.4 BLAKE2B-256-HMAC

The BLAKE2B-256-HMAC mechanism, denoted **CKM\_BLAKE2B\_256\_HMAC**, is a special case of the general-length BLAKE2B-256-HMAC mechanism in Section 6.34.3.

It has no parameter, and always produces an output of length 32.

### 6.34.5 BLAKE2B-256 key derivation

BLAKE2B-256 key derivation, denoted **CKM\_BLAKE2B\_256\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-256 hash function and the relevant length is 32 bytes.

## 6.34.6 BLAKE2B-256 HMAC key generation

The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_256\_KEY\_GEN**, is a key generation mechanism for BLAKE2B-256-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_256\_HMAC** key sizes, in bytes.

## 6.35 BLAKE2B-384

Table 176, BLAKE2B-384 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_384				✓			
CKM_BLAKE2B_384_HMAC_GENE RAL		✓					
CKM_BLAKE2B_384_HMAC		✓					
CKM_BLAKE2B_384_KEY_DERIVE							✓
CKM_BLAKE2B_384_KEY_GEN				✓			

### 6.35.1 Definitions

CKM\_BLAKE2B\_384

CKM\_BLAKE2B\_384\_HMAC

CKM\_BLAKE2B\_384\_HMAC\_GENERAL

CKM\_BLAKE2B\_384\_KEY\_DERIVE

CKM\_BLAKE2B\_384\_KEY\_GEN

CKK\_BLAKE2B\_384\_HMAC

### 6.35.2 BLAKE2B-384 digest

The BLAKE2B-384 mechanism, denoted **CKM\_BLAKE2B\_384**, is a mechanism for message digesting, following the Blake2b Algorithm with a 384-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 177, BLAKE2B-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

### 6.35.3 General-length BLAKE2B-384-HMAC

The general-length BLAKE2B-384-HMAC mechanism, denoted **CKM\_BLAKE2B\_384\_HMAC\_GENERAL**, is the keyed variant of the BLAKE2B-384 hash function and length of the output should be in the range 1-48. The keys it uses are generic secret keys and CKK\_BLAKE2B\_384\_HMAC.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

*Table 178, General-length BLAKE2B-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_384_HMAC	Any	1-48, depending on parameters

The BLAKE2B-384-HMAC mechanism, denoted **CKM\_BLAKE2B\_384\_HMAC**, is a special case of the general-length BLAKE2B-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

### 6.35.5 BLAKE2B-384 key derivation

BLAKE2B-384 key derivation, denoted **CKM\_BLAKE2B\_384\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-384 hash function and the relevant length is 48 bytes.

### 6.35.6 BLAKE2B-384 HMAC key generation

The BLAKE2B-384-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_384\_KEY\_GEN**, is a key generation mechanism for NIST's BLAKE2B-384-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_384\_HMAC** key sizes, in bytes.

## 6.36 BLAKE2B-512

*Table 179, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sub>1</sub>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_512				✓			
CKM_BLAKE2B_512_HMAC_GENE_RAL		✓					
CKM_BLAKE2B_512_HMAC		✓					
CKM_BLAKE2B_512_KEY_DERIVE							✓
CKM_BLAKE2B_512_KEY_GEN				✓			

10516 **6.36.1 Definitions**

10517 CKM\_BLAKE2B\_512  
 10518 CKM\_BLAKE2B\_512\_HMAC  
 10519 CKM\_BLAKE2B\_512\_HMAC\_GENERAL  
 10520 CKM\_BLAKE2B\_512\_KEY\_DERIVE  
 10521 CKM\_BLAKE2B\_512\_KEY\_GEN  
 10522 CKK\_BLAKE2B\_512\_HMAC

10523 **6.36.2 BLAKE2B-512 digest**

10524 The BLAKE2B-512 mechanism, denoted **CKM\_BLAKE2B\_512**, is a mechanism for message digesting,  
 10525 following the Blake2b Algorithm with a 512-bit message digest defined in RFC 7693.

10526 It does not have a parameter.

10527 Constraints on the length of input and output data are summarized in the following table. For single-part  
 10528 digesting, the data and the digest may begin at the same location in memory.

10529 *Table 180, BLAKE2B-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

10530 **6.36.3 General-length BLAKE2B-512-HMAC**

10531 The general-length BLAKE2B-512-HMAC mechanism, denoted  
 10532 **CKM\_BLAKE2B\_512\_HMAC\_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and  
 10533 length of the output should be in the range 1-64. The keys it uses are generic secret keys and  
 10534 CKK\_BLAKE2B\_512\_HMAC.

10535

10536 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired  
 10537 output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures  
 10538 (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

10539 *Table 181, General-length BLAKE2B-512-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_512_HMAC	Any	1-64, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_512_HMAC	Any	1-64, depending on parameters

10540

#### 6.36.4 BLAKE2B-512-HMAC

10541 The BLAKE2B-512-HMAC mechanism, denoted **CKM\_BLAKE2B\_512\_HMAC**, is a special case of the general-length BLAKE2B-512-HMAC mechanism.

10542 It has no parameter, and always produces an output of length 64.

#### 6.36.5 BLAKE2B-512 key derivation

10543 BLAKE2B-512 key derivation, denoted **CKM\_BLAKE2B\_512\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-512 hash function and the relevant length is 64 bytes.

#### 6.36.6 BLAKE2B-512 HMAC key generation

10544 The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_512\_KEY\_GEN**, is a key generation mechanism for NIST's BLAKE2B-512-HMAC.

10545 It does not have a parameter.

10546 The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

10547 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-512-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

10548 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_512\_HMAC** key sizes, in bytes.

10549

### 6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

10550 The mechanisms in this section are for generating keys and IVs for performing password-based encryption. The method used to generate keys and IVs is specified in PKCS #5.

10551 *Table 182, PKCS 5 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_PBE_SHA1_DES3_EDE_CBC					✓		
CKM_PBE_SHA1_DES2_EDE_CBC					✓		
CKM_PBA_SHA1_WITH_SHA1_HMAC					✓		

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_PKCS5_PBKD2					✓		

10566 **6.37.1 Definitions**

10567 Mechanisms:

10568 CKM\_PBE\_SHA1\_DES3\_EDE\_CBC  
 10569 CKM\_PBE\_SHA1\_DES2\_EDE\_CBC  
 10570 CKM\_PKCS5\_PBKD2  
 10571 CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC

10572 **6.37.2 Password-based encryption/authentication mechanism parameters**

10573 ♦ **CK\_PBE\_PARAMS; CK\_PBE\_PARAMS\_PTR**

10574 **CK\_PBE\_PARAMS** is a structure which provides all of the necessary information required by the  
 10575 CKM\_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation  
 10576 mechanisms) and the CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC mechanism. It is defined as follows:

```
10577     typedef struct CK_PBE_PARAMS {
10578         CK_BYTE_PTR          pInitVector;
10579         CK_UTF8CHAR_PTR      pPassword;
10580         CK ULONG              ulPasswordLen;
10581         CK_BYTE_PTR          pSalt;
10582         CK ULONG              ulSaltLen;
10583         CK ULONG              ulIteration;
10584     } CK_PBE_PARAMS;
```

10585 The fields of the structure have the following meanings:

10587 pInitVector	pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;
10589 pPassword	points to the password to be used in the PBE key generation;
10590 ulPasswordLen	length in bytes of the password information;
10591 pSalt	points to the salt to be used in the PBE key generation;
10592 ulSaltLen	length in bytes of the salt information;
10593 ullIteration	number of iterations required for the generation.

10594 **CK\_PBE\_PARAMS\_PTR** is a pointer to a **CK\_PBE\_PARAMS**.

10595    **6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters**

- 10596    ♦ **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE;**  
10597    **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE\_PTR**

10598    **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE** is used to indicate the Pseudo-Random  
10599    Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

10600    `typedef CK ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;`

10601    The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

10603    *Table 183, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions*

PRF Identifier	Value	Parameter Type
CKP_PKCS5_PBKD2_HMAC_SHA1	0x00000001UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_GOSTR3411	0x00000002UL	This PRF uses GOST R34.11-94 hash to produce secret key value. <i>pPrfData</i> should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. <i>ulPrfDataLen</i> holds encoded OID length in bytes. If <i>pPrfData</i> is set to NULL_PTR, then <i>id-GostR3411-94-CryptoProParamSet</i> parameters will be used (RFC 4357, 11.2), and <i>ulPrfDataLen</i> must be 0.
CKP_PKCS5_PBKD2_HMAC_SHA224	0x00000003UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA256	0x00000004UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA384	0x00000005UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA512	0x00000006UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA512_224	0x00000007UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA512_256	0x00000008UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.

10604    **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE\_PTR** is a pointer to a  
10605    **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE**.

10606

10607 ◆ **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE;**  
 10608       **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE\_PTR**

10609 **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE** is used to indicate the source of the salt value when  
 10610 deriving a key using PKCS #5 PBKDF2. It is defined as follows:

10611       **typedef CK ULONG CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE;**

10613 The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined  
 10614 sources along with the corresponding data type for the *pSaltSourceData* field in the  
 10615 **CK\_PKCS5\_PBKD2\_PARAMS2** structure defined below.

10616 *Table 184, PKCS #5 PBKDF2 Key Generation: Salt sources*

Source Identifier	Value	Data Type
CKZ_SALT_SPECIFIED	0x00000001	Array of CK_BYTE containing the value of the salt value.

10617 **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE\_PTR** is a pointer to a  
 10618 **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE**.

10619 ◆ **CK\_PKCS5\_PBKD2\_PARAMS2; CK\_PKCS5\_PBKD2\_PARAMS2\_PTR**

10620 **CK\_PKCS5\_PBKD2\_PARAMS2** is a structure that provides the parameters to the  
 10621 **CKM\_PKCS5\_PBKD2** mechanism. The structure is defined as follows:

```
10622       typedef struct CK_PKCS5_PBKD2_PARAMS2 {
10623           CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE     saltSource;
10624           CK_VOID_PTR                             pSaltSourceData;
10625           CK ULONG                                 ulSaltSourceDataLen;
10626           CK ULONG                                 iterations;
10627           CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE   prf;
10628           CK_VOID_PTR                             pPrfData;
10629           CK ULONG                                 ulPrfDataLen;
10630           CK_UTF8CHAR_PTR                         pPassword;
10631           CK ULONG                                 ulPasswordLen;
10632       } CK_PKCS5_PBKD2_PARAMS2;
10633
```

10634 The fields of the structure have the following meanings:

10635       **saltSource**      source of the salt value

10636       **pSaltSourceData**    data used as the input for the salt source

10637       **ulSaltSourceDataLen**   length of the salt source input

10638       **iterations**        number of iterations to perform when generating each block of  
 10639                           random data

10640       **prf**                pseudo-random function used to generate the key

10641       **pPrfData**        data used as the input for PRF in addition to the salt value

10642       **ulPrfDataLen**      length of the input data for the PRF

10643       **pPassword**       points to the password to be used in the PBE key generation

10644       **ulPasswordLen**     length in bytes of the password information

10645 **CK\_PKCS5\_PBKD2\_PARAMS2\_PTR** is a pointer to a **CK\_PKCS5\_PBKD2\_PARAMS2**.

## 10646 6.37.4 PKCS #5 PBKD2 key generation

10647 PKCS #5 PBKDF2 key generation, denoted **CKM\_PKCS5\_PBKD2**, is a mechanism used for generating  
10648 a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

10649 It has a parameter, a **CK\_PKCS5\_PBKD2\_PARAMS2** structure. The parameter specifies the salt value  
10650 source, pseudo-random function, and iteration count used to generate the new key.

10651 Since this mechanism can be used to generate any type of secret key, new key templates must contain  
10652 the **CKA\_KEY\_TYPE** and **CKA\_VALUE\_LEN** attributes. If the key type has a fixed length the  
10653 **CKA\_VALUE\_LEN** attribute may be omitted.

## 10654 6.38 PKCS #12 password-based encryption/authentication 10655 mechanisms

10656 The mechanisms in this section are for generating keys and IVs for performing password-based  
10657 encryption or authentication. The method used to generate keys and IVs is based on a method that was  
10658 specified in PKCS #12.

10659 We specify here a general method for producing various types of pseudo-random bits from a password,  
10660  $p$ ; a string of salt bits,  $s$ ; and an iteration count,  $c$ . The “type” of pseudo-random bits to be produced is  
10661 identified by an identification byte,  $ID$ , the meaning of which will be discussed later.

10662 Let  $H$  be a hash function built around a compression function  $f: \mathbb{Z}_2^u \times \mathbb{Z}_2^v \rightarrow \mathbb{Z}_2^u$  (that is,  $H$  has a chaining  
10663 variable and output of length  $u$  bits, and the message input to the compression function of  $H$  is  $v$  bits).  
10664 For MD2 and MD5,  $u=128$  and  $v=512$ ; for SHA-1,  $u=160$  and  $v=512$ .

10665 We assume here that  $u$  and  $v$  are both multiples of 8, as are the lengths in bits of the password and salt  
10666 strings and the number  $n$  of pseudo-random bits required. In addition,  $u$  and  $v$  are of course nonzero.

- 10667 1. Construct a string,  $D$  (the “diversifier”), by concatenating  $v/8$  copies of  $ID$ .
- 10668 2. Concatenate copies of the salt together to create a string  $S$  of length  $v\lceil s/v \rceil$  bits (the final copy of the  
10669 salt may be truncated to create  $S$ ). Note that if the salt is the empty string, then so is  $S$ .
- 10670 3. Concatenate copies of the password together to create a string  $P$  of length  $v\lceil p/v \rceil$  bits (the final copy  
10671 of the password may be truncated to create  $P$ ). Note that if the password is the empty string, then so  
10672 is  $P$ .
- 10673 4. Set  $I=S||P$  to be the concatenation of  $S$  and  $P$ .
- 10674 5. Set  $j=\lceil n/u \rceil$ .
- 10675 6. For  $i=1, 2, \dots, j$ , do the following:
  - 10676 a. Set  $A_i=H^c(D||I)$ , the  $c^{\text{th}}$  hash of  $D||I$ . That is, compute the hash of  $D||I$ ; compute the hash of  
10677 that hash; etc.; continue in this fashion until a total of  $c$  hashes have been computed, each on  
10678 the result of the previous hash.
  - 10679 b. Concatenate copies of  $A_i$  to create a string  $B$  of length  $v$  bits (the final copy of  $A_i$  may be  
10680 truncated to create  $B$ ).
  - 10681 c. Treating  $I$  as a concatenation  $I_0, I_1, \dots, I_{k-1}$  of  $v$ -bit blocks, where  $k=\lceil s/v \rceil+\lceil p/v \rceil$ , modify  $I$  by  
10682 setting  $I_j=(I_j+B+1) \bmod 2^v$  for each  $j$ . To perform this addition, treat each  $v$ -bit block as a  
10683 binary number represented most-significant bit first.
- 10684 7. Concatenate  $A_1, A_2, \dots, A_j$  together to form a pseudo-random bit string,  $A$ .
- 10685 8. Use the first  $n$  bits of  $A$  as the output of this entire process.

10686 When the password-based encryption mechanisms presented in this section are used to generate a key  
10687 and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To  
10688 generate a key, the identifier byte  $ID$  is set to the value 1; to generate an IV, the identifier byte  $ID$  is set to  
10689 the value 2.

10690 When the password based authentication mechanism presented in this section is used to generate a key  
10691 from a password, salt, and an iteration count, the above algorithm is used. The identifier byte  $ID$  is set to  
10692 the value 3.

### 6.38.1 SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM\_PBE\_SHA1\_DES3\_EDE\_CBC**, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### 6.38.2 SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM\_PBE\_SHA1\_DES2\_EDE\_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### 6.38.3 SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

## 6.39 SSL

Table 185,SSL Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen - Key / Key Pair	Wrap & Unwrap	Derive
CKM_SSL3_PRE_MASTER_KEY_GEN					✓		
CKM_TLS_PRE_MASTER_KEY_GEN					✓		

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen . Key / Key Pair	Wrap & Unwrap	Derive
CKM_SSL3_MASTER_KEY_DERIVE							✓
CKM_SSL3_MASTER_KEY_DERIVE_DH							✓
CKM_SSL3_KEY_AND_MAC_DERIVE							✓
CKM_SSL3_MD5_MAC		✓					
CKM_SSL3_SHA1_MAC		✓					

10728 **6.39.1 Definitions**

10729 Mechanisms:

10730 CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN  
 10731 CKM\_TLS\_PRE\_MASTER\_KEY\_GEN  
 10732 CKM\_SSL3\_MASTER\_KEY\_DERIVE  
 10733 CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE  
 10734 CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH  
 10735 CKM\_SSL3\_MD5\_MAC  
 10736 CKM\_SSL3\_SHA1\_MAC

10737 **6.39.2 SSL mechanism parameters**

10738 ♦ **CK\_SSL3\_RANDOM\_DATA**

10739 **CK\_SSL3\_RANDOM\_DATA** is a structure which provides information about the random data of a client  
 10740 and a server in an SSL context. This structure is used by both the **CKM\_SSL3\_MASTER\_KEY\_DERIVE**  
 10741 and the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanisms. It is defined as follows:

```
10742     typedef struct CK_SSL3_RANDOM_DATA {
10743         CK_BYTE_PTR pClientRandom;
10744         CK ULONG ulClientRandomLen;
10745         CK_BYTE_PTR pServerRandom;
10746         CK ULONG ulServerRandomLen;
10747     } CK_SSL3_RANDOM_DATA;
```

10748

10749 The fields of the structure have the following meanings:

10750           pClientRandom	pointer to the client's random data
10751           ulClientRandomLen	length in bytes of the client's random data
10752           pServerRandom	pointer to the server's random data
10753           ulServerRandomLen	length in bytes of the server's random data

10754 ◆ **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS;**  
10755     **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR**

10756 **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
10757 **CKM\_SSL3\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

```
10758     typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {  
10759         CK_SSL3_RANDOM_DATA RandomInfo;  
10760         CK_VERSION_PTR pVersion;  
10761     } CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

10762 The fields of the structure have the following meanings:

10764     RandomInfo     client's and server's random data information.

10765     pVersion       pointer to a **CK\_VERSION** structure which receives the SSL  
10766                    protocol version information

10767 **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a  
10768 **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS**.

10769 ◆ **CK\_SSL3\_KEY\_MAT\_OUT; CK\_SSL3\_KEY\_MAT\_OUT\_PTR**

10770 **CK\_SSL3\_KEY\_MAT\_OUT** is a structure that contains the resulting key handles and initialization vectors  
10771 after performing a C\_DeriveKey function with the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism. It  
10772 is defined as follows:

```
10773     typedef struct CK_SSL3_KEY_MAT_OUT {  
10774         CK_OBJECT_HANDLE hClientMacSecret;  
10775         CK_OBJECT_HANDLE hServerMacSecret;  
10776         CK_OBJECT_HANDLE hClientKey;  
10777         CK_OBJECT_HANDLE hServerKey;  
10778         CK_BYTE_PTR pIVClient;  
10779         CK_BYTE_PTR pIVServer;  
10780     } CK_SSL3_KEY_MAT_OUT;
```

10781 The fields of the structure have the following meanings:

10783     hClientMacSecret     key handle for the resulting Client MAC Secret key

10784     hServerMacSecret     key handle for the resulting Server MAC Secret key

10785     hClientKey         key handle for the resulting Client Secret key

10786     hServerKey         key handle for the resulting Server Secret key

10787     pIVClient         pointer to a location which receives the initialization vector (IV)  
10788                    created for the client (if any)

10789     pIVServer         pointer to a location which receives the initialization vector (IV)  
10790                    created for the server (if any)

10791 **CK\_SSL3\_KEY\_MAT\_OUT\_PTR** is a pointer to a **CK\_SSL3\_KEY\_MAT\_OUT**.

10792 ◆ **CK\_SSL3\_KEY\_MAT\_PARAMS; CK\_SSL3\_KEY\_MAT\_PARAMS\_PTR**

10793 **CK\_SSL3\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the  
10794 **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

```

10795     typedef struct CK_SSL3_KEY_MAT_PARAMS {
10796         CK ULONG             ulMacSizeInBits;
10797         CK ULONG             ulKeySizeInBits;
10798         CK ULONG             ulIVSizeInBits;
10799         CK_BBOOL            bIsExport;
10800         CK_SSL3_RANDOM_DATA RandomInfo;
10801         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
10802     } CK_SSL3_KEY_MAT_PARAMS;

```

10803

10804 The fields of the structure have the following meanings:

10805	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the protocol handshake phase
10806	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the protocol handshake phase
10807	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0
10808	bIsExport	a Boolean value which indicates whether the keys have to be derived for an export version of the protocol
10809	RandomInfo	client's and server's random data information.
10810	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structures which receives the handles for the keys generated and the IVs

10811 **CK\_SSL3\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_SSL3\_KEY\_MAT\_PARAMS**.

### 10812 6.39.3 Pre-master key generation

10813 Pre-master key generation in SSL 3.0, denoted **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN**, is a mechanism which generates a 48-byte generic secret key. It is used to produce the "pre\_master" key used in SSL version 3.0 for RSA-like cipher suites.

10814 It has one parameter, a **CK\_VERSION** structure, which provides the client's SSL version number.

10815 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

10816 The template sent along with this mechanism during a **C\_GenerateKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

10817 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

10818 **CKM\_TLS\_PRE\_MASTER\_KEY\_GEN** has identical functionality as **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN**. It exists only for historical reasons, please use **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN** instead.

### 10819 6.39.4 Master key derivation

10820 Master key derivation in SSL 3.0, denoted **CKM\_SSL3\_MASTER\_KEY\_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master\_secret" key used in the SSL protocol from the "pre\_master" key. This mechanism returns the value of the client version, which is built into the "pre\_master" key as well as a handle to the derived "master\_secret" key.

10840 It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the  
10841 passing of random data to the token as well as the returning of the protocol version number which is part  
10842 of the pre-master key. This structure is defined in Section 6.39.

10843 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10844 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
10845 be specified in the template; otherwise they are assigned default values.

10846 The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
10847 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
10848 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to  
10849 specify any of them.

10850 This mechanism has the following rules about key sensitivity and extractability:

- 10851 • The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
10852 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
10853 default value.
- 10854 • If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
10855 will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
10856 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
10857 **CKA\_SENSITIVE** attribute.
- 10858 • Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
10859 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
10860 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
10861 value from its **CKA\_EXTRACTABLE** attribute.

10862 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure  
10863 both indicate 48 bytes.

10864 Note that the **CK\_VERSION** structure pointed to by the **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS**  
10865 structure's *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns,  
10866 this structure will hold the SSL version associated with the supplied pre\_master key.

10867 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre\_master" secret with an  
10868 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher  
10869 suites.

### 10870 6.39.5 Master key derivation for Diffie-Hellman

10871 Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH**,  
10872 is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic  
10873 secret key. It is used to produce the "master\_secret" key used in the SSL protocol from the "pre\_master"  
10874 key.

10875 It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the  
10876 passing of random data to the token. This structure is defined in Section 6.39. The *pVersion* field of the  
10877 structure must be set to NULL\_PTR since the version number is not embedded in the "pre\_master" key  
10878 as it is for RSA-like cipher suites.

10879 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
10880 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
10881 be specified in the template, or else are assigned default values.

10882 The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
10883 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
10884 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to  
10885 specify any of them.

10886 This mechanism has the following rules about key sensitivity and extractability:

- 10887 • The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
10888 be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
10889 default value.

- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte "pre\_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

### 6.39.6 Key and MAC derivation

Key, MAC and IV derivation in SSL 3.0, denoted **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master\_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK\_SSL3\_KEY\_MAT\_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 6.39.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.

The two MACing keys ("client\_write\_MAC\_secret" and "server\_write\_MAC\_secret") are always given a type of **CKK\_GENERIC\_SECRET**. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client\_write\_key" and "server\_write\_key") are typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the *ullVSizeInBits* field of the **CK\_SSL3\_KEY\_MAT\_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ullVSizeInBits* field.

All four keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the four key handle fields in the **CK\_SSL3\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK\_SSL3\_KEY\_MAT\_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism returns all of its key handles in the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a **NULPTR**.

10940 If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the  
10941 token.

### 6.39.7 MD5 MACing in SSL 3.0

10943 MD5 MACing in SSL3.0, denoted **CKM\_SSL3\_MD5\_MAC**, is a mechanism for single- and multiple-part  
10944 signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This  
10945 technique is very similar to the HMAC technique.

10946 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which specifies the length in bytes of the  
10947 signatures produced by this mechanism.

10948 Constraints on key types and the length of input and output data are summarized in the following table:

10949 *Table 186, MD5 MACing in SSL 3.0: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

10950 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10951 specify the supported range of generic secret key sizes, in bits.

### 6.39.8 SHA-1 MACing in SSL 3.0

10953 SHA-1 MACing in SSL3.0, denoted **CKM\_SSL3\_SHA1\_MAC**, is a mechanism for single- and multiple-part  
10954 signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This  
10955 technique is very similar to the HMAC technique.

10956 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which specifies the length in bytes of the  
10957 signatures produced by this mechanism.

10958 Constraints on key types and the length of input and output data are summarized in the following table:

10959 *Table 187, SHA-1 MACing in SSL 3.0: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

10960 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
10961 specify the supported range of generic secret key sizes, in bits.

## 6.40 TLS 1.2 Mechanisms

10963 Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2  
10964 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS  
10965 PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2  
10966 mechanisms contains a new value in the parameters structure to specify the hash function.

10967 This section also specifies **CKM\_TLS12\_MAC** which should be used in place of **CKM\_TLS\_PRF** to  
10968 calculate the verify\_data in the TLS "finished" message.

10969 This section also specifies **CKM\_TLS\_KDF** that can be used in place of **CKM\_TLS\_PRF** to implement  
10970 key material exporters.

10971

10972 *Table 188, TLS 1.2 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_TLS12_MASTER_KEY_DERIVE							✓
CKM_TLS12_MASTER_KEY_DERIVE_DH							✓
CKM_TLS12_KEY_AND_MAC_DERIVE							✓
CKM_TLS12_KEY_SAFE_DERIVE							✓
CKM_TLS_KDF							✓
CKM_TLS12_MAC		✓					
CKM_TLS12_KDF							✓

## 6.40.1 Definitions

Mechanisms:

```
CKM_TLS12_MASTER_KEY_DERIVE
CKM_TLS12_MASTER_KEY_DERIVE_DH
CKM_TLS12_KEY_AND_MAC_DERIVE
CKM_TLS12_KEY_SAFE_DERIVE
CKM_TLS_KDF
CKM_TLS12_MAC
CKM_TLS12_KDF
```

## 6.40.2 TLS 1.2 mechanism parameters

- ◆ **CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS;**
- CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR**

**CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS12\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_TLS12_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_VERSION_PTR pVersion;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TLS12_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

RandomInfo client's and server's random data information.

pVersion pointer to a **CK\_VERSION** structure which receives the SSL protocol version information

prfHashMechanism base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

**CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS**.

11002 ◆ **CK\_TLS12\_KEY\_MAT\_PARAMS; CK\_TLS12\_KEY\_MAT\_PARAMS\_PTR**

11003 **CK\_TLS12\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the  
11004 **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

```
11005     typedef struct CK_TLS12_KEY_MAT_PARAMS {  
11006         CK ULONG ulMacSizeInBits;  
11007         CK ULONG ulKeySizeInBits;  
11008         CK ULONG ullIVSizeInBits;  
11009         CK_BBOOL bIsExport;  
11010         CK_SSL3_RANDOM_DATA RandomInfo;  
11011         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
11012         CK_MECHANISM_TYPE prfHashMechanism;  
11013     } CK_TLS12_KEY_MAT_PARAMS;
```

11014

11015 The fields of the structure have the following meanings:

11016	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the 11017 protocol handshake phase. If no MAC key is required, the length 11018 should be set to 0.
11019	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the 11020 protocol handshake phase
11021	ullIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol 11022 handshake phase. If no IV is required, the length should be set to 0
11023	bIsExport	must be set to CK_FALSE because export cipher suites must not be 11024 used in TLS 1.1 and later.
11025	RandomInfo	client's and server's random data information.
11026	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structures which receives 11027 the handles for the keys generated and the IVs
11028	prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to 11029 derive the master key.

11030 **CK\_TLS12\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_TLS12\_KEY\_MAT\_PARAMS**.

11031 ◆ **CK\_TLS\_KDF\_PARAMS; CK\_TLS\_KDF\_PARAMS\_PTR**

11032 **CK\_TLS\_KDF\_PARAMS** is a structure that provides the parameters to the CKM\_TLS\_KDF mechanism.  
11033 It is defined as follows:

```
11034     typedef struct CK_TLS_KDF_PARAMS {  
11035         CK_MECHANISM_TYPE prfMechanism;  
11036         CK_BYTE_PTR pLabel;  
11037         CK ULONG ulLabelLength;  
11038         CK_SSL3_RANDOM_DATA RandomInfo;  
11039         CK_BYTE_PTR pContextData;  
11040         CK ULONG ulContextDataLength;  
11041     } CK_TLS_KDF_PARAMS;
```

11042

11043 The fields of the structure have the following meanings:

11044	prfMechanism	the hash mechanism used in the TLS1.2 PRF construct or 11045 CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
-------	--------------	---------------------------------------------------------------------------------------------------------------------------

11046                    pLabel      a pointer to the label for this key derivation  
 11047                    ulLabelLength      length of the label in bytes  
 11048                    RandomInfo      the random data for the key derivation  
 11049                    pContextData      a pointer to the context data for this key derivation. NULL\_PTR if not  
 11050                    present  
 11051                    ulContextDataLength      length of the context data in bytes. 0 if not present.  
 11052      **CK\_TLS\_KDF\_PARAMS\_PTR** is a pointer to a **CK\_TLS\_KDF\_PARAMS**.

#### 11053      ♦ **CK\_TLS\_MAC\_PARAMS; CK\_TLS\_MAC\_PARAMS\_PTR**

11054      **CK\_TLS\_MAC\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS\_MAC**  
 11055      mechanism. It is defined as follows:

```

    11056      typedef struct CK_TLS_MAC_PARAMS {
    11057         CK_MECHANISM_TYPE prfHashMechanism;
    11058         CK ULONG ulMacLength;
    11059         CK ULONG ulServerOrClient;
    11060      } CK_TLS_MAC_PARAMS;
  
```

11061      The fields of the structure have the following meanings:

11063                    prfHashMechanism 11064	the hash mechanism used in the TLS12 PRF construct or CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
11065                    ulMacLength 11066	the length of the MAC tag required or offered. Always 12 octets in TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a longer value in TLS1.2.
11068                    ulServerOrClient 11069	1 to use the label "server finished", 2 to use the label "client finished". All other values are invalid.

11070      **CK\_TLS\_MAC\_PARAMS\_PTR** is a pointer to a **CK\_TLS\_MAC\_PARAMS**.

11071

#### 11072      ♦ **CK\_TLS\_PRF\_PARAMS; CK\_TLS\_PRF\_PARAMS\_PTR**

11073      **CK\_TLS\_PRF\_PARAMS** is a structure, which provides the parameters to the **CKM\_TLS\_PRF**  
 11074      mechanism. It is defined as follows:

```

    11075      typedef struct CK_TLS_PRF_PARAMS {
    11076         CK_BYTE_PTR      pSeed;
    11077         CK ULONG      ulSeedLen;
    11078         CK_BYTE_PTR      pLabel;
    11079         CK ULONG      ullLabelLen;
    11080         CK_BYTE_PTR      pOutput;
    11081         CK ULONG_PTR      pulOutputLen;
    11082      } CK_TLS_PRF_PARAMS;
  
```

11083

11084      The fields of the structure have the following meanings:

11085                    pSeed 11086                    ulSeedLen 11087                    pLabel	<i>pointer to the input seed</i> <i>length in bytes of the input seed</i> <i>pointer to the identifying label</i>
---------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------

11088                   ulLabelLen     *length in bytes of the identifying label*  
 11089                   pOutput       *pointer receiving the output of the operation*  
 11090                   pulOutputLen   *pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output*  
 11091  
 11092  
 11093 CK\_TLS\_PRF\_PARAMS\_PTR is a pointer to a CK\_TLS\_PRF\_PARAMS.

### 11094 6.40.3 TLS MAC

11095 The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces  
 11096 the use of the **CKM\_TLS\_PRF** function for TLS1.0 and 1.1 and that mechanism is deprecated.

11097 **CKM\_TLS\_MAC** takes a parameter of CK\_TLS\_MAC\_PARAMS. To use this mechanism with TLS1.0  
 11098 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note:  
 11099 Although **CKM\_TLS\_PRF** is deprecated as a mechanism for C\_DeriveKey, the manifest value is retained  
 11100 for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

11101 In TLS1.0 and 1.1 the "finished" message verify\_data (i.e. the output signature from the MAC mechanism)  
 11102 is always 12 bytes. In TLS1.2 the "finished" message verify\_data is a minimum of 12 bytes, defaults to 12  
 11103 bytes, but may be negotiated to longer length.

11104 *Table 189, General-length TLS MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	>=12 bytes
C_Verify	generic secret	any	>=12 bytes

11105

### 11106 6.40.4 Master key derivation

11107 Master key derivation in TLS 1.0, denoted **CKM\_TLS\_MASTER\_KEY\_DERIVE**, is a mechanism used to  
 11108 derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the  
 11109 "master\_secret" key used in the TLS protocol from the "pre\_master" key. This mechanism returns the  
 11110 value of the client version, which is built into the "pre\_master" key as well as a handle to the derived  
 11111 "master\_secret" key.

11112 It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the  
 11113 passing of random data to the token as well as the returning of the protocol version number which is part  
 11114 of the pre-master key. This structure is defined in Section 6.39.

11115 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 11116 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
 11117 be specified in the template, or else are assigned default values.

11118 The mechanism also contributes the **CKA\_ALLOWED\_MECHANISMS** attribute consisting only of  
 11119 **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**, **CKM\_TLS12\_KEY\_SAFE\_DERIVE**, **CKM\_TLS12\_KDF** and  
 11120 **CKM\_TLS12\_MAC**.

11121 The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
 11122 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
 11123 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to  
 11124 specify any of them.

11125 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the

11131        derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
11132        **CKA\_SENSITIVE** attribute.

- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

11137      For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure  
11138      both indicate 48 bytes.

11139      Note that the **CK\_VERSION** structure pointed to by the **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS**  
11140      structure's *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns,  
11141      this structure will hold the SSL version associated with the supplied pre\_master key.

11142      Note that this mechanism is only useable for cipher suites that use a 48-byte "pre\_master" secret with an  
11143      embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher  
11144      suites.

#### 11145      6.40.5 Master key derivation for Diffie-Hellman

11146      Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM\_TLS\_MASTER\_KEY\_DERIVE\_DH**, is  
11147      a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret  
11148      key. It is used to produce the "master\_secret" key used in the TLS protocol from the "pre\_master" key.

11149      It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the  
11150      passing of random data to the token. This structure is defined in Section 6.39. The *pVersion* field of the  
11151      structure must be set to NULL\_PTR since the version number is not embedded in the "pre\_master" key  
11152      as it is for RSA-like cipher suites.

11153      The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
11154      key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
11155      be specified in the template, or else are assigned default values.

11156      The mechanism also contributes the **CKA\_ALLOWED\_MECHANISMS** attribute consisting only of  
11157      **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**, **CKM\_TLS12\_KEY\_SAFE\_DERIVE**, **CKM\_TLS12\_KDF** and  
11158      **CKM\_TLS12\_MAC**.

11159      The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
11160      class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
11161      attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to  
11162      specify any of them.

11163      This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both  
11165      be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some  
11166      default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
11168      will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
11169      derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
11170      **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
11173      CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
11174      value from its **CKA\_EXTRACTABLE** attribute.

11175      For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure  
11176      both indicate 48 bytes.

11177      Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte  
11178      "pre\_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but  
11179      excludes the RSA cipher suites.

11180 **6.40.6 Key and MAC derivation**

11181 Key, MAC and IV derivation in TLS 1.0, denoted **CKM\_TLS\_KEY\_AND\_MAC\_DERIVE**, is a mechanism  
11182 used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the  
11183 "master\_secret" key and random data. This mechanism returns the key handles for the keys generated in  
11184 the process, as well as the IVs created.

11185 It has a parameter, a **CK\_SSL3\_KEY\_MAT\_PARAMS** structure, which allows for the passing of random  
11186 data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a  
11187 structure which receives the handles and IVs which were generated. This structure is defined in Section  
11188 6.39.

11189 This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs  
11190 are requested by the caller) back to the caller. The keys are all given an object class of  
11191 **CKO\_SECRET\_KEY**.

11192 The two MACing keys ("client\_write\_MAC\_secret" and "server\_write\_MAC\_secret") (if present) are  
11193 always given a type of **CKK\_GENERIC\_SECRET**. They are flagged as valid for signing and verification.

11194 The other two keys ("client\_write\_key" and "server\_write\_key") are typed according to information found  
11195 in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, they are  
11196 flagged as valid for encryption, decryption, and derivation operations.

11197 For **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**, IVs will be generated and returned if the *ullVSizeInBits*  
11198 field of the **CK\_SSL3\_KEY\_MAT\_PARAMS** field has a nonzero value. If they are generated, their length  
11199 in bits will agree with the value in the *ullVSizeInBits* field.

11200

11201 Note Well: CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE produces both private (key) and public (IV)  
11202 data. It is possible to "leak" private data by the simple expedient of decreasing the length of  
11203 private data requested. E.g. Setting *ulMacSizeInBits* and *ulKeySizeInBits* to 0 (or other lengths  
11204 less than the key size) will result in the private key data being placed in the destination  
11205 designated for the IV's. Repeated calls with the same master key and same RandomInfo but with  
11206 differing lengths for the private key material will result in different data being leaked.<

11207

11208 All four keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**,  
11209 **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template  
11210 provided to **C\_DeriveKey** may not specify values for any of these attributes which differ from those held  
11211 by the base key.

11212 Note that the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS**  
11213 structure's *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the four  
11214 key handle fields in the **CK\_SSL3\_KEY\_MAT\_OUT** structure will be modified to hold handles to the  
11215 newly-created keys; in addition, the buffers pointed to by the **CK\_SSL3\_KEY\_MAT\_OUT** structure's  
11216 *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller).  
11217 Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

11218 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.  
11219 For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a  
11220 successful completion. However, since the **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE** mechanism returns  
11221 all of its key handles in the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the  
11222 **CK\_SSL3\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*  
11223 passed to **C\_DeriveKey** is unnecessary, and should be a **NULL\_PTR**.

11224 If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the  
11225 token.

11226 **6.40.7 CKM\_TLS12\_KEY\_SAFE\_DERIVE**

11227 **CKM\_TLS12\_KEY\_SAFE\_DERIVE** is identical to **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE** except that it  
11228 shall never produce IV data, and the *ullVSizeInBits* field of **CK\_TLS12\_KEY\_MAT\_PARAMS** is ignored

11229 and treated as 0. All of the other conditions and behavior described for  
11230 CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE, with the exception of the black box warning, apply to this  
11231 mechanism.

11232 CKM\_TLS12\_KEY\_SAFE\_DERIVE is provided as a separate mechanism to allow a client to control the  
11233 export of IV material (and possible leaking of key material) through the use of the  
11234 CKA\_ALLOWED\_MECHANISMS key attribute.

#### 11235 **6.40.8 Generic Key Derivation using the TLS PRF**

11236 **CKM\_TLS\_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF  
11237 function to produce additional key material for protocols that want to leverage the TLS key negotiation  
11238 mechanism. **CKM\_TLS\_KDF** has a parameter of **CK\_TLS\_KDF\_PARAMS**. If the protocol using this  
11239 mechanism does not use context information, the *pContextData* field shall be set to **NULL\_PTR** and the  
11240 *ulContextDataLength* field shall be set to 0.

11241 To use this mechanism with TLS1.0 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in  
11242 place of a hash mechanism. Note: Although **CKM\_TLS\_PRF** is deprecated as a mechanism for  
11243 **C\_DeriveKey**, the manifest value is retained for use with this mechanism to indicate the use of the  
11244 TLS1.0/1.1 Pseudo-random function.

11245 This mechanism can be used to derive multiple keys (e.g. similar to  
11246 **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**) by first deriving the key stream as a **CKK\_GENERIC\_SECRET**  
11247 of the necessary length and doing subsequent derives against that derived key using the  
11248 **CKM\_EXTRACT\_KEY\_FROM\_KEY** mechanism to split the key stream into the actual operational keys.

11249 The mechanism should not be used with the labels defined for use with TLS, but the token does not  
11250 enforce this behavior.

11251 This mechanism has the following rules about key sensitivity and extractability:

- 11252 • If the original key has its **CKA\_SENSITIVE** attribute set to **CK\_TRUE**, so does the derived key. If not,  
11253 then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from the  
11254 original key.
- 11255 • Similarly, if the original key has its **CKA\_EXTRACTABLE** attribute set to **CK\_FALSE**, so does the  
11256 derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the  
11257 supplied template or from the original key.
- 11258 • The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to **CK\_TRUE** if and only if the original  
11259 key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to **CK\_TRUE**.
- 11260 • Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to **CK\_TRUE** if and only if  
11261 the original key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to **CK\_TRUE**.

#### 11262 **6.40.9 Generic Key Derivation using the TLS12 PRF**

11263 **CKM\_TLS12\_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF  
11264 function to produce additional key material for protocols that want to leverage the TLS key negotiation  
11265 mechanism. **CKM\_TLS12\_KDF** has a parameter of **CK\_TLS\_KDF\_PARAMS**. If the protocol using this  
11266 mechanism does not use context information, the *pContextData* field shall be set to **NULL\_PTR** and the  
11267 *ulContextDataLength* field shall be set to 0.

11268 To use this mechanism with TLS1.0 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in  
11269 place of a hash mechanism. Note: Although **CKM\_TLS\_PRF** is deprecated as a mechanism for  
11270 **C\_DeriveKey**, the manifest value is retained for use with this mechanism to indicate the use of the  
11271 TLS1.0/1.1 Pseudo-random function.

11272 This mechanism can be used to derive multiple keys (e.g. similar to  
11273 **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**) by first deriving the key stream as a **CKK\_GENERIC\_SECRET**  
11274 of the necessary length and doing subsequent derives against that derived key stream using the  
11275 **CKM\_EXTRACT\_KEY\_FROM\_KEY** mechanism to split the key stream into the actual operational keys.

- 11276 The mechanism should not be used with the labels defined for use with TLS, but the token does not  
 11277 enforce this behavior.
- 11278 This mechanism has the following rules about key sensitivity and extractability:
- If the original key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from the original key.
  - Similarly, if the original key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from the original key.
  - The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
  - Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## 6.41 WTLS

11290 Details can be found in [WTLS].

11291 When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue  
 11292 that there would be no need to have distinct handling of the client and server side of the handshake.  
 11293 However, since in WTLS the server and client use different sequence numbers, there could be instances  
 11294 (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and  
 11295 server side differ, and hence this motivates the introduced split.

11296  
 11297 *Table 190, WTLS Mechanisms vs. Functions*

Mechanism	Functions						
	Encry pt & Decry pt	Sign & Verif y	SR & VR <sup>1</sup>	Dige st	Ge n. Key / Key Pair	Wrap & Unwra p	Deriv e
CKM_WTLS_PRE_MASTER_KEY_GEN					✓		
CKM_WTLS_MASTER_KEY_DERIVE							✓
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC							✓
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_PRF							✓

### 6.41.1 Definitions

11298 Mechanisms:  
 11299     CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN  
 11300     CKM\_WTLS\_MASTER\_KEY\_DERIVE  
 11301     CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC

```
11303      CKM_WTLS_PRF
11304      CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
11305      CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE
```

## 11306 6.41.2 WTLS mechanism parameters

### 11307 ♦ CK\_WTLS\_RANDOM\_DATA; CK\_WTLS\_RANDOM\_DATA\_PTR

11308 **CK\_WTLS\_RANDOM\_DATA** is a structure, which provides information about the random data of a client  
11309 and a server in a WTLS context. This structure is used by the **CKM\_WTLS\_MASTER\_KEY\_DERIVE**  
11310 mechanism. It is defined as follows:

```
11311     typedef struct CK_WTLS_RANDOM_DATA {
11312         CK_BYTE_PTR pClientRandom;
11313         CK ULONG ulClientRandomLen;
11314         CK_BYTE_PTR pServerRandom;
11315         CK ULONG ulServerRandomLen;
11316     } CK_WTLS_RANDOM_DATA;
```

11317

11318 The fields of the structure have the following meanings:

11319	pClientRandom	pointer to the client's random data
11320	pClientRandomLen	length in bytes of the client's random data
11321	pServerRaondom	pointer to the server's random data
11322	ulServerRandomLen	length in bytes of the server's random data

11323 **CK\_WTLS\_RANDOM\_DATA\_PTR** is a pointer to a **CK\_WTLS\_RANDOM\_DATA**.

### 11324 ♦ CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS; 11325 CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR

11326 **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure, which provides the parameters to the  
11327 **CKM\_WTLS\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

```
11328     typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
11329         CK_MECHANISM_TYPE DigestMechanism;
11330         CK_WTLS_RANDOM_DATA RandomInfo;
11331         CK_BYTE_PTR pVersion;
11332     } CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

11333

11334 The fields of the structure have the following meanings:

11335	DigestMechanism	the mechanism type of the digest mechanism to be used (possible 11336 types can be found in [WTLS])
11337	RandomInfo	Client's and server's random data information
11338	pVersion	pointer to a <b>CK_BYTE</b> which receives the WTLS protocol version 11339 information

11340 **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a  
11341 **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS**.

11342 ◆ **CK\_WTLS\_PRF\_PARAMS; CK\_WTLS\_PRF\_PARAMS\_PTR**

11343 **CK\_WTLS\_PRF\_PARAMS** is a structure, which provides the parameters to the **CKM\_WTLS\_PRF**  
11344 mechanism. It is defined as follows:

```
11345     typedef struct CK_WTLS_PRF_PARAMS {  
11346         CK_MECHANISM_TYPE DigestMechanism;  
11347         CK_BYTE_PTR pSeed;  
11348         CK ULONG ulSeedLen;  
11349         CK_BYTE_PTR pLabel;  
11350         CK ULONG ulLabelLen;  
11351         CK_BYTE_PTR pOutput;  
11352         CK ULONG_PTR pulOutputLen;  
11353     } CK_WTLS_PRF_PARAMS;
```

11354

11355 The fields of the structure have the following meanings:

Digest Mechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
pSeed	pointer to the input seed
ulSeedLen	length in bytes of the input seed
pLabel	pointer to the identifying label
ulLabelLen	length in bytes of the identifying label
pOutput	pointer receiving the output of the operation
pulOutputLen	pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

11366 **CK\_WTLS\_PRF\_PARAMS\_PTR** is a pointer to a **CK\_WTLS\_PRF\_PARAMS**.

11367 ◆ **CK\_WTLS\_KEY\_MAT\_OUT; CK\_WTLS\_KEY\_MAT\_OUT\_PTR**

11368 **CK\_WTLS\_KEY\_MAT\_OUT** is a structure that contains the resulting key handles and initialization  
11369 vectors after performing a C\_DeriveKey function with the  
11370 **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE** or with the  
11371 **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

```
11372     typedef struct CK_WTLS_KEY_MAT_OUT {  
11373         CK_OBJECT_HANDLE hMacSecret;  
11374         CK_OBJECT_HANDLE hKey;  
11375         CK_BYTE_PTR pIV;  
11376     } CK_WTLS_KEY_MAT_OUT;
```

11377

11378 The fields of the structure have the following meanings:

hMacSecret	Key handle for the resulting MAC secret key
hKey	Key handle for the resulting secret key
pIV	Pointer to a location which receives the initialization vector (IV) created (if any)

11383 **CK\_WTLS\_KEY\_MAT\_OUT\_PTR** is a pointer to a **CK\_WTLS\_KEY\_MAT\_OUT**.

11384 ◆ **CK\_WTLS\_KEY\_MAT\_PARAMS; CK\_WTLS\_KEY\_MAT\_PARAMS\_PTR**

11385 **CK\_WTLS\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the  
11386 **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE** and the  
11387 **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanisms. It is defined as follows:

```
11388     typedef struct CK_WTLS_KEY_MAT_PARAMS {  
11389         CK_MECHANISM_TYPE          DigestMechanism;  
11390         CK ULONG                  ulMacSizeInBits;  
11391         CK ULONG                  ulKeySizeInBits;  
11392         CK ULONG                  ulIVSizeInBits;  
11393         CK ULONG                  ulSequenceNumber;  
11394         CK_BBOOL                 bIsExport;  
11395         CK_WTLS_RANDOM_DATA       RandomInfo;  
11396         CK_WTLS_KEY_MAT_OUT_PTR  pReturnedKeyMaterial;  
11397     } CK_WTLS_KEY_MAT_PARAMS;
```

11398

11399 The fields of the structure have the following meanings:

11400	Digest Mechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
11402	ulMaxSizeInBits	the length (in bits) of the MACing key agreed upon during the protocol handshake phase
11404	ulKeySizeInBits	the length (in bits) of the secret key agreed upon during the handshake phase
11406	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0.
11408	ulSequenceNumber	the current sequence number used for records sent by the client and server respectively
11410	bIsExport	a boolean value which indicates whether the keys have to be derived for an export version of the protocol. If this value is true (i.e., the keys are exportable) then ulKeySizeInBits is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C_DeriveKey function call (either the <b>CKA_KEY_TYPE</b> or the <b>CKA_VALUE_LEN</b> attribute).
11417	RandomInfo	client's and server's random data information
11418	pReturnedKeyMaterial	points to a <b>CK_WTLS_KEY_MAT_OUT</b> structure which receives the handles for the keys generated and the IV

11420 **CK\_WTLS\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_WTLS\_KEY\_MAT\_PARAMS**.

11421 **6.41.3 Pre master secret key generation for RSA key exchange suite**

11422 Pre master secret key generation for the RSA key exchange suite in WTLS denoted  
11423 **CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN**, is a mechanism, which generates a variable length secret key.  
11424 It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This  
11425 mechanism returns a handle to the pre master secret key.

11426 It has one parameter, a **CK\_BYTE**, which provides the client's WTLS version.

11427 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE** and **CKA\_VALUE** attributes to the new  
11428 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
11429 be specified in the template, or else are assigned default values.

11430 The template sent along with this mechanism during a **C\_GenerateKey** call may indicate that the object  
11431 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
11432 attribute indicates the length of the pre master secret key.

11433 For this mechanism, the **ulMinKeySize** field of the **CK\_MECHANISM\_INFO** structure shall indicate 20  
11434 bytes.

#### 11435 **6.41.4 Master secret key derivation**

11436 Master secret derivation in WTLS, denoted **CKM\_WTLS\_MASTER\_KEY\_DERIVE**, is a mechanism used  
11437 to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master  
11438 secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client  
11439 version, which is built into the pre master secret key as well as a handle to the derived master secret key.

11440 It has a parameter, a **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for passing  
11441 the mechanism type of the digest mechanism to be used as well as the passing of random data to the  
11442 token as well as the returning of the protocol version number which is part of the pre master secret key.

11443 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
11444 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
11445 be specified in the template, or else are assigned default values.

11446 The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
11447 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
11448 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to  
11449 specify any of them.

11450 This mechanism has the following rules about key sensitivity and extractability:

11451 The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be  
11452 specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default  
11453 value.

11454 If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will  
11455 as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived  
11456 key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.

11457 Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
11458 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE,  
11459 then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the opposite value from its  
11460 **CKA\_EXTRACTABLE** attribute.

11461 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK\_MECHANISM\_INFO** structure  
11462 both indicate 20 bytes.

11463 Note that the **CK\_BYTE** pointed to by the **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure's  
11464 *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns, this byte will  
11465 hold the WTLS version associated with the supplied pre master secret key.

11466 Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret  
11467 key with an embedded version number. This includes the RSA key exchange suites, but excludes the  
11468 Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

#### 11469 **6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve 11470 Cryptography**

11471 Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted  
11472 **CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC**, is a mechanism used to derive a 20 byte generic  
11473 secret key from variable length secret key. It is used to produce the master secret key used in WTLS from  
11474 the pre master secret key. This mechanism returns a handle to the derived master secret key.

11475 It has a parameter, a **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the  
11476 passing of the mechanism type of the digest mechanism to be used as well as random data to the token.  
11477 The *pVersion* field of the structure must be set to NULL\_PTR since the version number is not embedded  
11478 in the pre master secret key as it is for RSA-like key exchange suites.

11479 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
11480 key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may  
11481 be specified in the template, or else are assigned default values.  
11482 The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object  
11483 class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN**  
11484 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to  
11485 specify any of them.  
11486 This mechanism has the following rules about key sensitivity and extractability:  
11487 The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be  
11488 specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default  
11489 value.  
11490 If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will  
11491 as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived  
11492 key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.  
11493 Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
11494 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE,  
11495 then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its  
11496 **CKA\_EXTRACTABLE** attribute.  
11497 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure  
11498 both indicate 20 bytes.  
11499 Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte  
11500 pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic  
11501 Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

## 11502 **6.41.6 WTLS PRF (pseudorandom function)**

11503 PRF (pseudo random function) in WTLS, denoted **CKM\_WTLS\_PRF**, is a mechanism used to produce a  
11504 securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.  
11505 It has a parameter, a **CK\_WTLS\_PRF\_PARAMS** structure, which allows for passing the mechanism type  
11506 of the digest mechanism to be used, the passing of the input seed and its length, the passing of an  
11507 identifying label and its length and the passing of the length of the output to the token and for receiving  
11508 the output.  
11509 This mechanism produces securely generated pseudo-random output of the length specified in the  
11510 parameter.  
11511 This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template  
11512 sent along with this mechanism during a **C\_DeriveKey** function call, which means the template shall be a  
11513 NULL\_PTR. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result  
11514 of a successful completion. However, since the **CKM\_WTLS\_PRF** mechanism returns the requested  
11515 number of output bytes in the **CK\_WTLS\_PRF\_PARAMS** structure specified as the mechanism  
11516 parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.  
11517 If a call to **C\_DeriveKey** with this mechanism fails, then no output will be generated.

## 11518 **6.41.7 Server Key and MAC derivation**

11519 Server key, MAC and IV derivation in WTLS, denoted  
11520 **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate  
11521 cryptographic keying material used by a cipher suite from the master secret key and random data. This  
11522 mechanism returns the key handles for the keys generated in the process, as well as the IV created.  
11523 It has a parameter, a **CK\_WTLS\_KEY\_MAT\_PARAMS** structure, which allows for the passing of the  
11524 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic  
11525 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which  
11526 were generated.

11527 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested  
11528 by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.  
11529 The MACing key (server write MAC secret) is always given a type of **CKK\_GENERIC\_SECRET**. It is  
11530 flagged as valid for signing, verification and derivation operations.  
11531 The other key (server write key) is typed according to information found in the template sent along with  
11532 this mechanism during a **C\_DeriveKey** function call. By default, it is flagged as valid for encryption,  
11533 decryption, and derivation operations.  
11534 An IV (server write IV) will be generated and returned if the *ullVSizeInBits* field of the  
11535 **CK\_WTLS\_KEY\_MAT\_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree  
11536 with the value in the *ullVSizeInBits* field  
11537 Both keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**,  
11538 **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template  
11539 provided to **C\_DeriveKey** may not specify values for any of these attributes that differ from those held by  
11540 the base key.  
11541 Note that the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS**  
11542 structure's *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the two key  
11543 handle fields in the **CK\_WTLS\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-  
11544 created keys; in addition, the buffer pointed to by the **CK\_WTLS\_KEY\_MAT\_OUT** structure's *pIV* field will  
11545 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a  
11546 buffer with sufficient space to hold any IV that will be returned.  
11547 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.  
11548 For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a  
11549 successful completion. However, since the **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE**  
11550 mechanism returns all of its key handles in the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the  
11551 **CK\_WTLS\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*  
11552 passed to **C\_DeriveKey** is unnecessary, and should be a **NONE\_PTR**.  
11553 If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

#### 6.41.8 Client key and MAC derivation

11555 Client key, MAC and IV derivation in WTLS, denoted **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE**,  
11556 is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from  
11557 the master secret key and random data. This mechanism returns the key handles for the keys generated  
11558 in the process, as well as the IV created.  
11559 It has a parameter, a **CK\_WTLS\_KEY\_MAT\_PARAMS** structure, which allows for the passing of the  
11560 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic  
11561 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which  
11562 were generated.  
11563 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested  
11564 by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.  
11565 The MACing key (client write MAC secret) is always given a type of **CKK\_GENERIC\_SECRET**. It is  
11566 flagged as valid for signing, verification and derivation operations.  
11567 The other key (client write key) is typed according to information found in the template sent along with this  
11568 mechanism during a **C\_DeriveKey** function call. By default, it is flagged as valid for encryption,  
11569 decryption, and derivation operations.  
11570 An IV (client write IV) will be generated and returned if the *ullVSizeInBits* field of the  
11571 **CK\_WTLS\_KEY\_MAT\_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree  
11572 with the value in the *ullVSizeInBits* field  
11573 Both keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**,  
11574 **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template  
11575 provided to **C\_DeriveKey** may not specify values for any of these attributes that differ from those held by  
11576 the base key.

11577 Note that the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS**  
 11578 structure's *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the two key  
 11579 handle fields in the **CK\_WTLS\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-  
 11580 created keys; in addition, the buffer pointed to by the **CK\_WTLS\_KEY\_MAT\_OUT** structure's *pIV* field will  
 11581 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a  
 11582 buffer with sufficient space to hold any IV that will be returned.  
 11583 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.  
 11584 For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a  
 11585 successful completion. However, since the **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanism  
 11586 returns all of its key handles in the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the  
 11587 **CK\_WTLS\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*  
 11588 passed to **C\_DeriveKey** is unnecessary, and should be a **NONE\_PTR**.  
 11589 If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

## 11590 6.42 SP 800-108 Key Derivation

11591 NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a  
 11592 Feedback Mode KDF and a Double Pipeline Mode KDF.

11593 This section defines a unique mechanism for each type of KDF. These mechanisms can be used to  
 11594 derive one or more symmetric keys from a single base symmetric key.

11595 The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the  
 11596 PRFs accept two pieces of input; a base key and some input data. The base key is taken from the  
 11597 *hBaseKey* parameter to **C\_Derive**. The input data is constructed from an iteration variable (internally  
 11598 defined by the KDF/PRF) and the data provided in the **CK\_PRF\_DATA\_PARAM** array that is part of the  
 11599 mechanism parameter.

11600 *Table 191, SP800-108 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SP800_108_COUNTER_KDF							✓
CKM_SP800_108_FEEDBACK_KDF							✓
CKM_SP800_108_DOUBLE_PIPELINE_KDF							✓

11601  
 11602 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO**  
 11603 structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms  
 11604 support multiple PRF types and key types; as such the values reported by *ulMinKeySize* and  
 11605 *ulMaxKeySize* specify the minimum and maximum supported base key size when all PRF and keys types  
 11606 are considered. For example, a Cryptoki implementation may support CKK\_GENERIC\_SECRET keys  
 11607 that can be as small as 8-bits in length and therefore *ulMinKeySize* could report 8-bits. However, for an  
 11608 AES-CMAC PRF the base key must be of type CKK\_AES and must be either 16-bytes, 24-bytes or 32-  
 11609 bytes in lengths and therefore the value reported by *ulMinKeySize* could be misleading. Depending on  
 11610 the PRF type selected, additional key size restrictions may apply.

### 11611 6.42.1 Definitions

11612 Mechanisms:

11613     CKM\_SP800\_108\_COUNTER\_KDF  
 11614     CKM\_SP800\_108\_FEEDBACK\_KDF

```
11615      CKM_SP800_108_DOUBLE_PIPELINE_KDF
11616
11617 Data Field Types:
11618      CK_SP800_108_ITERATION_VARIABLE
11619      CK_SP800_108_COUNTER
11620      CK_SP800_108_DKM_LENGTH
11621      CK_SP800_108_BYTE_ARRAY
11622
11623 DKM Length Methods:
11624      CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS
11625      CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS
```

## 11626 6.42.2 Mechanism Parameters

### 11627 ♦ CK\_SP800\_108\_PRF\_TYPE

11628 The **CK\_SP800\_108\_PRF\_TYPE** field of the mechanism parameter is used to specify the type of PRF  
11629 that is to be used. It is defined as follows:

```
11630     typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;
```

11631 The **CK\_SP800\_108\_PRF\_TYPE** field reuses the existing mechanisms definitions. The following table  
11632 lists the supported PRF types:

11633 *Table 192, SP800-108 Pseudo Random Functions*

Pseudo Random Function Identifiers
CKM_SHA_1_HMAC
CKM_SHA224_HMAC
CKM_SHA256_HMAC
CKM_SHA384_HMAC
CKM_SHA512_HMAC
CKM_SHA3_224_HMAC
CKM_SHA3_256_HMAC
CKM_SHA3_384_HMAC
CKM_SHA3_512_HMAC
CKM_DES3_CMAC
CKM_AES_CMAC

11634

### 11635 ♦ CK\_PRF\_DATA\_TYPE

11636 Each mechanism parameter contains an array of **CK\_PRF\_DATA\_PARAM** structures. The  
11637 **CK\_PRF\_DATA\_PARAM** structure contains **CK\_PRF\_DATA\_TYPE** field. The **CK\_PRF\_DATA\_TYPE**  
11638 field is used to identify the type of data identified by each **CK\_PRF\_DATA\_PARAM** element in the array.  
11639 Depending on the type of KDF used, some data field types are mandatory, some data field types are  
11640 optional and some data field types are not allowed. These requirements are defined on a per-mechanism  
11641 basis in the sections below. The **CK\_PRF\_DATA\_TYPE** is defined as follows:

```
11642     typedef CK ULONG CK_PRF_DATA_TYPE;
```

11643 The following table lists all of the supported data field types:

11644 *Table 193, SP800-108 PRF Data Field Types*

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	Identifies the iteration variable defined internally by the KDF.
CK_SP800_108_COUNTER	Identifies an optional counter value represented as a binary string. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. The value of the counter is defined by the KDF's internal loop counter.
CK_SP800_108_DKM_LENGTH	Identifies the length in bits of the derived keying material (DKM) represented as a binary string. Exact formatting of the length value is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.
CK_SP800_108_BYTE_ARRAY	Identifies a generic byte array of data. This data type can be used to provide "context", "label", "separator bytes" as well as any other type of encoding information required by the higher level protocol.

11645

11646 ◆ **CK\_PRF\_DATA\_PARAM**

11647 **CK\_PRF\_DATA\_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter  
11648 supports an array of **CK\_PRF\_DATA\_PARAM** structures. The **CK\_PRF\_DATA\_PARAM** is defined as  
11649 follows:

```
11650     typedef struct CK_PRF_DATA_PARAM
11651     {
11652         CK_PRF_DATA_TYPE      type;
11653         CK_VOID_PTR           pValue;
11654         CK ULONG               ulValueLen;
11655     } CK_PRF_DATA_PARAM;
11656
11657     typedef CK_PRF_DATA_PARAM CK_PTR CK_PRF_DATA_PARAM_PTR
```

11659 The fields of the **CK\_PRF\_DATA\_PARAM** structure have the following meaning:

11660 type defines the type of data pointed to by pValue

11661 pValue pointer to the data defined by type

11662 ulValueLen size of the data pointed to by pValue

11663 If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to

11664 CK\_SP800\_108\_ITERATION\_VARIABLE, then *pValue* must be set the appropriate value for the KDF's  
11665 iteration variable type. For the Counter Mode KDF, *pValue* must be assigned a valid  
11666 CK\_SP800\_108\_COUNTER\_FORMAT\_PTR and *ulValueLen* must be set to  
11667 sizeof(CK\_SP800\_108\_COUNTER\_FORMAT). For all other KDF types, *pValue* must be set to  
11668 NULL\_PTR and *ulValueLen* must be set to 0.

11669

11670 If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_COUNTER, then  
11671 *pValue* must be assigned a valid CK\_SP800\_108\_COUNTER\_FORMAT\_PTR and *ulValueLen* must be  
11672 set to sizeof(CK\_SP800\_108\_COUNTER\_FORMAT).

11673

11674 If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_DKM\_LENGTH then  
 11675 *pValue* must be assigned a valid CK\_SP800\_108\_DKM\_LENGTH\_FORMAT\_PTR and *ulValueLen* must  
 11676 be set to sizeof(CK\_SP800\_108\_DKM\_LENGTH\_FORMAT).

11677

11678 If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_BYTE\_ARRAY, then  
 11679 *pValue* must be assigned a valid CK\_BYTE\_PTR value and *ulValueLen* must be set to a non-zero length.

11680 ◆ **CK\_SP800\_108\_COUNTER\_FORMAT**

11681 **CK\_SP800\_108\_COUNTER\_FORMAT** is used to define the encoding format for a counter value. The  
 11682 **CK\_SP800\_108\_COUNTER\_FORMAT** is defined as follows:

```
11683     typedef struct CK_SP800_108_COUNTER_FORMAT
11684     {
11685         CK_BBOOL      bLittleEndian;
11686         CK ULONG      ulWidthInBits;
11687     } CK_SP800_108_COUNTER_FORMAT;
```

11688

```
11689     typedef CK_SP800_108_COUNTER_FORMAT CK_PTR
11690     CK_SP800_108_COUNTER_FORMAT_PTR
```

11691

11692 The fields of the CK\_SP800\_108\_COUNTER\_FORMAT structure have the following meaning:  
 11693       **bLittleEndian**     defines if the counter should be represented in Big Endian or Little  
 11694                          Endian format  
 11695        **ulWidthInBits**    defines the number of bits used to represent the counter value

11696 ◆ **CK\_SP800\_108\_DKM\_LENGTH\_METHOD**

11697 **CK\_SP800\_108\_DKM\_LENGTH\_METHOD** is used to define how the DKM length value is calculated.  
 11698 The **CK\_SP800\_108\_DKM\_LENGTH\_METHOD** type is defined as follows:

```
11699     typedef CK ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

11700 The following table lists all of the supported DKM Length Methods:

11701 *Table 194, SP800-108 DKM Length Methods*

<b>DKM Length Method Identifier</b>	<b>Description</b>
CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS	Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF.
CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS	Specifies that the DKM length should be set to the sum of the length of all segments of output produced by the PRF by this invocation of the KDF.

11702

11703 ◆ **CK\_SP800\_108\_DKM\_LENGTH\_FORMAT**

11704 **CK\_SP800\_108\_DKM\_LENGTH\_FORMAT** is used to define the encoding format for the DKM length  
 11705 value. The **CK\_SP800\_108\_DKM\_LENGTH\_FORMAT** is defined as follows:

```
11706     typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```

```

11707     {
11708         CK_SP800_108_DKM_LENGTH_METHOD    dkmLengthMethod;
11709         CK_BBOOL                         bLittleEndian;
11710         CK ULONG                          ulWidthInBits;
11711     } CK_SP800_108_DKM_LENGTH_FORMAT;
11712
11713     typedef CK_SP800_108_DKM_LENGTH_FORMAT CK_PTR
11714     CK_SP800_108_DKM_LENGTH_FORMAT_PTR
11715
11716 The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:
11717     dkmLengthMethod      defines the method used to calculate the DKM length value
11718     bLittleEndian        defines if the DKM length value should be represented in Big
11719                     Endian or Little Endian format
11720     ulWidthInBits       defines the number of bits used to represent the DKM length value

```

#### ◆ CK\_DERIVED\_KEY

11722 **CK\_DERIVED\_KEY** is used to define an additional key to be derived as well as provide a  
11723 **CK\_OBJECT\_HANDLE\_PTR** to receive the handle for the derived keys. The **CK\_DERIVED\_KEY** is  
11724 defined as follows:

```

11725     typedef struct CK_DERIVED_KEY
11726     {
11727         CK_ATTRIBUTE_PTR      pTemplate;
11728         CK ULONG             ulAttributeCount;
11729         CK_OBJECT_HANDLE_PTR phKey;
11730     } CK_DERIVED_KEY;
11731
11732     typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR
11733

```

11734 The fields of the **CK\_DERIVED\_KEY** structure have the following meaning:  
11735 pTemplate pointer to a template that defines a key to derive  
11736 ulAttributeCount number of attributes in the template pointed to by pTemplate  
11737 phKey pointer to receive the handle for a derived key

#### ◆ CK\_SP800\_108\_KDF\_PARAMS, CK\_SP800\_108\_KDF\_PARAMS\_PTR

11739 **CK\_SP800\_108\_KDF\_PARAMS** is a structure that provides the parameters for the  
11740 **CKM\_SP800\_108\_COUNTER\_KDF** and **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** mechanisms.

```

11741
11742     typedef struct CK_SP800_108_KDF_PARAMS
11743     {
11744         CK_SP800_108_PRF_TYPE    prfType;
11745         CK ULONG                 ulNumberOfDataParams;
11746         CK_PRF_DATA_PARAM_PTR   pDataParams;
11747         CK ULONG                 ulAdditionalDerivedKeys;
11748         CK_DERIVED_KEY_PTR      pAdditionalDerivedKeys;
11749     } CK_SP800_108_KDF_PARAMS;

```

```

11750
11751     typedef CK_SP800_108_KDF_PARAMS CK_PTR
11752     CK_SP800_108_KDF_PARAMS_PTR;
11753
11754 The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:
11755     prfType      type of PRF
11756     ulNumberOfDataParams   number of elements in the array pointed to by pDataParams
11757             pDataParams    an array of CK_PRF_DATA_PARAM structures. The array defines
11758             input parameters that are used to construct the "data" input to the
11759             PRF.
11760     ulAdditionalDerivedKeys   number of additional keys that will be derived and the number of
11761             elements in the array pointed to by pAdditionalDerivedKeys. If
11762             pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
11763             be set to 0.
11764     pAdditionalDerivedKeys    an array of CK_DERIVED_KEY structures. If
11765             ulAdditionalDerivedKeys is set to 0, this parameter must be set to
11766             NULL_PTR

11767 ◆ CK_SP800_108_FEEDBACK_KDF_PARAMS,
11768 CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR

11769 The CK_SP800_108_FEEDBACK_KDF_PARAMS structure provides the parameters for the
11770 CKM_SP800_108_FEEDBACK_KDF mechanism. It is defined as follows:
11771     typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
11772     {
11773         CK_SP800_108_PRF_TYPE      prfType;
11774         CK ULONG                  ulNumberOfDataParams;
11775         CK_PRF_DATA_PARAM_PTR    pDataParams;
11776         CK ULONG                  ulIVLen;
11777         CK_BYTE_PTR              pIV;
11778         CK ULONG                  ulAdditionalDerivedKeys;
11779         CK_DERIVED_KEY_PTR       pAdditionalDerivedKeys;
11780     } CK_SP800_108_FEEDBACK_KDF_PARAMS;
11781
11782     typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
11783     CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;
11784
11785 The fields of the CK_SP800_108_FEEDBACK_KDF_PARAMS structure have the following meaning:
11786     prfType      type of PRF
11787     ulNumberOfDataParams   number of elements in the array pointed to by pDataParams
11788             pDataParams    an array of CK_PRF_DATA_PARAM structures. The array defines
11789             input parameters that are used to construct the "data" input to the
11790             PRF.
11791             ullIVLen    the length in bytes of the IV. If pIV is set to NULL_PTR, this
11792             parameter must be set to 0.
11793             pIV        an array of bytes to be used as the IV for the feedback mode KDF.
11794             This parameter is optional and can be set to NULL_PTR. If ullIVLen
11795             is set to 0, this parameter must be set to NULL_PTR.

```

11796            number of additional keys that will be derived and the number of  
 11797           elements in the array pointed to by pAdditionalDerivedKeys. If  
 11798           pAdditionalDerivedKeys is set to NULL\_PTR, this parameter must  
 11799           be set to 0.  
 11800            an array of CK\_DERIVED\_KEY structures. If  
 11801            is set to 0, this parameter must be set to  
 11802           NULL\_PTR.

### 6.42.3 Counter Mode KDF

11804         The SP800-108 Counter Mode KDF mechanism, denoted **CKM\_SP800\_108\_COUNTER\_KDF**,  
 11805         represents the KDF defined SP800-108 section 5.1. **CKM\_SP800\_108\_COUNTER\_KDF** is a  
 11806         mechanism for deriving one or more symmetric keys from a symmetric base key.

11807         It has a parameter, a **CK\_SP800\_108\_KDF\_PARAMS** structure.

11808         The following table lists the data field types that are supported for this KDF type and their meaning:

11809         Table 195, Counter Mode data field requirements

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	<p>This data field type is mandatory.</p> <p>This data field type identifies the location of the iteration variable in the constructed PRF input data.</p> <p>The iteration variable for this KDF type is a counter.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p>
CK_SP800_108_COUNTER	This data field type is invalid for this KDF type.
CK_SP800_108_DKM_LENGTH	<p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_BYTE_ARRAY	<p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p>

11810  
 11811         SP800-108 limits the amount of derived keying material that can be produced by a Counter Mode KDF by  
 11812         limiting the internal loop counter to  $(2^r - 1)$ , where "r" is the number of bits used to represent the counter.  
 11813         Therefore the maximum number of bits that can be produced is  $(2^r - 1)h$ , where "h" is the length in bits of  
 11814         the output of the selected PRF.

### 6.42.4 Feedback Mode KDF

11816         The SP800-108 Feedback Mode KDF mechanism, denoted **CKM\_SP800\_108\_FEEDBACK\_KDF**,  
 11817         represents the KDF defined SP800-108 section 5.2. **CKM\_SP800\_108\_FEEDBACK\_KDF** is a  
 11818         mechanism for deriving one or more symmetric keys from a symmetric base key.

11819         It has a parameter, a **CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS** structure.

11820         The following table lists the data field types that are supported for this KDF type and their meaning:

11821         Table 196, Feedback Mode data field requirements

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	<p>This data field type is mandatory.</p> <p>This data field type identifies the location of the iteration variable in the constructed PRF input data.</p> <p>The iteration variable is defined as K(i-1) in section 5.2 of SP800-108.</p> <p>The size, format and value of this data input is defined by the internal KDF structure and PRF output.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p>
CK_SP800_108_COUNTER	<p>This data field type is optional.</p> <p>This data field type identifies the location of the counter in the constructed PRF input data.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_DKM_LENGTH	<p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_BYTE_ARRAY	<p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p>

11822

11823 SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF  
 11824 by limiting the internal loop counter to  $(2^{32}-1)$ . Therefore the maximum number of bits that can be  
 11825 produced is  $(2^{32}-1)h$ , where "h" is the length in bits of the output of the selected PRF.

#### 11826 6.42.5 Double Pipeline Mode KDF

11827 The SP800-108 Double Pipeline Mode KDF mechanism, denoted  
 11828 **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF**, represents the KDF defined SP800-108 section 5.3.  
 11829 **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** is a mechanism for deriving one or more symmetric keys  
 11830 from a symmetric base key.

11831 It has a parameter, a CK\_SP800\_108\_KDF\_PARAMS structure.

11832 The following table lists the data field types that are supported for this KDF type and their meaning:

11833 *Table 197, Double Pipeline Mode data field requirements*

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	<p>This data field type is mandatory.</p> <p>This data field type identifies the location of the iteration variable in the constructed PRF input data.</p> <p>The iteration variable is defined as A(i) in section 5.3 of SP800-108.</p>

	<p>The size, format and value of this data input is defined by the internal KDF structure and PRF output.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p>
CK_SP800_108_COUNTER	<p>This data field type is optional.</p> <p>This data field type identifies the location of the counter in the constructed PRF input data.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_DKM_LENGTH	<p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_BYTE_ARRAY	<p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p>

11834

11835 SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode  
 11836 KDF by limiting the internal loop counter to  $(2^{32}-1)$ . Therefore the maximum number of bits that can be  
 11837 produced is  $(2^{32}-1)h$ , where "h" is the length in bits of the output of the selected PRF.

11838 The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method  
 11839 used to construct the PRF input data; the data/values identified by the array of CK\_PRF\_DATA\_PARAM  
 11840 structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3,  
 11841 the CK\_SP800\_108\_ITERATION\_VARIABLE and CK\_SP800\_108\_COUNTER data field types are not  
 11842 included in IV construction process. All other data field types are included in the construction process.

## 11843 6.42.6 Deriving Additional Keys

11844 The KDFs defined in this section can be used to derive more than one symmetric key from the base key.  
 11845 The **C\_Derive** function accepts one CK\_ATTRIBUTE\_PTR to define a single derived key and one  
 11846 CK\_OBJECT\_HANDLE\_PTR to receive the handle for the derived key.

11847 To derive additional keys, the mechanism parameter structure can be filled in with one or more  
 11848 CK\_DERIVED\_KEY structures. Each structure contains a CK\_ATTRIBUTE\_PTR to define a derived key  
 11849 and a CK\_OBJECT\_HANDLE\_PTR to receive the handle for the additional derived keys. The key  
 11850 defined by the **C\_Derive** function parameters is always derived before the keys defined by the  
 11851 CK\_DERIVED\_KEY array that is part of the mechanism parameter. The additional keys that are defined  
 11852 by the CK\_DERIVED\_KEY array are derived in the order they are defined in the array. That is to say that  
 11853 the derived keying material produced by the KDF is processed from left to right, and bytes are assigned  
 11854 first to the key defined by the **C\_Derive** function parameters, and then bytes are assigned to the keys that  
 11855 are defined by the CK\_DERIVED\_KEY array in the order they are defined in the array.

11856 Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration  
 11857 will produce enough keying material for the key being derived. Other times, additional internal iterations  
 11858 are performed to produce multiple segments which are concatenated together to produce enough keying  
 11859 material for the derived key(s).

11860 When deriving multiple keys, no key can be created using part of a segment that was used for another  
 11861 key. All keys must be created from disjoint segments. For example, if the parameters are defined such

11862 that a 48-byte key (defined by the **C\_Derive** function parameters) and a 16-byte key (defined by the  
11863 content of CK\_DERIVED\_KEY) are to be derived using **CKM\_SHA256\_HMAC** as a PRF, three internal  
11864 iterations of the KDF will be performed and three segments of PRF output will be produced. The first  
11865 segment and half of the second segment will be used to create the 48-byte key and the third segment will  
11866 be used to create the 16-byte key.

3 KDF Segments of Output:	32-byte segment	32-byte segment	32-byte segment
2 Derived Keys:	48-byte key	unused	16-byte key

11867  
11868 In the above example, if the CK\_SP800\_108\_DKM\_LENGTH data field type is specified with method  
11869 CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS, then the DKM length value will be 512 bits. If the  
11870 CK\_SP800\_108\_DKM\_LENGTH data field type is specified with method  
11871 CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS, then the DKM length value will be 768 bits.

11872 When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall  
11873 be derived. If the failure was caused by the content of a specific key's template (ie the template defined  
11874 by the content of *pTemplate*), the corresponding *phKey* value will be set to CK\_INVALID\_HANDLE to  
11875 identify the offending template.

## 11876 6.42.7 Key Derivation Attribute Rules

11877 The **CKM\_SP800\_108\_COUNTER\_KDF**, **CKM\_SP800\_108\_FEEDBACK\_KDF** and  
11878 **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** mechanisms have the following rules about key sensitivity  
11879 and extractability:

- 11880 • The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key(s) can  
11881 both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on  
11882 some default value.
- 11883 • If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key  
11884 will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the  
11885 derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its  
11886 **CKA\_SENSITIVE** attribute.
- 11887 • Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the  
11888 derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to  
11889 CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite*  
11890 value from its **CKA\_EXTRACTABLE** attribute.

## 11891 6.42.8 Constructing PRF Input Data

11892 SP800-108 defines the PRF input data for each KDF at a high level using terms like "label", "context",  
11893 "separator", "counter"...etc. The value, formatting and order of the input data is not strictly defined by  
11894 SP800-108, instead it is described as being defined by the "encoding scheme".

11895 To support any encoding scheme, these mechanisms construct the PRF input data from the array of  
11896 CK\_PRF\_DATA\_PARAM structures in the mechanism parameter. All of the values defined by the  
11897 CK\_PRF\_DATA\_PARAM array are concatenated in the order they are defined and passed in to the PRF  
11898 as the data parameter.

### 11899 6.42.8.1 Sample Counter Mode KDF

11900 SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

11901  $\text{PRF}(K, [i]_2 \parallel \text{Label} \parallel 0x00 \parallel \text{Context} \parallel [L]_2)$

11902 Section 5.1 does not define the number of bits used to represent the counter (the "r" value) or the DKM  
11903 length (the "L" value), so 16-bits is assumed for both cases. The following sample code shows how to  
11904 define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

```

11905     #define DIM(a) (sizeof((a))/sizeof((a)[0]))
11906
11907     CK_OBJECT_HANDLE hBaseKey;
11908     CK_OBJECT_HANDLE hDerivedKey;
11909     CK_ATTRIBUTE derivedKeyTemplate = { ... };
11910
11911     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
11912     CK_ULONG ulLabelLen = sizeof(baLabel);
11913     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
11914     CK_ULONG ulContextLen = sizeof(baContext);
11915
11916     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
11917     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
11918         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
11919
11920     CK_PRF_DATA_PARAM dataParams[] =
11921     {
11922         { CK_SP800_108_ITERATION_VARIABLE,
11923             &counterFormat, sizeof(counterFormat) },
11924         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
11925         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
11926         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
11927         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
11928     };
11929
11930     CK_SP800_108_KDF_PARAMS kdfParams =
11931     {
11932         CKM_AES_CMAC,
11933         DIM(dataParams),
11934         &dataParams,
11935         0, /* no addition derived keys */
11936         NULL /* no addition derived keys */
11937     };
11938
11939     CK_MECHANISM = mechanism
11940     {
11941         CKM_SP800_108_COUNTER_KDF,
11942         &kdfParams,
11943         sizeof(kdfParams)
11944     };
11945
11946     hBaseKey = GetBaseKeyHandle(.....);
11947
11948     rv = C_DeriveKey(
11949         hSession,
11950         &mechanism,
11951         hBaseKey,
11952         &derivedKeyTemplate,
11953         DIM(derivedKeyTemplate),
11954         &hDerivedKey);

```

#### 6.42.8.2 Sample SCP03 Counter Mode KDF

11955 The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:  
11956     PRF ( $K_1, Label \parallel 0x00 \parallel [L]_2 \parallel [i]_2 \parallel Context$ )

```

11958 SCP03 defines the number of bits used to represent the counter (the "r" value) and number of bits used to
11959 represent the DKM length (the "L" value) as 16-bits. The following sample code shows how to define this
11960 PRF input data using an array of CK_PRF_DATA_PARAM structures.
11961 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
11962
11963     CK_OBJECT_HANDLE hBaseKey;
11964     CK_OBJECT_HANDLE hDerivedKey;
11965     CK_ATTRIBUTE derivedKeyTemplate = { ... };
11966
11967     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
11968     CK ULONG ulLabelLen = sizeof(baLabel);
11969     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
11970     CK ULONG ulContextLen = sizeof(baContext);
11971
11972     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
11973     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
11974         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
11975
11976     CK_PRF_DATA_PARAM dataParams[] =
11977     {
11978         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
11979         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
11980         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
11981         { CK_SP800_108_ITERATION_VARIABLE,
11982             &counterFormat, sizeof(counterFormat) },
11983         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
11984     };
11985
11986     CK_SP800_108_KDF_PARAMS kdfParams =
11987     {
11988         CKM_AES_CMAC,
11989         DIM(dataParams),
11990         &dataParams,
11991         0, /* no addition derived keys */
11992         NULL /* no addition derived keys */
11993     };
11994
11995     CK_MECHANISM = mechanism
11996     {
11997         CKM_SP800_108_COUNTER_KDF,
11998         &kdfParams,
11999         sizeof(kdfParams)
12000     };
12001
12002     hBaseKey = GetBaseKeyHandle(.....);
12003
12004     rv = C_DeriveKey(
12005         hSession,
12006         &mechanism,
12007         hBaseKey,
12008         &derivedKeyTemplate,
12009         DIM(derivedKeyTemplate),
12010         &hDerivedKey);

```

### 6.42.8.3 Sample Feedback Mode KDF

SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

12013                     $\text{PRF}(K_I, K(i-1) \{\mid [i]_2\} \mid Label \mid 0x00 \mid Context \mid [L]_2)$

12014     Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and is included in this example. The following sample code shows how to define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

```

12018     #define DIM(a) (sizeof((a))/sizeof((a)[0]))
12019
12020     CK_OBJECT_HANDLE hBaseKey;
12021     CK_OBJECT_HANDLE hDerivedKey;
12022     CK_ATTRIBUTE derivedKeyTemplate = { ... };
12023
12024     CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
12025     CK ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
12026     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
12027     CK ULONG ulLabelLen = sizeof(baLabel);
12028     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
12029     CK ULONG ulContextLen = sizeof(baContext);
12030
12031     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
12032     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
12033         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
12034
12035     CK_PRF_DATA_PARAM dataParams[] =
12036     {
12037         { CK_SP800_108_ITERATION_VARIABLE,
12038             &counterFormat, sizeof(counterFormat) },
12039         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
12040         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
12041         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
12042         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
12043     };
12044
12045     CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =
12046     {
12047         CKM_AES_CMAC,
12048         DIM(dataParams),
12049         &dataParams,
12050         ulFeedbackIVLen,
12051         baFeedbackIV,
12052         0, /* no addition derived keys */
12053         NULL /* no addition derived keys */
12054     };
12055
12056     CK_MECHANISM = mechanism
12057     {
12058         CKM_SP800_108_FEEDBACK_KDF,
12059         &kdfParams,
12060         sizeof(kdfParams)
12061     };
12062
12063     hBaseKey = GetBaseKeyHandle(.....);
12064
12065     rv = C_DeriveKey(
12066         hSession,
12067         &mechanism,
12068         hBaseKey,
```

```
12069     &derivedKeyTemplate,  
12070     DIM(derivedKeyTemplate),  
12071     &hDerivedKey);
```

#### 6.42.8.4 Sample Double-Pipeline Mode KDF

12073 SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following  
12074 PRF inputs:

```
12075     PRF (KI, A(i-1))  
12076     PRF (KI, K(i-1) {|| [i]2 }|| Label || 0x00 || Context || [L]2)
```

12077 Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM  
12078 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it  
12079 is left out in this example. The following sample code shows how to define this PRF input data using an  
12080 array of CK\_PRF\_DATA\_PARAM structures.

```
12081 #define DIM(a) (sizeof((a))/sizeof((a)[0]))  
12082  
12083     CK_OBJECT_HANDLE hBaseKey;  
12084     CK_OBJECT_HANDLE hDerivedKey;  
12085     CK_ATTRIBUTE derivedKeyTemplate = { ... };  
12086  
12087     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};  
12088     CK_ULONG ulLabelLen = sizeof(baLabel);  
12089     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};  
12090     CK_ULONG ulContextLen = sizeof(baContext);  
12091  
12092     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat  
12093         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};  
12094  
12095     CK_PRF_DATA_PARAM dataParams[] =  
12096     {  
12097         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },  
12098         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },  
12099         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },  
12100         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }  
12101     };  
12102  
12103     CK_SP800_108_KDF_PARAMS kdfParams =  
12104     {  
12105         CKM_AES_CMAC,  
12106         DIM(dataParams),  
12107         &dataParams,  
12108         0, /* no addition derived keys */  
12109         NULL /* no addition derived keys */  
12110     };  
12111  
12112     CK_MECHANISM = mechanism  
12113     {  
12114         CKM_SP800_108_DOUBLE_PIPELINE_KDF,  
12115         &kdfParams,  
12116         sizeof(kdfParams)  
12117     };  
12118  
12119     hBaseKey = GetBaseKeyHandle(. . . . .);  
12120  
12121     rv = C_DeriveKey(  
12122         hSession,
```

```

12123     &mechanism,
12124     hBaseKey,
12125     &derivedKeyTemplate,
12126     DIM(derivedKeyTemplate),
12127     &hDerivedKey);

```

## 6.43 Miscellaneous simple key derivation mechanisms

Table 198, *Miscellaneous simple key derivation Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CONCATENATE_BASE_AND_KEY							✓
CKM_CONCATENATE_BASE_AND_DATA							✓
CKM_CONCATENATE_DATA_AND_BASE							✓
CKM_XOR_BASE_AND_DATA							✓
CKM_EXTRACT_KEY_FROM_KEY							✓

### 6.43.1 Definitions

Mechanisms:

```

CKM_CONCATENATE_BASE_AND_DATA
CKM_CONCATENATE_DATA_AND_BASE
CKM_XOR_BASE_AND_DATA
CKM_EXTRACT_KEY_FROM_KEY
CKM_CONCATENATE_BASE_AND_KEY

```

### 6.43.2 Parameters for miscellaneous simple key derivation mechanisms

- ◆ **CK\_KEY\_DERIVATION\_STRING\_DATA;**
- CK\_KEY\_DERIVATION\_STRING\_DATA\_PTR**

CK\_KEY\_DERIVATION\_STRING\_DATA provides the parameters for the CKM\_CONCATENATE\_BASE\_AND\_DATA, CKM\_CONCATENATE\_DATA\_AND\_BASE, and CKM\_XOR\_BASE\_AND\_DATA mechanisms. It is defined as follows:

```

typedef struct CK_KEY_DERIVATION_STRING_DATA {
    CK_BYTE_PTR pData;
    CK ULONG ulLen;
} CK_KEY_DERIVATION_STRING_DATA;

```

The fields of the structure have the following meanings:

pData	pointer to the byte string
ulLen	length of the byte string

**CK\_KEY\_DERIVATION\_STRING\_DATA\_PTR** is a pointer to a **CK\_KEY\_DERIVATION\_STRING\_DATA**.

12153 ◆ **CK\_EXTRACT\_PARAMS; CK\_EXTRACT\_PARAMS\_PTR**

12154 **CK\_EXTRACT\_PARAMS** provides the parameter to the **CKM\_EXTRACT\_KEY\_FROM\_KEY**  
12155 mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is  
12156 defined as follows:

12157     **typedef CK ULONG CK\_EXTRACT\_PARAMS;**

12158  
12159 **CK\_EXTRACT\_PARAMS\_PTR** is a pointer to a **CK\_EXTRACT\_PARAMS**.

12160 **6.43.3 Concatenation of a base key and another key**

12161 This mechanism, denoted **CKM\_CONCATENATE\_BASE\_AND\_KEY**, derives a secret key from the  
12162 concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys  
12163 specified are concatenated together in a buffer.

12164 This mechanism takes a parameter, a **CK\_OBJECT\_HANDLE**. This handle produces the key value  
12165 information which is appended to the end of the base key's value information (the base key is the key  
12166 whose handle is supplied as an argument to **C\_DeriveKey**).

12167 For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF,  
12168 then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 12169 • If no length or key type is provided in the template, then the key produced by this mechanism will be a  
12170 generic secret key. Its length will be equal to the sum of the lengths of the values of the two original  
12171 keys.
- 12172 • If no key type is provided in the template, but a length is, then the key produced by this mechanism  
12173 will be a generic secret key of the specified length.
- 12174 • If no length is provided in the template, but a key type is, then that key type must have a well-defined  
12175 length. If it does, then the key produced by this mechanism will be of the type specified in the  
12176 template. If it doesn't, an error will be returned.
- 12177 • If both a key type and a length are provided in the template, the length must be compatible with that  
12178 key type. The key produced by this mechanism will be of the specified type and length.

12179 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set  
12180 properly.

12181 If the requested type of key requires more bytes than are available by concatenating the two original keys'  
12182 values, an error is generated.

12183 This mechanism has the following rules about key sensitivity and extractability:

- 12184 • If either of the two original keys has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the  
12185 derived key. If not, then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied  
12186 template or from a default value.
- 12187 • Similarly, if either of the two original keys has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE,  
12188 so does the derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either  
12189 from the supplied template or from a default value.
- 12190 • The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if both of the  
12191 original keys have their **CKA\_ALWAYS\_SENSITIVE** attributes set to CK\_TRUE.
- 12192 • Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if  
12193 both of the original keys have their **CKA\_NEVER\_EXTRACTABLE** attributes set to CK\_TRUE.

12194 **6.43.4 Concatenation of a base key and data**

12195 This mechanism, denoted **CKM\_CONCATENATE\_BASE\_AND\_DATA**, derives a secret key by  
12196 concatenating data onto the end of a specified secret key.

12197 This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which  
12198 specifies the length and value of the data which will be appended to the base key to derive another key.  
12199 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then  
12200 the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.  
12201 • If no length or key type is provided in the template, then the key produced by this mechanism will be a  
12202 generic secret key. Its length will be equal to the sum of the lengths of the value of the original key  
12203 and the data.  
12204 • If no key type is provided in the template, but a length is, then the key produced by this mechanism  
12205 will be a generic secret key of the specified length.  
12206 • If no length is provided in the template, but a key type is, then that key type must have a well-defined  
12207 length. If it does, then the key produced by this mechanism will be of the type specified in the  
12208 template. If it doesn't, an error will be returned.  
12209 • If both a key type and a length are provided in the template, the length must be compatible with that  
12210 key type. The key produced by this mechanism will be of the specified type and length.  
12211 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set  
12212 properly.  
12213 If the requested type of key requires more bytes than are available by concatenating the original key's  
12214 value and the data, an error is generated.  
12215 This mechanism has the following rules about key sensitivity and extractability:  
12216 • If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not,  
12217 then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from a  
12218 default value.  
12219 • Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the  
12220 derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the  
12221 supplied template or from a default value.  
12222 • The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base  
12223 key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.  
12224 • Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if  
12225 the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

#### 6.43.5 Concatenation of data and a base key

12226 This mechanism, denoted **CKM\_CONCATENATE\_DATA\_AND\_BASE**, derives a secret key by  
12227 prepending data to the start of a specified secret key.  
12228 This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which  
12229 specifies the length and value of the data which will be prepended to the base key to derive another key.  
12230 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then  
12231 the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.  
12232 • If no length or key type is provided in the template, then the key produced by this mechanism will be a  
12233 generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the  
12234 original key.  
12235 • If no key type is provided in the template, but a length is, then the key produced by this mechanism  
12236 will be a generic secret key of the specified length.  
12237 • If no length is provided in the template, but a key type is, then that key type must have a well-defined  
12238 length. If it does, then the key produced by this mechanism will be of the type specified in the  
12239 template. If it doesn't, an error will be returned.  
12240 • If both a key type and a length are provided in the template, the length must be compatible with that  
12241 key type. The key produced by this mechanism will be of the specified type and length.  
12242 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set  
12243 properly.

- 12245 If the requested type of key requires more bytes than are available by concatenating the data and the  
12246 original key's value, an error is generated.
- 12247 This mechanism has the following rules about key sensitivity and extractability:
- If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
  - Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
  - The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
  - Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## 12258 **6.43.6 XORing of a key and data**

12259 XORing key derivation, denoted **CKM\_XOR\_BASE\_AND\_DATA**, is a mechanism which provides the  
12260 capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle  
12261 and some data.

12262 This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which  
12263 specifies the data with which to XOR the original key's value.

12264 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then  
12265 the value of the derived key will be taken from a buffer containing the string 0x88888888.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

12266 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set  
12267 properly.

12268 If the requested type of key requires more bytes than are available by taking the shorter of the data and  
12269 the original key's value, an error is generated.

12270 This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
- Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## 12291 6.43.7 Extraction of one key from another key

12292 Extraction of one key from another key, denoted **CKM\_EXTRACT\_KEY\_FROM\_KEY**, is a mechanism  
12293 which provides the capability of creating one secret key from the bits of another secret key.

12294 This mechanism has a parameter, a CK\_EXTRACT\_PARAMS, which specifies which bit of the original  
12295 key should be used as the first bit of the newly-derived key.

12296 We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte  
12297 value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the  
12298 value of the parameter to the CKM\_EXTRACT\_KEY\_FROM\_KEY mechanism is 21).

- 12299 1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this  
12300 binary string as holding the 32 bits of the key, labeled as b0, b1, ..., b31.
- 12301 2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We  
12302 obtain the binary string 1001 0101 0010 0110.
- 12303 3. The value of the new key is thus 0x9526.

12304 Note that when constructing the value of the derived key, it is permissible to wrap around the end of the  
12305 binary string representing the original key's value.

12306 If the original key used in this process is sensitive, then the derived key must also be sensitive for the  
12307 derivation to succeed.

- 12308 • If no length or key type is provided in the template, then an error will be returned.
- 12309 • If no key type is provided in the template, but a length is, then the key produced by this mechanism  
12310 will be a generic secret key of the specified length.
- 12311 • If no length is provided in the template, but a key type is, then that key type must have a well-defined  
12312 length. If it does, then the key produced by this mechanism will be of the type specified in the  
12313 template. If it doesn't, an error will be returned.
- 12314 • If both a key type and a length are provided in the template, the length must be compatible with that  
12315 key type. The key produced by this mechanism will be of the specified type and length.

12316 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set  
12317 properly.

12318 If the requested type of key requires more bytes than the original key has, an error is generated.

12319 This mechanism has the following rules about key sensitivity and extractability:

- 12320 • If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not,  
12321 then the derived key's **CKA\_SENSITIVE** attribute is set either from the supplied template or from a  
12322 default value.
- 12323 • Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the  
12324 derived key. If not, then the derived key's **CKA\_EXTRACTABLE** attribute is set either from the  
12325 supplied template or from a default value.
- 12326 • The derived key's **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base  
12327 key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
- 12328 • Similarly, the derived key's **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if  
12329 the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## 12330 6.44 CMS

12331 *Table 199, CMS Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CMS_SIG		✓	✓				

12332 **6.44.1 Definitions**

12333 Mechanisms:

12334 CKM\_CMS\_SIG

12335 **6.44.2 CMS Signature Mechanism Objects**

12336 These objects provide information relating to the CKM\_CMS\_SIG mechanism. CKM\_CMS\_SIG  
 12337 mechanism object attributes represent information about supported CMS signature attributes in the token.  
 12338 They are only present on tokens supporting the **CKM\_CMS\_SIG** mechanism, but must be present on  
 12339 those tokens.

12340 *Table 200, CMS Signature Mechanism Object Attributes*

Attribute	Data type	Meaning
CKA_REQUIRED_CMS_ATTRIBUTES	Byte array	Attributes the token always will include in the set of CMS signed attributes
CKA_DEFAULT_CMS_ATTRIBUTES	Byte array	Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application
CKA_SUPPORTED_CMS_ATTRIBUTES	Byte array	Attributes the token may include in the set of CMS signed attributes upon request by the application

12341 The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying  
 12342 values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-  
 12343 encoded. The list of attributes is defined in ASN.1 as:

```

12344     Attributes ::= SET SIZE (1..MAX) OF Attribute
12345     Attribute ::= SEQUENCE {
12346         attrType   OBJECT IDENTIFIER,
12347         attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
12348             OPTIONAL
12349     }
  
```

12350 The client may not set any of the attributes.

12351 **6.44.3 CMS mechanism parameters**

- **CK\_CMS\_SIG\_PARAMS, CK\_CMS\_SIG\_PARAMS\_PTR**

12352 **CK\_CMS\_SIG\_PARAMS** is a structure that provides the parameters to the **CKM\_CMS\_SIG** mechanism.  
 12353 It is defined as follows:

```

12355     typedef struct CK_CMS_SIG_PARAMS {
12356         CK_OBJECT_HANDLE      certificateHandle;
12357         CK_MECHANISM_PTR      pSigningMechanism;
12358         CK_MECHANISM_PTR      pDigestMechanism;
  
```

```

12359     CK_UTF8CHAR_PTR           pContentType;
12360     CK_BYTE_PTR                pRequestedAttributes;
12361     CK ULONG                   ulRequestedAttributesLen;
12362     CK_BYTE_PTR                pRequiredAttributes;
12363     CK ULONG                   ulRequiredAttributesLen;
12364 } CK_CMS_SIG_PARAMS;
12365
12366 The fields of the structure have the following meanings:
12367     certificateHandle          Object handle for a certificate associated with the signing key. The token may use information from this certificate to identify the signer in the SignerInfo result value. CertificateHandle may be NULL_PTR if the certificate is not available as a PKCS #11 object or if the calling application leaves the choice of certificate completely to the token.
12368
12369     pSigningMechanism          Mechanism to use when signing a constructed CMS SignedAttributes value. E.g. CKM_SHA1_RSA_PKCS.
12370
12371     pDigestMechanism          Mechanism to use when digesting the data. Value shall be NULL_PTR when the digest mechanism to use follows from the pSigningMechanism parameter.
12372
12373     pContentType               NULL-terminated string indicating complete MIME Content-type of message to be signed; or the value NULL_PTR if the message is a MIME object (which the token can parse to determine its MIME Content-type if required). Use the value "application/octet-stream" if the MIME type for the message is unknown or undefined. Note that the pContentType string shall conform to the syntax specified in RFC 2045, i.e. any parameters needed for correct presentation of the content by the token (such as, for example, a non-default "charset") must be present. The token must follow rules and procedures defined in RFC 2045 when presenting the content.
12374
12375     pRequestedAttributes       Pointer to DER-encoded list of CMS Attributes the caller requests to be included in the signed attributes. Token may freely ignore this list or modify any supplied values.
12376
12377     ulRequestedAttributesLen   Length in bytes of the value pointed to by pRequestedAttributes
12378
12379     pRequiredAttributes        Pointer to DER-encoded list of CMS Attributes (with accompanying values) required to be included in the resulting signed attributes. Token must not modify any supplied values. If the token does not support one or more of the attributes, or does not accept provided values, the signature operation will fail. The token will use its own default attributes when signing if both the pRequestedAttributes and pRequiredAttributes field are set to NULL_PTR.
12380
12381     ulRequiredAttributesLen   Length in bytes, of the value pointed to by pRequiredAttributes.
12382
12383
12384
12385
12386
12387
12388
12389
12390
12391
12392
12393
12394
12395
12396
12397
12398
12399
12400

```

#### 6.44.4 CMS signatures

The CMS mechanism, denoted **CKM\_CMS\_SIG**, is a multi-purpose mechanism based on the structures defined in PKCS #7 and RFC 2630. It supports single- or multiple-part signatures with and without message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value. The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be

12407 present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the  
12408 parameter *pSigningMechanism*.

12409 When this mechanism is used in successful calls to **C\_Sign** or **C\_SignFinal**, the *pSignature* return value  
12410 will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a  
12411 complete definition of all fields and types, see RFC 2630):

```
12412     SignerInfo ::= SEQUENCE {
12413         version CMSVersion,
12414         sid SignerIdentifier,
12415         digestAlgorithm DigestAlgorithmIdentifier,
12416         signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
12417         signatureAlgorithm SignatureAlgorithmIdentifier,
12418         signature SignatureValue,
12419         unsignedAttrs [1] IMPLICIT UnsignedAttributes
12420         OPTIONAL }
```

12421 The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value.  
12422 If *certificateHandle* is NULL\_PTR the choice of a suitable certificate reference in the **SignerInfo** result  
12423 value is left to the token (the token could, e.g., interact with the user).

12424 This mechanism shall not be used in calls to **C\_Verify** or **C\_VerifyFinal** (use the *pSigningMechanism*  
12425 mechanism instead).

12426 For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to  
12427 accept a proposed value or not. The token should never accept any proposed attribute values without  
12428 some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings  
12429 and not direct interaction). If a user rejects proposed values, or the signature request as such, the value  
12430 CKR\_FUNCTION\_REJECTED shall be returned.

12431 When possible, applications should use the **CKM\_CMS\_SIG** mechanism when generating CMS-  
12432 compatible signatures rather than lower-level mechanisms such as **CKM\_SHA1\_RSA\_PKCS**. This is  
12433 especially true when the signatures are to be made on content that the token is able to present to a user.  
12434 Exceptions may include those cases where the token does not support a particular signing attribute. Note  
12435 however that the token may refuse usage of a particular signature key unless the content to be signed is  
12436 known (i.e. the **CKM\_CMS\_SIG** mechanism is used).

12437 When a token does not have presentation capabilities, the PKCS #11-aware application may avoid  
12438 sending the whole message to the token by electing to use a suitable signature mechanism (e.g.  
12439 **CKM\_RSA\_PKCS**) as the *pSigningMechanism* value in the **CK\_CMS\_SIG\_PARAMS** structure, and  
12440 digesting the message itself before passing it to the token.

12441 PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to  
12442 provide messages to be signed by the token in a format possible for the token to present to the user.  
12443 Tokens that receive multipart MIME-messages for which only certain parts are possible to present may  
12444 fail the signature operation with a return value of **CKR\_DATA\_INVALID**, but may also choose to add a  
12445 signing attribute indicating which parts of the message were possible to present.

## 12446 6.45 Blowfish

12447 Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times.  
12448 The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex  
12449 initialization phase required before any encryption can take place, the actual encryption of data is very  
12450 efficient on large microprocessors.

12451

12452 *Table 201, Blowfish Mechanisms vs. Functions*

<b>Mechanism</b>	<b>Functions</b>						
	<b>Encrypt &amp; Decrypt</b>	<b>Sign &amp; Verify</b>	<b>SR &amp; VR<sup>1</sup></b>	<b>Digest</b>	<b>Gen. Key/ Key Pair</b>	<b>Wrap &amp; Unwrap</b>	<b>Derive</b>
CKM_BLOWFISH_CBC	✓					✓	
CKM_BLOWFISH_CBC_PAD	✓					✓	

## 12453 6.45.1 Definitions

12454 This section defines the key type "CKK\_BLOWFISH" for type CK\_KEY\_TYPE as used in the  
 12455 CKA\_KEY\_TYPE attribute of key objects.

12456 Mechanisms:

12457 CKM\_BLOWFISH\_KEY\_GEN  
 12458 CKM\_BLOWFISH\_CBC  
 12459 CKM\_BLOWFISH\_CBC\_PAD

## 12460 6.45.2 BLOWFISH secret key objects

12461 Blowfish secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_BLOWFISH) hold Blowfish  
 12462 keys. The following table defines the Blowfish secret key object attributes, in addition to the common  
 12463 attributes defined for this object class:

12464 *Table 202, BLOWFISH Secret Key Object*

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value the key can be any length up to 448 bits. Bit length restricted to a byte array.
CKA_VALUE_LEN <sup>2,3</sup>	CK ULONG	Length in bytes of key value

12465 <sup>1</sup>Refer to Table 11 for footnotes

12466 The following is a sample template for creating an Blowfish secret key object:

```
12467     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12468     CK_KEY_TYPE keyType = CKK_BLOWFISH;
12469     CK_UTF8CHAR label[] = "A blowfish secret key object";
12470     CK_BYTE value[16] = {...};
12471     CK_BBOOL true = CK_TRUE;
12472     CK_ATTRIBUTE template[] = {
12473         {CKA_CLASS, &class, sizeof(class)},
12474         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12475         {CKA_TOKEN, &true, sizeof(true)},
12476         {CKA_LABEL, label, sizeof(label)-1},
12477         {CKA_ENCRYPT, &true, sizeof(true)},
12478         {CKA_VALUE, value, sizeof(value)}}
12479     };
```

### 6.45.3 Blowfish key generation

The Blowfish key generation mechanism, denoted **CKM\_BLOWFISH\_KEY\_GEN**, is a key generation mechanism Blowfish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes in bytes.

### 6.45.4 Blowfish-CBC

Blowfish-CBC, denoted **CKM\_BLOWFISH\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 8-byte initialization vector.

This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 203, BLOWFISH-CBC: Key and Data Length

Function	Key type	Input Length	Output Length
C_Encrypt	BLOWFISH	Multiple of block size	Same as input length
C_Decrypt	BLOWFISH	Multiple of block size	Same as input length
C_WrapKey	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_UnwrapKey	BLOWFISH	Multiple of block size	Determined by type of key being unwrapped or <b>CKA_VALUE_LEN</b>

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of BLOWFISH key sizes, in bytes.

### 6.45.5 Blowfish-CBC with PKCS padding

Blowfish-CBC-PAD, denoted **CKM\_BLOWFISH\_CBC\_PAD**, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 8-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

12518 Constraints on key types and the length of data are summarized in the following table:

12519

12520 *Table 204, BLOWFISH-CBC with PKCS Padding: Key and Data Length*

Function	Key type	Input Length	Output Length
C_Encrypt	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_Decrypt	BLOWFISH	Multiple of block size	Between 1 and block length block size bytes shorter than input length
C_WrapKey	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_UnwrapKey	BLOWFISH	Multiple of block size	Between 1 and block length block size bytes shorter than input length

## 12521 **6.46 Twofish**

12522 Ref. <https://www.schneier.com/twofish.html>

### 12523 **6.46.1 Definitions**

12524 This section defines the key type “CKK\_TWOFISH” for type CK\_KEY\_TYPE as used in the  
12525 CKA\_KEY\_TYPE attribute of key objects.

12526 Mechanisms:

12527 CKM\_TWOFISH\_KEY\_GEN

12528 CKM\_TWOFISH\_CBC

12529 CKM\_TWOFISH\_CBC\_PAD

12530

### 12531 **6.46.2 Twofish secret key objects**

12532 Twofish secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_TWOFISH) hold Twofish  
12533 keys. The following table defines the Twofish secret key object attributes, in addition to the common  
12534 attributes defined for this object class:

12535 *Table 205, Twofish Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value 128-, 192-, or 256-bit key
CKA_VALUE_LEN <sup>2,3</sup>	CK ULONG	Length in bytes of key value

12536 Refer to Table 11 for footnotes

12537 The following is a sample template for creating an TWOFISH secret key object:

```
12538 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12539 CK_KEY_TYPE keyType = CKK_TWOFISH;
12540 CK_UTF8CHAR label[] = "A twofish secret key object";
12541 CK_BYTE value[16] = {...};
12542 CK_BBOOL true = CK_TRUE;
12543 CK_ATTRIBUTE template[] = {
12544     {CKA_CLASS, &class, sizeof(class)},
```

```

12545     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
12546     { CKA_TOKEN, &true, sizeof(true) },
12547     { CKA_LABEL, label, sizeof(label)-1 },
12548     { CKA_ENCRYPT, &true, sizeof(true) },
12549     { CKA_VALUE, value, sizeof(value) }
12550 } ;

```

### 6.46.3 Twofish key generation

12551 The Twofish key generation mechanism, denoted **CKM\_TWOFISH\_KEY\_GEN**, is a key generation  
 12552 mechanism Twofish.  
 12553 It does not have a parameter.  
 12554 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA\_VALUE\_LEN**  
 12555 attribute of the template for the key.  
 12556 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
 12557 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key  
 12558 supports) may be specified in the template for the key, or else are assigned default initial values.  
 12559 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12560 specify the supported range of key sizes, in bytes.

### 6.46.4 Twofish -CBC

12561 Twofish-CBC, denoted **CKM\_TWOFISH\_CBC**, is a mechanism for single- and multiple-part encryption  
 12562 and decryption; key wrapping; and key unwrapping.  
 12563 It has a parameter, a 16-byte initialization vector.

### 6.46.5 Twofish-CBC with PKCS padding

12564 Twofish-CBC-PAD, denoted **CKM\_TWOFISH\_CBC\_PAD**, is a mechanism for single- and multiple-part  
 12565 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block  
 12566 cipher padding method detailed in PKCS #7.  
 12567 It has a parameter, a 16-byte initialization vector.  
 12568 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the  
 12569 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for  
 12570 the **CKA\_VALUE\_LEN** attribute.

## 6.47 CAMELLIA

12571 Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.  
 12572 Camellia is described e.g. in IETF RFC 3713.  
 12573 *Table 206, Camellia Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen . Key / Key Pair	Wrap & Unwrap	Derive
CKM_CAMELLIA_KEY_GEN					✓		
CKM_CAMELLIA_ECB	✓					✓	

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen . Key / Key Pair	Wrap & Unwrap	Derive
CKM_CAMELLIA_CBC	✓					✓	
CKM_CAMELLIA_CBC_PAD	✓					✓	
CKM_CAMELLIA_MAC_GENERAL		✓					
CKM_CAMELLIA_MAC		✓					
CKM_CAMELLIA_ECB_ENCRYPT_DATA							✓
CKM_CAMELLIA_CBC_ENCRYPT_DATA							✓

## 6.47.1 Definitions

This section defines the key type “CKK\_CAMELLIA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

- CKM\_CAMELLIA\_KEY\_GEN
- CKM\_CAMELLIA\_ECB
- CKM\_CAMELLIA\_CBC
- CKM\_CAMELLIA\_MAC
- CKM\_CAMELLIA\_MAC\_GENERAL
- CKM\_CAMELLIA\_CBC\_PAD

## 6.47.2 Camellia secret key objects

Camellia secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_CAMELLIA) hold Camellia keys. The following table defines the Camellia secret key object attributes, in addition to the common attributes defined for this object class:

Table 207, Camellia Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN <sup>2,3,6</sup>	CK ULONG	Length in bytes of key value

Refer to Table 11 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAMELLIA;
CK_UTF8CHAR label[] = "A Camellia secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```

```
12601     { CKA_CLASS, &class, sizeof(class) },
12602     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
12603     { CKA_TOKEN, &true, sizeof(true) },
12604     { CKA_LABEL, label, sizeof(label)-1 },
12605     { CKA_ENCRYPT, &true, sizeof(true) },
12606     { CKA_VALUE, value, sizeof(value) }
12607 };
```

### 6.47.3 Camellia key generation

The Camellia key generation mechanism, denoted **CKM\_CAMELLIA\_KEY\_GEN**, is a key generation mechanism for Camellia.

It does not have a parameter.

The mechanism generates Camellia keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### 6.47.4 Camellia-ECB

Camellia-ECB, denoted **CKM\_CAMELLIA\_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 208, *Camellia-ECB: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of block size	
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

12635 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12636 specify the supported range of Camellia key sizes, in bytes.

## 6.47.5 Camellia-CBC

12638 Camellia-CBC, denoted **CKM\_CAMELLIA\_CBC**, is a mechanism for single- and multiple-part encryption  
 12639 and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

12640 It has a parameter, a 16-byte initialization vector.

12641 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
 12642 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the  
 12643 **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus  
 12644 one null bytes so that the resulting length is a multiple of the block size. The output data is the same  
 12645 length as the padded input data. It does not wrap the key type, key length, or any other information about  
 12646 the key; the application must convey these separately.

12647 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the  
 12648 **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the  
 12649 **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE**  
 12650 attribute of the new key; other attributes required by the key type must be specified in the template.

12651 Constraints on key types and the length of data are summarized in the following table:

12652 *Table 209, Camellia-CBC: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of the block size	
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

12653 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12654 specify the supported range of Camellia key sizes, in bytes.

## 6.47.6 Camellia-CBC with PKCS padding

12655 Camellia-CBC with PKCS padding, denoted **CKM\_CAMELLIA\_CBC\_PAD**, is a mechanism for single-  
 12656 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia;  
 12657 cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

- 12659 It has a parameter, a 16-byte initialization vector.
- 12660 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.
- 12663 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.
- 12667 Constraints on key types and the length of data are summarized in the following table:

12668 *Table 210, Camellia-CBC with PKCS Padding: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_CAMELLIA	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	between 1 and block length bytes shorter than input length

12669 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
12670 specify the supported range of Camellia key sizes, in bytes.

12671

## 12672 **6.47.7 CAMELLIA with Counter mechanism parameters**

### 12673 ♦ **CK\_CAMELLIA\_CTR\_PARAMS; CK\_CAMELLIA\_CTR\_PARAMS\_PTR**

12674 **CK\_CAMELLIA\_CTR\_PARAMS** is a structure that provides the parameters to the  
12675 **CKM\_CAMELLIA\_CTR** mechanism. It is defined as follows:

```
12676     typedef struct CK_CAMELLIA_CTR_PARAMS {
12677         CK ULONG ulCounterBits;
12678         CK_BYTE cb[16];
12679     } CK_CAMELLIA_CTR_PARAMS;
```

12681 *ulCounterBits* specifies the number of bits in the counter block (*cb*) that shall be incremented. This  
12682 number shall be such that  $0 < \text{ulCounterBits} \leq 128$ . For any values outside this range the mechanism  
12683 shall return **CKR\_MECHANISM\_PARAM\_INVALID**.

12684 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter  
12685 bits are the least significant bits of the counter block (*cb*). They are a big-endian value usually starting  
12686 with 1. The rest of '*cb*' is for the nonce, and maybe an optional IV.

12687 E.g. as defined in [RFC 3686]:

```

12688      0           1           2           3
12689      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
12690      +-----+-----+-----+-----+-----+-----+
12691      |                   Nonce                   |
12692      +-----+-----+-----+-----+-----+-----+
12693      |                   Initialization Vector (IV)   |
12694      |
12695      +-----+-----+-----+-----+-----+-----+-----+
12696      |                   Block Counter            |
12697      +-----+-----+-----+-----+-----+-----+-----+

```

12698

12699 This construction permits each packet to consist of up to  $2^{32}-1$  blocks = 4,294,967,295 blocks =
12700 68,719,476,720 octets.

12701 **CK\_CAMELLIA\_CTR\_PARAMS\_PTR** is a pointer to a **CK\_CAMELLIA\_CTR\_PARAMS**.

12702

## 12703 6.47.8 General-length Camellia-MAC

12704 General-length Camellia -MAC, denoted CKM\_CAMELLIA\_MAC\_GENERAL, is a mechanism for single-
12705 and multiple-part signatures and verification, based on Camellia and data authentication as defined
12706 in.[CAMELLIA]

12707 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length
12708 desired from the mechanism.

12709 The output bytes from this mechanism are taken from the start of the final Camellia cipher block produced
12710 in the MACing process.

12711 Constraints on key types and the length of data are summarized in the following table:

12712 *Table 211, General-length Camellia-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	1-block size, as specified in parameters
C_Verify	CKK_CAMELLIA	any	1-block size, as specified in parameters

12713 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure
12714 specify the supported range of Camellia key sizes, in bytes.

## 12715 6.47.9 Camellia-MAC

12716 Camellia-MAC, denoted by **CKM\_CAMELLIA\_MAC**, is a special case of the general-length Camellia-
12717 MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
12718 length.

12719 It does not have a parameter.

12720 Constraints on key types and the length of data are summarized in the following table:

12721 *Table 212, Camellia-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	$\frac{1}{2}$ block size (8 bytes)
C_Verify	CKK_CAMELLIA	any	$\frac{1}{2}$ block size (8 bytes)

12722 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure
12723 specify the supported range of Camellia key sizes, in bytes.

## 6.48 Key derivation by data encryption - Camellia

These mechanisms allow derivation of keys using the result of an encryption operation as the key value.  
They are for use with the C\_DeriveKey function.

### 6.48.1 Definitions

Mechanisms:

```
CKM_CAMELLIA_ECB_ENCRYPT_DATA  
CKM_CAMELLIA_CBC_ENCRYPT_DATA
```

```
typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {  
    CK_BYTE        iv[16];  
    CK_BYTE_PTR   pData;  
    CK ULONG      length;  
} CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;  
  
typedef CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

### 6.48.2 Mechanism Parameters

Uses CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS, and CK\_KEY\_DERIVATION\_STRING\_DATA.

Table 213, Mechanism Parameters for Camellia-based key derivation

CKM_CAMELLIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_CAMELLIA_CBC_ENCRYPT_DATA	Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

12743

## 6.49 ARIA

ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is described in NSRI "Specification of ARIA".

Table 214, ARIA Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_KEY_GEN					✓		
CKM_ARIA_ECB	✓					✓	
CKM_ARIA_CBC	✓					✓	
CKM_ARIA_CBC_PAD	✓					✓	

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_MAC_GENERAL		✓					
CKM_ARIA_MAC		✓					
CKM_ARIA_ECB_ENCRYPT_DATA							✓
CKM_ARIA_CBC_ENCRYPT_DATA							✓

## 12748 6.49.1 Definitions

12749 This section defines the key type "CKK\_ARIA" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE  
 12750 attribute of key objects.

12751 Mechanisms:

12752 CKM\_ARIA\_KEY\_GEN  
 12753 CKM\_ARIA\_ECB  
 12754 CKM\_ARIA\_CBC  
 12755 CKM\_ARIA\_MAC  
 12756 CKM\_ARIA\_MAC\_GENERAL  
 12757 CKM\_ARIA\_CBC\_PAD

## 12758 6.49.2 Aria secret key objects

12759 ARIA secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_ARIA) hold ARIA keys. The  
 12760 following table defines the ARIA secret key object attributes, in addition to the common attributes defined  
 12761 for this object class:

12762 *Table 215, ARIA Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN <sup>2,3,6</sup>	CK ULONG	Length in bytes of key value

12763 - Refer to Table 11 for footnotes

12764 The following is a sample template for creating an ARIA secret key object:

```
12765 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12766 CK_KEY_TYPE keyType = CKK_ARIA;
12767 CK_UTF8CHAR label[] = "An ARIA secret key object";
12768 CK_BYTE value[] = { ... };
12769 CK_BBOOL true = CK_TRUE;
12770 CK_ATTRIBUTE template[] = {
12771     {CKA_CLASS, &class, sizeof(class)},
12772     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12773     {CKA_TOKEN, &true, sizeof(true)},
12774     {CKA_LABEL, label, sizeof(label)-1},
12775     {CKA_ENCRYPT, &true, sizeof(true)},
```

```
12776     { CKA_VALUE, value, sizeof(value) }  
12777 } ;
```

### 6.49.3 ARIA key generation

12779 The ARIA key generation mechanism, denoted CKM\_ARIA\_KEY\_GEN, is a key generation mechanism  
12780 for Aria.

12781 It does not have a parameter.

12782 The mechanism generates ARIA keys with a particular length in bytes, as specified in the  
12783 **CKA\_VALUE\_LEN** attribute of the template for the key.

12784 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
12785 key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the  
12786 key supports) may be specified in the template for the key, or else are assigned default initial values.

12787 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
12788 specify the supported range of ARIA key sizes, in bytes.

### 6.49.4 ARIA-ECB

12790 ARIA-ECB, denoted **CKM\_ARIA\_ECB**, is a mechanism for single- and multiple-part encryption and  
12791 decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

12792 It does not have a parameter.

12793 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
12794 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the  
12795 **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus  
12796 one null bytes so that the resulting length is a multiple of the block size. The output data is the same  
12797 length as the padded input data. It does not wrap the key type, key length, or any other information about  
12798 the key; the application must convey these separately.

12799 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the  
12800 **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the  
12801 **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE**  
12802 attribute of the new key; other attributes required by the key type must be specified in the template.

12803 Constraints on key types and the length of data are summarized in the following table:

12804 *Table 216, ARIA-ECB: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of block size	
C_UnwrapKey	CKK_ARIA	multiple of block size	determined by type of key being unwrapped or <b>CKA_VALUE_LEN</b>	

12805 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
12806 specify the supported range of ARIA key sizes, in bytes.

### 6.49.5 ARIA-CBC

12808 ARIA-CBC, denoted **CKM\_ARIA\_CBC**, is a mechanism for single- and multiple-part encryption and  
12809 decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

12810 It has a parameter, a 16-byte initialization vector.

12811 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
 12812 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the  
 12813 **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus  
 12814 one null bytes so that the resulting length is a multiple of the block size. The output data is the same  
 12815 length as the padded input data. It does not wrap the key type, key length, or any other information about  
 12816 the key; the application must convey these separately.

12817 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the  
 12818 **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the  
 12819 **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE**  
 12820 attribute of the new key; other attributes required by the key type must be specified in the template.

12821 Constraints on key types and the length of data are summarized in the following table:

12822 *Table 217, ARIA-CBC: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of the block size	
C_UnwrapKey	CKK_ARIA	multiple of block size	determined by type of key being unwrapped or <b>CKA_VALUE_LEN</b>	

12823 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK\_MECHANISM\_INFO structure  
 12824 specify the supported range of Aria key sizes, in bytes.

#### 12825 **6.49.6 ARIA-CBC with PKCS padding**

12826 ARIA-CBC with PKCS padding, denoted **CKM\_ARIA\_CBC\_PAD**, is a mechanism for single- and  
 12827 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block  
 12828 chaining mode; and the block cipher padding method detailed in PKCS #7.

12829 It has a parameter, a 16-byte initialization vector.

12830 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the  
 12831 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified  
 12832 for the **CKA\_VALUE\_LEN** attribute.

12833 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,  
 12834 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for  
 12835 details). The entries in the table below for data length constraints when wrapping and unwrapping keys  
 12836 do not apply to wrapping and unwrapping private keys.

12837 Constraints on key types and the length of data are summarized in the following table:

12838 *Table 218, ARIA-CBC with PKCS Padding: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_ARIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_ARIA	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_ARIA	multiple of block size	between 1 and block length bytes shorter than input length

12839 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12840 specify the supported range of ARIA key sizes, in bytes.

## 6.49.7 General-length ARIA-MAC

12842 General-length ARIA -MAC, denoted **CKM\_ARIA\_MAC\_GENERAL**, is a mechanism for single- and  
 12843 multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

12844 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length  
 12845 desired from the mechanism.

12846 The output bytes from this mechanism are taken from the start of the final ARIA cipher block produced in  
 12847 the MACing process.

12848 Constraints on key types and the length of data are summarized in the following table:

12849 *Table 219, General-length ARIA-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	1-block size, as specified in parameters
C_Verify	CKK_ARIA	any	1-block size, as specified in parameters

12850 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12851 specify the supported range of ARIA key sizes, in bytes.

## 6.49.8 ARIA-MAC

12853 ARIA-MAC, denoted by **CKM\_ARIA\_MAC**, is a special case of the general-length ARIA-MAC  
 12854 mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

12855 It does not have a parameter.

12856 Constraints on key types and the length of data are summarized in the following table:

12857 *Table 220, ARIA-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	$\frac{1}{2}$ block size (8 bytes)
C_Verify	CKK_ARIA	any	$\frac{1}{2}$ block size (8 bytes)

12858 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 12859 specify the supported range of ARIA key sizes, in bytes.

## 6.50 Key derivation by data encryption - ARIA

12861 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.  
 12862 They are for use with the C\_DeriveKey function.

12863 **6.50.1 Definitions**

12864 Mechanisms:

```
12865     CKM_ARIA_ECB_ENCRYPT_DATA  
12866     CKM_ARIA_CBC_ENCRYPT_DATA  
12867  
12868     typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {  
12869         CK_BYTE           iv[16];  
12870         CK_BYTE_PTR      pData;  
12871         CK_ULONG          length;  
12872     } CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;  
12873  
12874     typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
12875             CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

12876 **6.50.2 Mechanism Parameters**

12877 Uses CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS, and CK\_KEY\_DERIVATION\_STRING\_DATA.

12878 *Table 221, Mechanism Parameters for Aria-based key derivation*

CKM_ARIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_ARIA_CBC_ENCRYPT_DATA	Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

12879

12880 **6.51 SEED**

12881 SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It  
12882 has a 128-bit key size and a 128-bit block size.

12883 Its specification has been published as Internet [RFC 4269].

12884 RFCs have been published defining the use of SEED in

12885 TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

12886 IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

12887 CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

12888

12889 TLS cipher suites that use SEED include:

```
12890     CipherSuite TLS_RSA_WITH_SEED_CBC_SHA      = { 0x00,  
12891         0x96};  
12892     CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA   = { 0x00,  
12893         0x97};  
12894     CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA   = { 0x00,  
12895         0x98};  
12896     CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA = { 0x00,  
12897         0x99};
```

```

12898     CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00,
12899         0x9A};
12900     CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00,
12901         0x9B};

12902
12903 As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as
12904 in a MAC algorithm such as HMAC.
12905 OIDs have been published for all these uses. A list may be seen at
12906 http://www.alvestrand.no/objectid/1.2.410.200004.1.html
12907
12908 Table 222, SEED Mechanisms vs. Functions

```

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SEED_KEY_GEN					✓		
CKM_SEED_ECB			✓				
CKM_SEED_CBC			✓				
CKM_SEED_CBC_PAD	✓					✓	
CKM_SEED_MAC_GENERAL			✓				
CKM_SEED_MAC				✓			
CKM_SEED_ECB_ENCRYPT_DATA							✓
CKM_SEED_CBC_ENCRYPT_DATA							✓

## 6.51.1 Definitions

This section defines the key type "CKK\_SEED" for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

```

12913     CKM_SEED_KEY_GEN
12914     CKM_SEED_ECB
12915     CKM_SEED_CBC
12916     CKM_SEED_MAC
12917     CKM_SEED_MAC_GENERAL
12918     CKM_SEED_CBC_PAD
12919

```

For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** are always 16.

## 6.51.2 SEED secret key objects

SEED secret key objects (object class **CKO\_SECRET\_KEY**, key type **CKK\_SEED**) hold SEED keys. The following table defines the secret key object attributes, in addition to the common attributes defined for this object class:

12926 Table 223, SEED Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	Key value (always 16 bytes long)

12927 ^ Refer to Table 11 for footnotes

12928 The following is a sample template for creating a SEED secret key object:

```
12929 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12930 CK_KEY_TYPE keyType = CKK_SEED;
12931 CK_UTF8CHAR label[] = "A SEED secret key object";
12932 CK_BYTE value[] = {...};
12933 CK_BBOOL true = CK_TRUE;
12934 CK_ATTRIBUTE template[] = {
12935     {CKA_CLASS, &class, sizeof(class)},
12936     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12937     {CKA_TOKEN, &true, sizeof(true)},
12938     {CKA_LABEL, label, sizeof(label)-1},
12939     {CKA_ENCRYPT, &true, sizeof(true)},
12940     {CKA_VALUE, value, sizeof(value)}}
12941 };
```

### 12942 6.51.3 SEED key generation

12943 The SEED key generation mechanism, denoted CKM\_SEED\_KEY\_GEN, is a key generation mechanism  
12944 for SEED.

12945 It does not have a parameter.

12946 The mechanism generates SEED keys.

12947 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
12948 key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions  
12949 the key supports) may be specified in the template for the key, or else are assigned default initial values.

### 12950 6.51.4 SEED-ECB

12951 SEED-ECB, denoted **CKM\_SEED\_ECB**, is a mechanism for single- and multiple-part encryption and  
12952 decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

12953 It does not have a parameter.

### 12954 6.51.5 SEED-CBC

12955 SEED-CBC, denoted **CKM\_SEED\_CBC**, is a mechanism for single- and multiple-part encryption and  
12956 decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

12957 It has a parameter, a 16-byte initialization vector.

### 12958 6.51.6 SEED-CBC with PKCS padding

12959 SEED-CBC with PKCS padding, denoted **CKM\_SEED\_CBC\_PAD**, is a mechanism for single- and  
12960 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-  
12961 block chaining mode; and the block cipher padding method detailed in PKCS #7.

12962 It has a parameter, a 16-byte initialization vector.

## 12963 6.51.7 General-length SEED-MAC

12964 General-length SEED-MAC, denoted **CKM\_SEED\_MAC\_GENERAL**, is a mechanism for single- and  
12965 multiple-part signatures and verification, based on SEED and data authentication.

12966 It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length  
12967 desired from the mechanism.

12968 The output bytes from this mechanism are taken from the start of the final cipher block produced in the  
12969 MACing process.

## 12970 6.51.8 SEED-MAC

12971 SEED-MAC, denoted by **CKM\_SEED\_MAC**, is a special case of the general-length SEED-MAC  
12972 mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

12973 It does not have a parameter.

## 12974 6.52 Key derivation by data encryption - SEED

12975 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.  
12976 They are for use with the C\_DeriveKey function.

### 12977 6.52.1 Definitions

12978 Mechanisms:

```
12979     CKM_SEED_ECB_ENCRYPT_DATA  
12980     CKM_SEED_CBC_ENCRYPT_DATA
```

```
12981  
12982     typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS {  
12983         CK_BYTE           iv[16];  
12984         CK_BYTE_PTR      pData;  
12985         CK_ULONG          length;  
12986     } CK_SEED_CBC_ENCRYPT_DATA_PARAMS;  
12987  
12988     typedef CK_SEED_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
12989             CK_SEED_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

### 12990 6.52.2 Mechanism Parameters

12991 *Table 224, Mechanism Parameters for SEED-based key derivation*

CKM_SEED_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_SEED_CBC_ENCRYPT_DATA	Uses CK_SEED_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

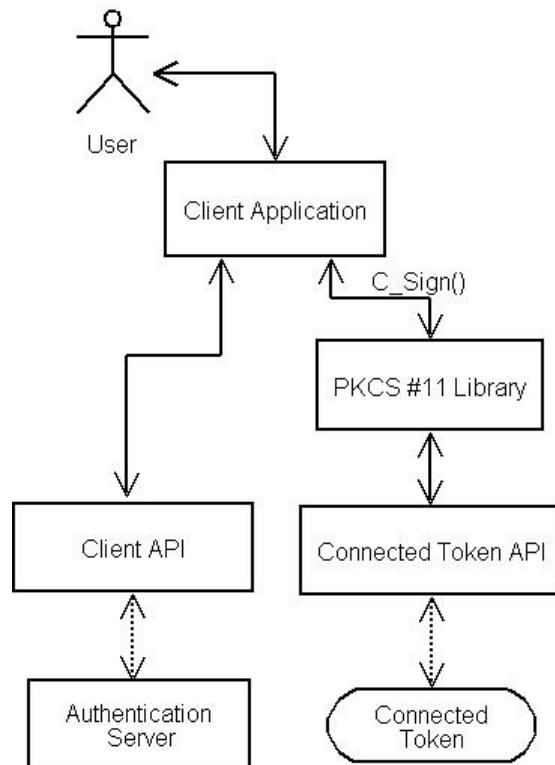
12992

12993 **6.53 OTP**

12994 **6.53.1 Usage overview**

12995 OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases  
12996 can be categorized according to the type of sought functionality.

12997 **6.53.2 Case 1: Generation of OTP values**

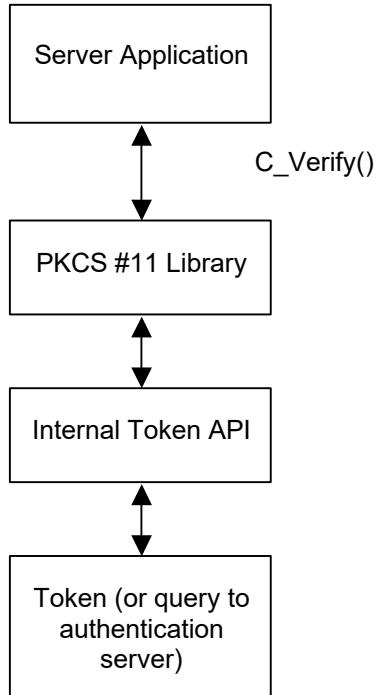


12998

12999 *Figure 2: Retrieving OTP values through C\_Sign*

13000 Figure 2 shows an integration of PKCS #11 into an application that needs to authenticate users holding  
13001 OTP tokens. In this particular example, a connected hardware token is used, but a software token is  
13002 equally possible. The application invokes **C\_Sign** to retrieve the OTP value from the token. In the  
13003 example, the application then passes the retrieved OTP value to a client API that sends it via the network  
13004 to an authentication server. The client API may implement a standard authentication protocol such as  
13005 RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's  
13006 ACE/Agent® software.

13007 **6.53.3 Case 2: Verification of provided OTP values**

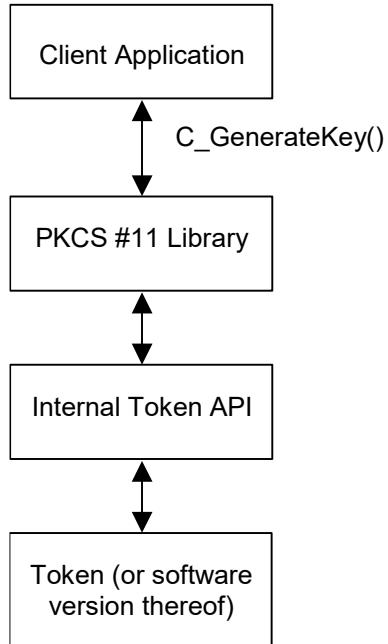


13008

13009 *Figure 3: Server-side verification of OTP values*

13010 Figure 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. In this case, a server  
13011 application invokes **C\_Verify** with the received OTP value as the signature value to be verified.

13012 **6.53.4 Case 3: Generation of OTP keys**



13013

13014 *Figure 4: Generation of an OTP key*

13015 Figure 4 shows an integration of PKCS #11 into an application that generates OTP keys. The application  
13016 invokes **C\_GenerateKey** to generate an OTP key of a particular type on the token. The key may  
13017 subsequently be used as a basis to generate OTP values.

13018 **6.53.5 OTP objects**

13019 **6.53.5.1 Key objects**

13020 OTP key objects (object class **CKO OTP KEY**) hold secret keys used by OTP tokens. The following  
13021 table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys,  
13022 all of which are inherited by this class:

13023 *Table 225: Common OTP key attributes*

<b>Attribute</b>	<b>Data type</b>	<b>Meaning</b>
CKA OTP FORMAT	CK ULONG	Format of OTP values produced with this key: CK OTP FORMAT DECIMAL = Decimal (default) (UTF8-encoded) CK OTP FORMAT HEXADECIMAL = Hexadecimal (UTF8-encoded) CK OTP FORMAT ALPHANUMERIC = Alphanumeric (UTF8-encoded) CK OTP FORMAT BINARY = Only binary values.
CKA OTP LENGTH <sup>9</sup>	CK ULONG	Default length of OTP values (in the CKA OTP FORMAT) produced with this key.
CKA OTP USER FRIENDLY MODE <sup>9</sup>	CK BBOOL	Set to CK TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF USER FRIENDLY OTP below.
CKA OTP CHALLENGE REQUIREMENT <sup>9</sup>	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A challenge must be supplied. CK OTP PARAM OPTIONAL = A challenge may be supplied but need not be. CK OTP PARAM IGNORED = A challenge, if supplied, will be ignored.
CKA OTP TIME REQUIREMENT <sup>9</sup>	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A time value must be supplied. CK OTP PARAM OPTIONAL = A time value may be supplied but need not be. CK OTP PARAM IGNORED = A time value, if supplied, will be ignored.

CKA OTP COUNTER REQUIREMENT <sup>9</sup>	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A counter value must be supplied. CK OTP PARAM OPTIONAL = A counter value may be supplied but need not be. CK OTP PARAM IGNORED = A counter value, if supplied, will be ignored.
CKA OTP PIN REQUIREMENT <sup>9</sup>	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A PIN value must be supplied. CK OTP PARAM OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it) CK OTP PARAM IGNORED = A PIN value, if supplied, will be ignored.
CKA OTP COUNTER	Byte array	Value of the associated internal counter. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP TIME	RFC 2279 string	Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP USER IDENTIFIER	RFC 2279 string	Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE IDENTIFIER	RFC 2279 string	Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE LOGO	Byte array	Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE LOGO TYPE	RFC 2279 string	MIME type of the CKA OTP SERVICE LOGO attribute value. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA VALUE <sup>1, 4, 6, 7</sup>	Byte array	Value of the key.
CKA VALUE LEN <sup>2, 3</sup>	CK ULONG	Length in bytes of key value.

13025 Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-  
 13026 PKCS #11 application should therefore always consult the state of the CKA OTP\_PIN\_REQUIREMENT  
 13027 attribute before each call to **C\_SignInit**, as the value of this attribute may change dynamically.  
 13028 For OTP tokens with multiple keys, the keys may be enumerated using **C\_FindObjects**. The  
 13029 **CKA OTP SERVICE IDENTIFIER** and/or the **CKA OTP SERVICE LOGO** attribute may be used to  
 13030 distinguish between keys. The actual choice of key for a particular operation is however application-  
 13031 specific and beyond the scope of this document.  
 13032 For all OTP keys, the CKA\_ALLOWED\_MECHANISMS attribute should be set as required.

## 13033 **6.53.6 OTP-related notifications**

13034 This document extends the set of defined notifications as follows:

13035 <b>CKN OTP CHANGED</b>	Cryptoki is informing the application that the OTP for a key on a 13036 connected token just changed. This notification is particularly useful 13037 when applications wish to display the current OTP value for time- 13038 based mechanisms.
------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 13039 **6.53.7 OTP mechanisms**

13040 The following table shows, for the OTP mechanisms defined in this document, their support by different  
 13041 cryptographic operations. For any particular token, of course, a particular operation may well support  
 13042 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one  
 13043 mechanism for some operation supports any other mechanism for any other operation (or even supports  
 13044 that same mechanism for any other operation).

13045 *Table 226: OTP mechanisms vs. applicable functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SECURID_KEY_GEN					✓		
CKM_SECURID		✓					
CKM_HOTP_KEY_GEN					✓		
CKM_HOTP		✓					
CKM_ACTI_KEY_GEN					✓		
CKM_ACTI		✓					

13046 The remainder of this section will present in detail the OTP mechanisms and the parameters that are  
 13047 supplied to them.

## 13048 **6.53.7.1 OTP mechanism parameters**

### 13049 ♦ **CK OTP PARAM TYPE**

13050 **CK OTP PARAM TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

13051       **typedef CK ULONG CK OTP PARAM TYPE;**

13052 The following **CK OTP PARAM TYPE** types are defined:

13053 *Table 227, OTP parameter types*

Parameter	Data type	Meaning
CK OTP PIN	RFC 2279 string	A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.
CK OTP CHALLENGE	Byte array	Challenge to use when computing or verifying challenge-based OTP values.
CK OTP TIME	RFC 2279 string	UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values.
CK OTP COUNTER	Byte array	Counter value to use when computing or verifying counter-based OTP values.
CK OTP FLAGS	CK_FLAGS	Bit flags indicating the characteristics of the sought OTP as defined below.
CK OTP OUTPUT LENGTH	CK ULONG	Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.
CK OTP OUTPUT FORMAT	CK ULONG	Returned OTP format (allowed values are the same as for CKA OTP FORMAT). This parameter is only intended for <b>C_Sign</b> output, see paragraphs below. When not present, the returned OTP format will be the same as the value of the CKA OTP FORMAT attribute for the key in question.
CK OTP VALUE	Byte array	An actual OTP value. This parameter type is intended for <b>C_Sign</b> output, see paragraphs below.

13054

13055 The following table defines the possible values for the CK OTP FLAGS type:

13056 *Table 228: OTP Mechanism Flags*

Bit flag	Mask	Meaning
CKF_NEXT OTP	0x00000001	True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if the CKF_NEXT OTP flag is set and the OTP mechanism in question does not support the concept of "next" OTP or the library is not capable of generating the next OTP <sup>9</sup> .

---

<sup>9</sup> Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by C\_Sign so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to C\_Sign should it turn out that the library does not support the CKF\_NEXT OTP flag.

Bit flag	Mask	Meaning
CKF_EXCLUDE_TIME	0x00000002	True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_COUNTER	0x00000004	True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_CHALLENGE	0x00000008	True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_PIN	0x00000010	True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_USER_FRIENDLY OTP	0x00000020	True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK_OTP_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if this flag is set when CKA_OTP_USER_FRIENDLY_MODE for the key in question is CK_FALSE.

13057 Note: Even if CKA\_OTP\_FORMAT is not set to CK\_OTP\_FORMAT\_BINARY, then there may still be  
 13058 value in setting the CKF\_USER\_FRIENDLY OTP flag (assuming CKA\_OTP\_USER\_FRIENDLY\_MODE  
 13059 is CK\_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may  
 13060 become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned  
 13061 OTP value to a user should not set the CKF\_USER\_FRIENDLY OTP flag.

#### ◆ CK\_OTP\_PARAM; CK\_OTP\_PARAM\_PTR

13063 **CK\_OTP\_PARAM** is a structure that includes the type, value, and length of an OTP parameter. It is  
 13064 defined as follows:

```

13065     typedef struct CK OTP PARAM {
13066         CK OTP PARAM TYPE type;
13067         CK VOID PTR pValue;
13068         CK ULONG ulValueLen;
13069     } CK OTP PARAM;
13070
13071     The fields of the structure have the following meanings:
13072         type      the parameter type
13073         pValue    pointer to the value of the parameter
13074         ulValueLen length in bytes of the value
13075
13076     If a parameter has no value, then ulValueLen = 0, and the value of pValue is irrelevant. Note that pValue 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).
13077     CK OTP PARAM PTR is a pointer to a CK OTP PARAM.
13078
13079     ◆ CK OTP PARAMS; CK OTP PARAMS PTR
13080     CK OTP PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:
13081
13082     typedef struct CK OTP PARAMS {
13083         CK OTP PARAM PTR pParams;
13084         CK ULONG ulCount;
13085     } CK OTP PARAMS;
13086
13087     The fields of the structure have the following meanings:
13088         pParams    pointer to an array of OTP parameters
13089         ulCount    the number of parameters in the array
13090
13091     When calling C_SignInit or C_VerifyInit with a mechanism that takes a CK OTP PARAMS structure as a parameter, the CK OTP PARAMS structure shall be populated in accordance with the
13092     CKA OTP X REQUIREMENT key attributes for the identified key, where X is PIN, CHALLENGE, TIME,
13093     or COUNTER.
13094
13095     For example, if CKA OTP TIME REQUIREMENT = CK OTP PARAM MANDATORY, then the
13096     CK OTP TIME parameter shall be present. If CKA OTP TIME REQUIREMENT =
13097     CK OTP PARAM OPTIONAL, then a CK OTP TIME parameter may be present. If it is not present,
13098     then the library may collect it (during the C_Sign call). If CKA OTP TIME REQUIREMENT =
13099     CK OTP PARAM IGNORED, then a provided CK OTP TIME parameter will always be ignored.
13100     Additionally, a provided CK OTP TIME parameter will always be ignored if CKF EXCLUDE_TIME is set
13101     in a CK OTP FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value
13102     itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is
13103     an error (CKR_MECHANISM_PARAM_INVALID) to set the CKF EXCLUDE_TIME flag when the
13104     CKA OTP TIME REQUIREMENT attribute is CK OTP PARAM MANDATORY.
13105
13106     The above discussion holds for all CKA OTP X REQUIREMENT attributes (i.e.,
13107     CKA OTP PIN REQUIREMENT, CKA OTP CHALLENGE REQUIREMENT,
13108     CKA OTP COUNTER REQUIREMENT, CKA OTP TIME REQUIREMENT). A library may set a
13109     particular CKA OTP X REQUIREMENT attribute to CK OTP PARAM OPTIONAL even if it is required
13110     by the mechanism as long as the token (or the library itself) has the capability of providing the value to the
computation. One example of this is a token with an on-board clock.

```

13111 In addition, applications may use the CK OTP FLAGS, the CK OTP OUTPUT FORMAT and the  
13112 CKA OTP LENGTH parameters to set additional parameters.  
13113

13114 ◆ **CK OTP SIGNATURE INFO, CK OTP SIGNATURE INFO PTR**

13115 **CK OTP SIGNATURE INFO** is a structure that is returned by all OTP mechanisms in successful calls to  
13116 **C\_Sign** (**C\_SignFinal**). The structure informs applications of actual parameter values used in particular  
13117 OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key  
13118 belongs to the class **CKO OTP KEY** and is defined as follows:

```
13119     typedef struct CK OTP SIGNATURE INFO {  
13120         CK OTP PARAM PTR pParams;  
13121         CK ULONG ulCount;  
13122     } CK OTP SIGNATURE INFO;
```

13123 The fields of the structure have the following meanings:

13124 pParams pointer to an array of OTP parameter values  
13125 ulCount the number of parameters in the array

13126 After successful calls to **C\_Sign** or **C\_SignFinal** with an OTP mechanism, the *pSignature* parameter will  
13127 be set to point to a **CK OTP SIGNATURE INFO** structure. One of the parameters in this structure will be  
13128 the OTP value itself, identified with the **CK OTP VALUE** tag. Other parameters may be present for  
13129 informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP  
13130 validations, authentication protocols may permit authenticating parties to send some or all of these  
13131 parameters in addition to OTP values themselves. Applications should therefore check for their presence  
13132 in returned **CK OTP SIGNATURE INFO** values whenever such circumstances apply.

13133 Since **C\_Sign** and **C\_SignFinal** follows the convention described in Section 5.2 on producing output, a  
13134 call to **C\_Sign** (or **C\_SignFinal**) with *pSignature* set to **NULL\_PTR** will return (in the *pulSignatureLen*  
13135 parameter) the required number of bytes to hold the **CK OTP SIGNATURE INFO** structure as well as all  
13136 the data in all its **CK OTP PARAM** components. If an application allocates a memory block based on  
13137 this information, it shall therefore not subsequently de-allocate components of such a received value but  
13138 rather de-allocate the complete **CK OTP PARAMS** structure itself. A Cryptoki library that is called with a  
13139 non-**NULL** *pSignature* pointer will assume that it points to a *contiguous* memory block of the size  
13140 indicated by the *pulSignatureLen* parameter.

13141 When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself,  
13142 e.g. the value of the **CK OTP VALUE** component of a **CK OTP PARAM** structure returned by a call to  
13143 **C\_Sign**. The **CK OTP PARAM** value supplied in the **C\_VerifyInit** call sets the values to use in the  
13144 verification operation.

13145 **CK OTP SIGNATURE INFO PTR** points to a **CK OTP SIGNATURE INFO**.

## 13146 6.53.8 RSA SecurID

### 13147 6.53.8.1 RSA SecurID secret key objects

13148 RSA SecurID secret key objects (object class **CKO OTP KEY**, key type **CKK SECURID**) hold RSA  
13149 SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in  
13150 addition to the common attributes defined for this object class:

13151 *Table 229, RSA SecurID secret key object attributes*

Attribute	Data type	Meaning
CKA OTP TIME INTERVAL <sup>1</sup>	CK ULONG	Interval between OTP values produced with this key, in seconds. Default is 60.

13152 <sup>1</sup> Refer to Table 11 for footnotes

13153 The following is a sample template for creating an RSA SecurID secret key object:

```

13154     CK_OBJECT_CLASS class = CKO OTP KEY;
13155     CK_KEY_TYPE keyType = CKK SECURID;
13156     CK_DATE endDate = {...};
13157     CK_UTF8CHAR label[] = "RSA SecurID secret key object";
13158     CK_BYTE keyId[] = {...};
13159     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;
13160     CK ULONG outputLength = 6;
13161     CK ULONG needPIN = CK OTP PARAM MANDATORY;
13162     CK ULONG timeInterval = 60;
13163     CK_BYTE value[] = {...};
13164     CK_BBOOL true = CK TRUE;
13165     CK_ATTRIBUTE template[] = {
13166         {CKA_CLASS, &class, sizeof(class)},
13167         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13168         {CKA_END_DATE, &endDate, sizeof(endDate)},
13169         {CKA_TOKEN, &true, sizeof(true)},
13170         {CKA_SENSITIVE, &true, sizeof(true)},
13171         {CKA_LABEL, label, sizeof(label)-1},
13172         {CKA_SIGN, &true, sizeof(true)},
13173         {CKA_VERIFY, &true, sizeof(true)},
13174         {CKA_ID, keyId, sizeof(keyId)},
13175         {CKA OTP FORMAT, &outputFormat, sizeof(outputFormat)},
13176         {CKA OTP LENGTH, &outputLength, sizeof(outputLength)},
13177         {CKA OTP PIN REQUIREMENT, &needPIN, sizeof(needPIN)},
13178         {CKA OTP TIME INTERVAL, &timeInterval,
13179             sizeof(timeInterval)},
13180         {CKA_VALUE, value, sizeof(value)}}
13181     };

```

### 13182 6.53.8.2 RSA SecurID key generation

13183 The RSA SecurID key generation mechanism, denoted **CKM SECURID KEY GEN**, is a key generation  
 13184 mechanism for the RSA SecurID algorithm.

13185 It does not have a parameter.

13186 The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the  
 13187 template for the key.

13188 The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE\_LEN**, and  
 13189 **CKA\_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be  
 13190 specified in the template for the key, or else are assigned default initial values.

13191 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 13192 specify the supported range of SecurID key sizes, in bytes.

### 13193 6.53.8.3 SecurID OTP generation and validation

13194 **CKM SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.

13195 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.

13196 When signing or verifying using the **CKM SECURID** mechanism, *pData* shall be set to **NULL\_PTR** and  
 13197 *ulDataLen* shall be set to 0.

13198 **6.53.8.4 Return values**

13199 Support for the CKM\_SECURID mechanism extends the set of return values for C\_Verify with the  
13200 following values:

- 13201 • CKR\_NEW\_PIN\_MODE: The supplied OTP was not accepted and the library requests a new OTP  
13202 computed using a new PIN. The new PIN is set through means out of scope for this document.
- 13203 • CKR\_NEXT OTP: The supplied OTP was correct but indicated a larger than normal drift in the  
13204 token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the  
13205 application should provide the next one-time password to the library for verification.

13206 **6.53.9 OATH HOTP**

13207 **6.53.9.1 OATH HOTP secret key objects**

13208 HOTP secret key objects (object class **CKO OTP KEY**, key type **CKK\_HOTP**) hold generic secret keys  
13209 and associated counter values.

13210 The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a  
13211 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may  
13212 not be revealed if the object has its **CKA SENSITIVE** attribute set to CK\_TRUE or its  
13213 **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.

13214 For HOTP keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.  
13215 network byte order) form. The same holds true for a **CK OTP COUNTER** value in a **CK OTP PARAM**  
13216 structure.

13217 The following is a sample template for creating a HOTP secret key object:

```
13218     CK_OBJECT_CLASS class = CKO OTP KEY;
13219     CK_KEY_TYPE keyType = CKK_HOTP;
13220     CK_UTF8CHAR label[] = "HOTP secret key object";
13221     CK_BYTE keyId[] = {...};
13222     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;
13223     CK ULONG outputLength = 6;
13224     CK DATE endDate = {...};
13225     CK_BYTE counterValue[8] = {0};
13226     CK_BYTE value[] = {...};
13227         CK_BBOOL true = CK_TRUE;
13228     CK_ATTRIBUTE template[] = {
13229         {CKA CLASS, &class, sizeof(class)},
13230         {CKA KEY TYPE, &keyType, sizeof(keyType)},
13231         {CKA END DATE, &endDate, sizeof(endDate)},
13232         {CKA TOKEN, &true, sizeof(true)},
13233         {CKA SENSITIVE, &true, sizeof(true)},
13234         {CKA LABEL, label, sizeof(label)-1},
13235         {CKA SIGN, &true, sizeof(true)},
13236         {CKA VERIFY, &true, sizeof(true)},
13237         {CKA ID, keyId, sizeof(keyId)},
13238         {CKA OTP FORMAT, &outputFormat, sizeof(outputFormat)},
13239         {CKA OTP LENGTH, &outputLength, sizeof(outputLength)},
13240         {CKA OTP COUNTER, counterValue, sizeof(counterValue)},
13241         {CKA VALUE, value, sizeof(value)}
13242     };
```

13243 **6.53.9.2 HOTP key generation**

13244 The HOTP key generation mechanism, denoted **CKM\_HOTP\_KEY\_GEN**, is a key generation mechanism  
13245 for the HOTP algorithm.

13246 It does not have a parameter.

13247 The mechanism generates HOTP keys with a particular set of attributes as specified in the template for  
13248 the key.

13249 The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA OTP COUNTER**,  
13250 **CKA\_VALUE** and **CKA\_VALUE\_LEN** attributes to the new key. Other attributes supported by the HOTP  
13251 key type may be specified in the template for the key, or else are assigned default initial values.

13252 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13253 specify the supported range of HOTP key sizes, in bytes.

13254 **6.53.9.3 HOTP OTP generation and validation**

13255 **CKM\_HOTP** is the mechanism for the retrieval and verification of HOTP OTP values based on the current  
13256 internal counter, or a provided counter.

13257 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.

13258 As for the **CKM\_SECURID** mechanism, when signing or verifying using the **CKM\_HOTP** mechanism,  
13259 *pData* shall be set to **NULL\_PTR** and *ulDataLen* shall be set to 0.

13260 For verify operations, the counter value **CK OTP COUNTER** must be provided as a **CK OTP PARAM**  
13261 parameter to **C VerifyInit**. When verifying an OTP value using the **CKM\_HOTP** mechanism, *pSignature*  
13262 shall be set to the OTP value itself, e.g. the value of the **CK OTP VALUE** component of a  
13263 **CK OTP PARAM** structure in the case of an earlier call to **C Sign**.

13264 **6.53.10 ActivIdentity ACTI**

13265 **6.53.10.1 ACTI secret key objects**

13266 ACTI secret key objects (object class **CKO OTP KEY**, key type **CKK\_ACTI**) hold ActivIdentity ACTI  
13267 secret keys.

13268 For ACTI keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.  
13269 network byte order) form. The same holds true for the **CK OTP COUNTER** value in the  
13270 **CK OTP PARAM** structure.

13271 The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a  
13272 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may  
13273 not be revealed if the object has its **CKA SENSITIVE** attribute set to CK\_TRUE or its  
13274 **CKA EXTRACTABLE** attribute set to CK\_FALSE.

13275 The **CKA OTP TIME** value may be set at key generation; however, some tokens may set it to a fixed  
13276 initial value. Depending on the token's security policy, this value may not be modified and/or may not be  
13277 revealed if the object has its **CKA SENSITIVE** attribute set to CK\_TRUE or its **CKA EXTRACTABLE**  
13278 attribute set to CK\_FALSE.

13279 The following is a sample template for creating an ACTI secret key object:

```
13280     CK_OBJECT_CLASS class = CKO OTP KEY;  
13281     CK KEY TYPE keyType = CKK_ACTI;  
13282     CK_UTF8CHAR label[] = "ACTI secret key object";  
13283     CK_BYTE keyId[] = {...};  
13284     CK ULONG outputFormat = CK OTP FORMAT DECIMAL;  
13285     CK ULONG outputLength = 6;  
13286     CK DATE endDate = {...};  
13287     CK_BYTE counterValue[8] = {0};
```

```

13288     CK_BYTE value[] = { ... };
13289     CK_BBOOL true = CK_TRUE;
13290     CK_ATTRIBUTE template[] = {
13291         { CKA_CLASS, &class, sizeof(class) },
13292         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
13293         { CKA_END_DATE, &endDate, sizeof(endDate) },
13294         { CKA_TOKEN, &true, sizeof(true) },
13295         { CKA_SENSITIVE, &true, sizeof(true) },
13296         { CKA_LABEL, label, sizeof(label)-1 },
13297         { CKA_SIGN, &true, sizeof(true) },
13298         { CKA_VERIFY, &true, sizeof(true) },
13299         { CKA_ID, keyId, sizeof(keyId) },
13300         { CKA OTP_FORMAT, &outputFormat,
13301             sizeof(outputFormat) },
13302         { CKA OTP_LENGTH, &outputLength,
13303             sizeof(outputLength) },
13304         { CKA OTP_COUNTER, counterValue,
13305             sizeof(counterValue) },
13306         { CKA_VALUE, value, sizeof(value) }
13307     };

```

### 13308 **6.53.10.2 ACTI key generation**

13309 The ACTI key generation mechanism, denoted **CKM\_ACTI\_KEY\_GEN**, is a key generation mechanism  
 13310 for the ACTI algorithm.

13311 It does not have a parameter.

13312 The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the  
 13313 key.

13314 The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE** and  
 13315 **CKA\_VALUE\_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be  
 13316 specified in the template for the key, or else are assigned default initial values.

13317 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 13318 specify the supported range of ACTI key sizes, in bytes.

### 13319 **6.53.10.3 ACTI OTP generation and validation**

13320 **CKM\_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

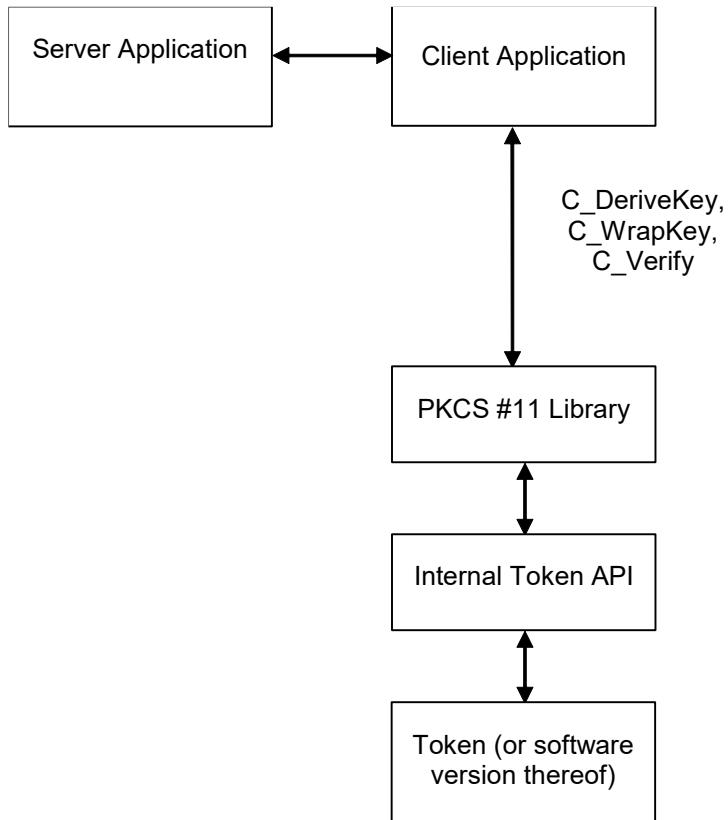
13321 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.

13322 When signing or verifying using the **CKM\_ACTI** mechanism, *pData* shall be set to **NONE\_PTR** and  
 13323 *ulDataLen* shall be set to 0.

13324 When verifying an OTP value using the **CKM\_ACTI** mechanism, *pSignature* shall be set to the OTP value  
 13325 itself, e.g. the value of the **CK OTP VALUE** component of a **CK OTP PARAM** structure in the case of  
 13326 an earlier call to **C\_Sign**.

13327 **6.54 CT-KIP**

13328 **6.54.1 Principles of Operation**



13329

13330 *Figure 5: PKCS #11 and CT-KIP integration*

13331 Figure 5 shows an integration of PKCS #11 into an application that generates cryptographic keys through  
13332 the use of CT-KIP. The application invokes **C\_DeriveKey** to derive a key of a particular type on the token.  
13333 The key may subsequently be used as a basis to e.g., generate one-time password values. The  
13334 application communicates with a CT-KIP server that participates in the key derivation and stores a copy  
13335 of the key in its database. The key is transferred to the server in wrapped form, after a call to  
13336 **C\_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls  
13337 to **C\_Verify**.

13338 **6.54.2 Mechanisms**

13339 The following table shows, for the mechanisms defined in this document, their support by different  
13340 cryptographic operations. For any particular token, of course, a particular operation may well support  
13341 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one  
13342 mechanism for some operation supports any other mechanism for any other operation (or even supports  
13343 that same mechanism for any other operation).

13344 *Table 230: CT-KIP Mechanisms vs. applicable functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR <sup>1</sup>	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_KIP_DERIVE							✓
CKM_KIP_WRAP						✓	
CKM_KIP_MAC		✓					

13345 The remainder of this section will present in detail the mechanisms and the parameters that are supplied  
 13346 to them.

### 13347 6.54.3 Definitions

13348 Mechanisms:

13349 CKM\_KIP\_DERIVE

13350 CKM\_KIP\_WRAP

13351 CKM\_KIP\_MAC

### 13352 6.54.4 CT-KIP Mechanism parameters

#### 13353 ◆ CK\_KIP\_PARAMS; CK\_KIP\_PARAMS\_PTR

13354 CK\_KIP\_PARAMS is a structure that provides the parameters to all the CT-KIP related mechanisms: The  
 13355 CKM\_KIP\_DERIVE key derivation mechanism, the CKM\_KIP\_WRAP key wrap and key unwrap  
 13356 mechanism, and the CKM\_KIP\_MAC signature mechanism. The structure is defined as follows:

```
13357     typedef struct CK_KIP_PARAMS {
13358         CK_MECHANISM_PTR pMechanism;
13359         CK_OBJECT_HANDLE hKey;
13360         CK_BYTE_PTR pSeed;
13361         CK_ULONG ulSeedLen;
13362     } CK_KIP_PARAMS;
```

13363 The fields of the structure have the following meanings:

13364 pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-  
 13365 256)

13366 hKey handle to a key that will contribute to the entropy of the derived key  
 13367 (CKM\_KIP\_DERIVE) or will be used in the MAC operation  
 13368 (CKM\_KIP\_MAC)

13369 pSeed pointer to an input seed

13370 ulSeedLen length in bytes of the input seed

13371 CK\_KIP\_PARAMS\_PTR is a pointer to a CK\_KIP\_PARAMS structure.

### 13372 6.54.5 CT-KIP key derivation

13373 The CT-KIP key derivation mechanism, denoted CKM\_KIP\_DERIVE, is a key derivation mechanism that  
 13374 is capable of generating secret keys of potentially any type, subject to token limitations.

13375 It takes a parameter of type CK\_KIP\_PARAMS which allows for the passing of the desired underlying  
 13376 cryptographic mechanism as well as some other data. In particular, when the hKey parameter is a handle

13377 to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C\_DeriveKey**.  
 13378 The *pSeed* parameter may be used to seed the key derivation operation.  
 13379 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the  
 13380 template for the key.  
 13381 The mechanism contributes the **CKA\_CLASS** and **CKA\_VALUE** attributes to the new key. Other  
 13382 attributes supported by the key type may be specified in the template for the key, or else will be assigned  
 13383 default initial values. Since the mechanism is generic, the **CKA\_KEY\_TYPE** attribute should be set in the  
 13384 template, if the key is to be used with a particular mechanism.

### 13385 **6.54.6 CT-KIP key wrap and key unwrap**

13386 The CT-KIP key wrap and unwrap mechanism, denoted **CKM\_KIP\_WRAP**, is a key wrap mechanism that  
 13387 is capable of wrapping and unwrapping generic secret keys.  
 13388 It takes a parameter of type **CK\_KIP\_PARAMS**, which allows for the passing of the desired underlying  
 13389 cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of  
 13390 **CK\_KIP\_PARAMS**.

### 13391 **6.54.7 CT-KIP signature generation**

13392 The CT-KIP signature (MAC) mechanism, denoted **CKM\_KIP\_MAC**, is a mechanism used to produce a  
 13393 message authentication code of arbitrary length. The keys it uses are secret keys.  
 13394 It takes a parameter of type **CK\_KIP\_PARAMS**, which allows for the passing of the desired underlying  
 13395 cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed*  
 13396 and the *uSeedLen* parameters of **CT\_KIP\_PARAMS**.  
 13397 This mechanism produces a MAC of the length specified by *puSignatureLen* parameter in calls to  
 13398 **C\_Sign**.  
 13399 If a call to **C\_Sign** with this mechanism fails, then no output will be generated.

## 13400 **6.55 GOST 28147-89**

13401 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

13402

13403 *Table 231, GOST 28147-89 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOST28147_KEY_GEN					✓		
CKM_GOST28147_ECB	✓					✓	
CKM_GOST28147	✓					✓	
CKM_GOST28147_MAC		✓					
CKM_GOST28147_KEY_WRAP						✓	

13404

### 13405 **6.55.1 Definitions**

13406 This section defines the key type “CKK\_GOST28147” for type **CK\_KEY\_TYPE** as used in the  
 13407 **CKA\_KEY\_TYPE** attribute of key objects and domain parameter objects.

13408 Mechanisms:  
 13409 CKM\_GOST28147\_KEY\_GEN  
 13410 CKM\_GOST28147\_ECB  
 13411 CKM\_GOST28147  
 13412 CKM\_GOST28147\_MAC  
 13413 CKM\_GOST28147\_KEY\_WRAP

## 13414 6.55.2 GOST 28147-89 secret key objects

13415 GOST 28147-89 secret key objects (object class **CKO\_SECRET\_KEY**, key type **CKK\_GOST28147**) hold  
 13416 GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in  
 13417 addition to the common attributes defined for this object class:

13418 *Table 232, GOST 28147-89 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	32 bytes in little endian order
CKA_GOST28147_PARAMS <sup>1,3,5</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89.  When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID

13419 - Refer to Table 11 for footnotes

13420 The following is a sample template for creating a GOST 28147-89 secret key object:

```
13421 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13422 CK_KEY_TYPE keyType = CKK_GOST28147;
13423 CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";
13424 CK_BYTE value[32] = {...};
13425 CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
13426             0x02, 0x1f, 0x00};
13427 CK_BBOOL true = CK_TRUE;
13428 CK_ATTRIBUTE template[] = {
13429     {CKA_CLASS, &class, sizeof(class)},
13430     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13431     {CKA_TOKEN, &true, sizeof(true)},
13432     {CKA_LABEL, label, sizeof(label)-1},
13433     {CKA_ENCRYPT, &true, sizeof(true)},
13434     {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},
13435     {CKA_VALUE, value, sizeof(value)}}
13436 };
```

## 13437 6.55.3 GOST 28147-89 domain parameter objects

13438 GOST 28147-89 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type  
 13439 **CKK\_GOST28147**) hold GOST 28147-89 domain parameters.

13440 The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the  
 13441 common attributes defined for this object class:

13442 Table 233, GOST 28147-89 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE <sup>1</sup>	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type <i>Gost28147-89-ParamSetParameters</i> )
CKA_OBJECT_ID <sup>1</sup>	Byte array	DER-encoding of the object identifier indicating the domain parameters

13443 Refer to Table 11 for footnotes

13444 For any particular token, there is no guarantee that a token supports domain parameters loading up  
13445 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should  
13446 take in account that **CKA\_VALUE** attribute may be inaccessible.

13447 The following is a sample template for creating a GOST 28147-89 domain parameter object:

```
13448 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13449 CK_KEY_TYPE keyType = CKK_GOST28147;
13450 CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic
13451     parameters object";
13452 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
13453     0x1f, 0x00};
13454 CK_BYTE value[] = {
13455     0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,
13456     0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0x9b, 0x02, 0x98, 0x75, 0x61, 0x3b,
13457     0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0x0c, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,
13458     0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0xb3, 0x8e, 0xb4, 0x2c, 0x77, 0xe7,
13459     0xcd, 0x46, 0xca, 0xfa, 0xd6, 0x6a, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0xa4,
13460     0xab, 0x03, 0xf2, 0x21, 0x65, 0xb8, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,
13461     0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,
13462     0x00, 0x05, 0x00
13463 };
13464 CK_BBOOL true = CK_TRUE;
13465 CK_ATTRIBUTE template[] = {
13466     {CKA_CLASS, &class, sizeof(class)},
13467     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13468     {CKA_TOKEN, &true, sizeof(true)},
13469     {CKA_LABEL, label, sizeof(label)-1},
13470     {CKA_OBJECT_ID, oid, sizeof(oid)},
13471     {CKA_VALUE, value, sizeof(value)}
13472 };
```

## 13473 6.55.4 GOST 28147-89 key generation

13474 The GOST 28147-89 key generation mechanism, denoted **CKM\_GOST28147\_KEY\_GEN**, is a key  
13475 generation mechanism for GOST 28147-89.

13476 It does not have a parameter.

13477 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
13478 key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object  
13479 class **CKO\_SECRET\_KEY**.

13480 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** are not  
13481 used.

## 13482 6.55.5 GOST 28147-89-ECB

13483 GOST 28147-89-ECB, denoted **CKM\_GOST28147\_ECB**, is a mechanism for single and multiple-part  
13484 encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic  
13485 codebook mode.

13486 It does not have a parameter.

13487 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
13488 wrap/unwrap every secret key that it supports.

13489 For wrapping (**C\_WrapKey**), the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key  
13490 that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple  
13491 of the block size.

13492 For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result  
13493 according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports  
13494 it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the  
13495 **CKA\_VALUE** attribute of the new key.

13496 Constraints on key types and the length of data are summarized in the following table:

13497 *Table 234, GOST 28147-89-ECB: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_Decrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_WrapKey	CKK_GOST28147	Any	Input length rounded up to multiple of block size
C_UnwrapKey	CKK_GOST28147	Multiple of block size	Determined by type of key being unwrapped

13498

13499 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13500 are not used.

## 13501 6.55.6 GOST 28147-89 encryption mode except ECB

13502 GOST 28147-89 encryption mode except ECB, denoted **CKM\_GOST28147**, is a mechanism for single  
13503 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on  
13504 [GOST 28147-89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2.  
13505 Encryption's parameters are specified in object identifier of attribute **CKA\_GOST28147\_PARAMS**.

13506 It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero  
13507 initialization vector is used.

13508 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to  
13509 wrap/unwrap every secret key that it supports.

13510 For wrapping (**C\_WrapKey**), the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key  
13511 that is wrapped.

13512 For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as  
13513 the **CKA\_VALUE** attribute of the new key.

13514 Constraints on key types and the length of data are summarized in the following table:

13515 *Table 235, GOST 28147-89 encryption modes except ECB: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_GOST28147	Any	For counter mode and CFB is the same as input length. For CBC is the same as input length padded on the trailing end with up to block size so that the resulting length is a multiple of the block size
C_Decrypt	CKK_GOST28147	Any	
C_WrapKey	CKK_GOST28147	Any	
C_UnwrapKey	CKK_GOST28147	Any	

13516

13517 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13518 are not used.

### 13519 6.55.7 GOST 28147-89-MAC

13520 GOST 28147-89-MAC, denoted **CKM\_GOST28147\_MAC**, is a mechanism for data integrity and  
13521 authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

13522 MACing parameters are specified in object identifier of attribute **CKA\_GOST28147\_PARAMS**.

13523 The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block  
13524 produced in the MACing process.

13525 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a  
13526 zero initialization vector is used.

13527 Constraints on key types and the length of data are summarized in the following table:

13528 *Table 236, GOST28147-89-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_GOST28147	Any	4 bytes
C_Verify	CKK_GOST28147	Any	4 bytes

13529

13530 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13531 are not used.

13532

### 13533 6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89

13534 GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by  
13535 **CKM\_GOST28147\_KEY\_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on  
13536 GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation  
13537 mechanism for GOST 28147-89.

13538 For wrapping (**C\_WrapKey**), the mechanism first computes MAC from the value of the **CKA\_VALUE**  
13539 attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA\_VALUE**  
13540 attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

13541 For unwrapping (**C\_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that  
13542 was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC  
13543 has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is  
13544 disallowed. The mechanism contributes the result as the **CKA\_VALUE** attribute of the unwrapped key.

13545 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a  
13546 zero initialization vector is used.

13547 Constraints on key types and the length of data are summarized in the following table:

13548 Table 237, GOST 28147-89 keys as KEK: Key and Data Length

Function	Key type	Input length	Output length
C_WrapKey	CKK_GOST28147	32 bytes	36 bytes
C_UnwrapKey	CKK_GOST28147	32 bytes	36 bytes

13549

13550 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13551 are not used.

13552

## 13553 6.56 GOST R 34.11-94

13554 GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit  
13555 message digest defined in [GOST R 34.11-94].

13556

13557 Table 238, GOST R 34.11-94 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOSTR3411				✓			
CKM_GOSTR3411_HMAC		✓					

13558

### 13559 6.56.1 Definitions

13560 This section defines the key type "CKK\_GOSTR3411" for type CK\_KEY\_TYPE as used in the  
13561 CKA\_KEY\_TYPE attribute of domain parameter objects.

13562 Mechanisms:

13563 CKM\_GOSTR3411

13564 CKM\_GOSTR3411\_HMAC

### 13565 6.56.2 GOST R 34.11-94 domain parameter objects

13566 GOST R 34.11-94 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type  
13567 **CKK\_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

13568 The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the  
13569 common attributes defined for this object class:

13570 Table 239, GOST R 34.11-94 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE <sup>1</sup>	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type <i>GostR3411-94-ParamSetParameters</i> )
CKA_OBJECT_ID <sup>1</sup>	Byte array	DER-encoding of the object identifier indicating the domain parameters

13571 - Refer to Table 11 for footnotes

13572 For any particular token, there is no guarantee that a token supports domain parameters loading up  
 13573 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should  
 13574 take in account that **CKA\_VALUE** attribute may be inaccessible.

13575 The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```

13576     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13577     CK_KEY_TYPE keyType = CKK_GOSTR3411;
13578     CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic
13579         parameters object";
13580     CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
13581         0x1e, 0x00};
13582     CK_BYTE value[] = {
13583         0x30, 0x64, 0x04, 0x40, 0x4e, 0x57, 0x64, 0xd1, 0xab, 0x8d, 0xcb, 0xbf,
13584         0x94, 0x1a, 0x7a, 0x4d, 0x2c, 0xd1, 0x10, 0x10, 0xd6, 0xa0, 0x57, 0x35,
13585         0x8d, 0x38, 0xf2, 0xf7, 0x0f, 0x49, 0xd1, 0x5a, 0xea, 0x2f, 0x8d, 0x94,
13586         0x62, 0xee, 0x43, 0x09, 0xb3, 0xf4, 0xa6, 0xa2, 0x18, 0xc6, 0x98, 0xe3,
13587         0xc1, 0x7c, 0xe5, 0x7e, 0x70, 0x6b, 0x09, 0x66, 0xf7, 0x02, 0x3c, 0x8b,
13588         0x55, 0x95, 0xbf, 0x28, 0x39, 0xb3, 0x2e, 0xcc, 0x04, 0x20, 0x00, 0x00,
13589         0x00, 0x00,
13590         0x00, 0x00,
13591         0x00, 0x00
13592     };
13593     CK_BBOOL true = CK_TRUE;
13594     CK_ATTRIBUTE template[] = {
13595         {CKA_CLASS, &class, sizeof(class)},
13596         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13597         {CKA_TOKEN, &true, sizeof(true)},
13598         {CKA_LABEL, label, sizeof(label)-1},
13599         {CKA_OBJECT_ID, oid, sizeof(oid)},
13600         {CKA_VALUE, value, sizeof(value)}
13601     };

```

### 13602 **6.56.3 GOST R 34.11-94 digest**

13603 GOST R 34.11-94 digest, denoted **CKM\_GOSTR3411**, is a mechanism for message digesting based on  
 13604 GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

13605 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter  
 13606 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]  
 13607 (section 11.2) must be used.

13608 Constraints on the length of input and output data are summarized in the following table. For single-part  
 13609 digesting, the data and the digest may begin at the same location in memory.

13610 *Table 240, GOST R 34.11-94: Data Length*

<b>Function</b>	<b>Input length</b>	<b>Digest length</b>
C_Digest	Any	32 bytes

13611

13612 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 13613 are not used.

## 6.56.4 GOST R 34.11-94 HMAC

GOST R 34.11-94 HMAC mechanism, denoted **CKM\_GOSTR3411\_HMAC**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type **CKK\_GENERIC\_SECRET** or **CKK\_GOST28147**.

To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Signatures (MACs) produced by this mechanism are of 32 bytes long.

Constraints on the length of input and output data are summarized in the following table:

Table 241, GOST R 34.11-94 HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 byte
C_Verify	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 bytes

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

## 6.57 GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

13632

Table 242, GOST R34.10-2001 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	S R & V R	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Deriv e
CKM_GOSTR3410_KEY_PAIR_GEN					✓		
CKM_GOSTR3410		✓ <sup>1</sup>					
CKM_GOSTR3410_WITH_GOSTR3411		✓					
CKM_GOSTR3410_KEY_WRAP						✓	
CKM_GOSTR3410_DERIVE							✓

<sup>1</sup> Single-part operations only

13635

### 6.57.1 Definitions

This section defines the key type "CKK\_GOSTR3410" for type **CK\_KEY\_TYPE** as used in the **CKA\_KEY\_TYPE** attribute of key objects and domain parameter objects.

Mechanisms:

13640 CKM\_GOSTR3410\_KEY\_PAIR\_GEN  
 13641 CKM\_GOSTR3410  
 13642 CKM\_GOSTR3410\_WITH\_GOSTR3411  
 13643 CKM\_GOSTR3410  
 13644 CKM\_GOSTR3410\_KEY\_WRAP  
 13645 CKM\_GOSTR3410\_DERIVE

## 6.57.2 GOST R 34.10-2001 public key objects

13647 GOST R 34.10-2001 public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_GOSTR3410**)  
 13648 hold GOST R 34.10-2001 public keys.

13649 The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the  
 13650 common attributes defined for this object class:

13651 *Table 243, GOST R 34.10-2001 Public Key Object Attributes*

Attribute	Data Type	Meaning
CKA_VALUE <sup>1,4</sup>	Byte array	64 bytes for public key; 32 bytes for each coordinates X and Y of Elliptic Curve point P(X, Y) in little endian order
CKA_GOSTR3410_PARAMS <sup>1,3</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411_PARAMS <sup>1,3,8</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS <sup>8</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

13652 <sup>1</sup>Refer to Table 11 for footnotes

13653 The following is a sample template for creating an GOST R 34.10-2001 public key object:

```

13654 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
13655 CK_KEY_TYPE keyType = CKK_GOSTR3410;
13656 CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";
13657 CK_BYTE gostR3410params_oid[] =
13658     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13659 CK_BYTE gostR3411params_oid[] =
  
```

```

13660     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00 };
13661 CK_BYTE gost28147params_oid[] =
13662     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00 };
13663 CK_BYTE value[64] = {...};
13664 CK_BBOOL true = CK_TRUE;
13665 CK_ATTRIBUTE template[] = {
13666     { CKA_CLASS, &class, sizeof(class) },
13667     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
13668     { CKA_TOKEN, &true, sizeof(true) },
13669     { CKA_LABEL, label, sizeof(label)-1 },
13670     { CKA_GOSTR3410_PARAMS, gostR3410params_oid,
13671         sizeof(gostR3410params_oid) },
13672     { CKA_GOSTR3411_PARAMS, gostR3411params_oid,
13673         sizeof(gostR3411params_oid) },
13674     { CKA_GOST28147_PARAMS, gost28147params_oid,
13675         sizeof(gost28147params_oid) },
13676     { CKA_VALUE, value, sizeof(value) }
13677 } ;

```

### 13678 6.57.3 GOST R 34.10-2001 private key objects

13679 GOST R 34.10-2001 private key objects (object class **CKO\_PRIVATE\_KEY**, key type  
13680 **CKK\_GOSTR3410**) hold GOST R 34.10-2001 private keys.

13681 The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the  
13682 common attributes defined for this object class:

13683 *Table 244, GOST R 34.10-2001 Private Key Object Attributes*

Attribute	Data Type	Meaning
CKA_VALUE <sup>1,4,6,7</sup>	Byte array	32 bytes for private key in little endian order
CKA_GOSTR3410_PARAMS <sup>1,4,6</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001.  When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411_PARAMS <sup>1,4,6,8</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94.  When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS <sup>4,6,8</sup>	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89.

Attribute	Data Type	Meaning
		When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

- 13684 - Refer to Table 11 for footnotes
- 13685 Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key pair, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.
- 13688 The following is a sample template for creating an GOST R 34.10-2001 private key object:
- ```

13690     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
13691     CK_KEY_TYPE keyType = CKK_GOSTR3410;
13692     CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key
13693         object";
13694     CK_BYTE subject[] = {...};
13695     CK_BYTE id[] = {123};
13696     CK_BYTE gostR3410params_oid[] =
13697         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13698     CK_BYTE gostR3411params_oid[] =
13699         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
13700     CK_BYTE gost28147params_oid[] =
13701         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
13702     CK_BYTE value[32] = {...};
13703     CK_BBOOL true = CK_TRUE;
13704     CK_ATTRIBUTE template[] = {
13705         {CKA_CLASS, &class, sizeof(class)},
13706         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13707         {CKA_TOKEN, &true, sizeof(true)},
13708         {CKA_LABEL, label, sizeof(label)-1},
13709         {CKA_SUBJECT, subject, sizeof(subject)},
13710         {CKA_ID, id, sizeof(id)},
13711         {CKA_SENSITIVE, &true, sizeof(true)},
13712         {CKA_SIGN, &true, sizeof(true)},
13713         {CKA_GOSTR3410_PARAMS, gostR3410params_oid,
13714             sizeof(gostR3410params_oid)},
13715         {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
13716             sizeof(gostR3411params_oid)},
13717         {CKA_GOST28147_PARAMS, gost28147params_oid,
13718             sizeof(gost28147params_oid)},
13719         {CKA_VALUE, value, sizeof(value)}}
13720     };
13721

```

## 13722 6.57.4 GOST R 34.10-2001 domain parameter objects

13723 GOST R 34.10-2001 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**, key type  
13724 **CKK\_GOSTR3410**) hold GOST R 34.10-2001 domain parameters.

13725 The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the  
13726 common attributes defined for this object class:

13727 *Table 245, GOST R 34.10-2001 Domain Parameter Object Attributes*

| Attribute                  | Data Type  | Meaning                                                                                                                        |
|----------------------------|------------|--------------------------------------------------------------------------------------------------------------------------------|
| CKA_VALUE <sup>1</sup>     | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type <i>GostR3410-2001-ParamSetParameters</i> ) |
| CKA_OBJECT_ID <sup>1</sup> | Byte array | DER-encoding of the object identifier indicating the domain parameters                                                         |

13728 <sup>1</sup>Refer to Table 11 for footnotes

13729 For any particular token, there is no guarantee that a token supports domain parameters loading up  
13730 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should  
13731 take in account that **CKA\_VALUE** attribute may be inaccessible.

13732 The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```
13733 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
13734 CK_KEY_TYPE keyType = CKK_GOSTR3410;
13735 CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic
13736             parameters object";
13737 CK_BYTE oid[] =
13738     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
13739 CK_BYTE value[] =
13740     {0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x20, 0x5f, 0xbff, 0xf4, 0x98,
13741     0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
13742     0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x51, 0x4c, 0x0c, 0xe9,
13743     0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00,
13744     0x00, 0x00,
13745     0x00, 0x00,
13746     0x00, 0x04, 0x31, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
13747     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x50, 0xfe,
13748     0x8a, 0x18, 0x92, 0x97, 0x61, 0x54, 0xc5, 0x9c, 0xfc, 0x19, 0x3a, 0xcc,
13749     0xf5, 0xb3, 0x02, 0x01, 0x02, 0x02, 0x20, 0x08, 0xe2, 0xa8, 0xa0, 0xe6,
13750     0x51, 0x47, 0xd4, 0xbd, 0x63, 0x16, 0x03, 0x0e, 0x16, 0xd1, 0x9c, 0x85,
13751     0xc9, 0x7f, 0x0a, 0x9c, 0xa2, 0x67, 0x12, 0x2b, 0x96, 0xab, 0xbc, 0xea,
13752     0x7e, 0x8f, 0xc8
13753 };
13754 CK_BBOOL true = CK_TRUE;
13755 CK_ATTRIBUTE template[] = {
13756     {CKA_CLASS, &class, sizeof(class)},
13757     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13758     {CKA_TOKEN, &true, sizeof(true)},
13759     {CKA_LABEL, label, sizeof(label)-1},
13760     {CKA_OBJECT_ID, oid, sizeof(oid)},
13761     {CKA_VALUE, value, sizeof(value)}
13762 };
```

13763

## 13764 6.57.5 GOST R 34.10-2001 mechanism parameters

### 13765 ♦ CK\_GOSTR3410\_KEY\_WRAP\_PARAMS

13766 CK\_GOSTR3410\_KEY\_WRAP\_PARAMS is a structure that provides the parameters to the  
13767 CKM\_GOSTR3410\_KEY\_WRAP mechanism. It is defined as follows:

```
13768     typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {  
13769         CK_BYTE_PTR          pWrapOID;  
13770         CK ULONG             ulWrapOIDLen;  
13771         CK_BYTE_PTR          pUKM;  
13772         CK ULONG             ulUKMLen;  
13773         CK_OBJECT_HANDLE    hKey;  
13774     } CK_GOSTR3410_KEY_WRAP_PARAMS;
```

13775

13776 The fields of the structure have the following meanings:

|                     |                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pWrapOID</i>     | pointer to a data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89. If pointer takes NULL_PTR value in C_WrapKey operation then parameters are specified in object identifier of attribute CKA_GOSTR3411_PARAMS must be used. For C_UnwrapKey operation the pointer is not used and must take NULL_PTR value anytime |
| <i>ulWrapOIDLen</i> | length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89                                                                                                                                                                                                                                                      |
| <i>pUKM</i>         | pointer to a data with UKM. If pointer takes NULL_PTR value in C_WrapKey operation then random value of UKM will be used. If pointer takes non-NULL_PTR value in C_UnwrapKey operation then the pointer value will be compared with UKM value of wrapped key. If these two values do not match the wrapped key will be rejected                                 |
| <i>ulUKMLen</i>     | length of UKM data. If <i>pUKM</i> -pointer is different from NULL_PTR then equal to 8                                                                                                                                                                                                                                                                          |
| <i>hKey</i>         | key handle. Key handle of a sender for C_WrapKey operation. Key handle of a receiver for C_UnwrapKey operation. When key handle takes CK_INVALID_HANDLE value then an ephemeral (one time) key pair of a sender will be used                                                                                                                                    |

13777 CK\_GOSTR3410\_KEY\_WRAP\_PARAMS\_PTR is a pointer to a  
13778 CK\_GOSTR3410\_KEY\_WRAP\_PARAMS.

### 13779 ♦ CK\_GOSTR3410\_DERIVE\_PARAMS

13780 CK\_GOSTR3410\_DERIVE\_PARAMS is a structure that provides the parameters to the  
13781 CKM\_GOSTR3410\_DERIVE mechanism. It is defined as follows:

```
13782     typedef struct CK_GOSTR3410_DERIVE_PARAMS {  
13783         CK EC KDF_TYPE kdf;  
13784         CK BYTE_PTR      pPublicData;  
13785         CK ULONG          ulPublicDataLen;
```

```
13786     CK_BYTE_PTR      pUKM;
13787     CK ULONG          ulUKMLen;
13788 } CK_GOSTR3410_DERIVE_PARAMS;
```

13789

13790 The fields of the structure have the following meanings:

|                                 |                                                                                                                                                                                                |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>kdf</i>                      | additional key diversification algorithm identifier.<br>Possible values are CKD_NULL and<br>CKD_CPDIVERSIFY_KDF. In case of CKD_NULL,<br>result of the key derivation function                 |
|                                 | described in [RFC 4357], section 5.2 is used directly; In<br>case of CKD_CPDIVERSIFY_KDF, the resulting key<br>value is additionally processed with algorithm from [RFC<br>4357], section 6.5. |
| <i>pPublicData</i> <sup>1</sup> | pointer to data with public key of a receiver                                                                                                                                                  |
| <i>ulPublicDataLen</i>          | length of data with public key of a receiver (must be 64)                                                                                                                                      |
| <i>pUKM</i>                     | pointer to a UKM data                                                                                                                                                                          |
| <i>ulUKMLen</i>                 | length of UKM data in bytes (must be 8)                                                                                                                                                        |

13791

13792 1 Public key of a receiver is an octet string of 64 bytes long. The public key octets correspond to the concatenation of X and Y coordinates of a point. Any one of  
13793 them is 32 bytes long and represented in little endian order.

13794 CK\_GOSTR3410\_DERIVE\_PARAMS\_PTR is a pointer to a CK\_GOSTR3410\_DERIVE\_PARAMS.

## 13795 6.57.6 GOST R 34.10-2001 key pair generation

13796 The GOST R 34.10-2001 key pair generation mechanism, denoted  
13797 **CKM\_GOSTR3410\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.

13798 This mechanism does not have a parameter.

13799 The mechanism generates GOST R 34.10-2001 public/private key pairs with particular  
13800 GOST R 34.10-2001 domain parameters, as specified in the **CKA\_GOSTR3410\_PARAMS**,  
13801 **CKA\_GOSTR3411\_PARAMS**, and **CKA\_GOST28147\_PARAMS** attributes of the template for the public  
13802 key. Note that **CKA\_GOST28147\_PARAMS** attribute may not be present in the template.

13803 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
13804 public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE**, and **CKA\_GOSTR3410\_PARAMS**,  
13805 **CKA\_GOSTR3411\_PARAMS**, **CKA\_GOST28147\_PARAMS** attributes to the new private key.

13806 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13807 are not used.

## 13808 6.57.7 GOST R 34.10-2001 without hashing

13809 The GOST R 34.10-2001 without hashing mechanism, denoted **CKM\_GOSTR3410**, is a mechanism for  
13810 single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the  
13811 part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

13812 This mechanism does not have a parameter.

13813 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes  
13814 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values s and r',  
13815 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC  
13816 4490] section 3.2, and [RFC 4491] section 2.2.2.

13817 The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of  
13818 GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

13819 *Table 246, GOST R 34.10-2001 without hashing: Key and Data Length*

| Function              | Key type      | Input length | Output length |
|-----------------------|---------------|--------------|---------------|
| C_Sign <sup>1</sup>   | CKK_GOSTR3410 | 32 bytes     | 64 bytes      |
| C_Verify <sup>1</sup> | CKK_GOSTR3410 | 32 bytes     | 64 bytes      |

13820 <sup>1</sup> Single-part operations only.

13821 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13822 are not used.

## 13823 **6.57.8 GOST R 34.10-2001 with GOST R 34.11-94**

13824 The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM\_GOSTR3410\_WITH\_GOSTR3411**, is a  
13825 mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire  
13826 GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

13827 As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating  
13828 GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are  
13829 specified in object identifier of attribute **CKA\_GOSTR3411\_PARAMS** must be used.

13830 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes  
13831 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values s and r',  
13832 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC  
13833 4490] section 3.2, and [RFC 4491] section 2.2.2.

13834 The input for the mechanism is signed or should be signed message of any length. Single- and multiple-  
13835 part signature operations are available.

13836 *Table 247, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length*

| Function | Key type      | Input length | Output length |
|----------|---------------|--------------|---------------|
| C_Sign   | CKK_GOSTR3410 | Any          | 64 bytes      |
| C_Verify | CKK_GOSTR3410 | Any          | 64 bytes      |

13837 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13838 are not used.

## 13839 **6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001**

13840 GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by  
13841 **CKM\_GOSTR3410\_KEY\_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on  
13842 GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation  
13843 mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used.  
13844 Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section  
13845 4.2.

13846 It has a parameter, a **CK\_GOSTR3410\_KEY\_WRAP\_PARAMS** structure defined in section 6.57.5.

13847 For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as  
13848 the **CKA\_VALUE** attribute of the new key.

13849 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13850 are not used.

## 13851 **6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys**

13852 Common key derivation, denoted **CKM\_GOSTR3410\_DERIVE**, is a mechanism for key derivation with  
13853 assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object

13854 class **CKO\_DOMAIN\_PARAMETERS** and key type **CKK\_GOSTR3410**. An algorithm for key derivation  
13855 from [RFC 4357] (section 5.2) must be used.

13856 The mechanism contributes the result as the **CKA\_VALUE** attribute of the new private key. All other  
13857 attributes must be specified in a template for creating private key object.

## 13858 6.58 ChaCha20

13859 ChaCha20 is a secret-key stream cipher described in [CHACHA].

13860 *Table 248, ChaCha20 Mechanisms vs. Functions*

| Mechanism            | Functions         |               |                      |        |                   |               |        |
|----------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
|                      | Encrypt & Decrypt | Sign & Verify | SR & VR <sup>1</sup> | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_CHACHA20_KEY_GEN |                   |               |                      |        | ✓                 |               |        |
| CKM_CHACHA20         | ✓                 |               |                      |        |                   | ✓             |        |

13861

### 13862 6.58.1 Definitions

13863 This section defines the key type "CKK\_CHACHA20" for type CK\_KEY\_TYPE as used in the  
13864 CKA\_KEY\_TYPE attribute of key objects.

13865 Mechanisms:

13866 CKM\_CHACHA20\_KEY\_GEN

13867 CKM\_CHACHA20

### 13868 6.58.2 ChaCha20 secret key objects

13869 ChaCha20 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_CHACHA20) hold  
13870 ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the  
13871 common attributes defined for this object class:

13872 *Table 249, ChaCha20 Secret Key Object*

| Attribute                    | Data type  | Meaning                                                                    |
|------------------------------|------------|----------------------------------------------------------------------------|
| CKA_VALUE <sup>1,4,6,7</sup> | Byte array | Key length is fixed at 256 bits.<br>Bit length restricted to a byte array. |
| CKA_VALUE_LEN <sup>2,3</sup> | CK ULONG   | Length in bytes of key value                                               |

13873 The following is a sample template for creating a ChaCha20 secret key object:

```
13874 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13875 CK_KEY_TYPE keyType = CKK_CHACHA20;
13876 CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
13877 CK_BYTE value[32] = {...};
13878 CK_BBOOL true = CK_TRUE;
13879 CK_ATTRIBUTE template[] = {
13880     {CKA_CLASS, &class, sizeof(class)},
13881     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13882     {CKA_TOKEN, &true, sizeof(true)},
13883     {CKA_LABEL, label, sizeof(label)-1},
13884     {CKA_ENCRYPT, &true, sizeof(true)},
```

```
13885     { CKA_VALUE, value, sizeof(value) }  
13886 } ;  
13887 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first  
13888 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.
```

### 13889 6.58.3 ChaCha20 mechanism parameters

#### ◆ CK\_CHACHA20\_PARAMS; CK\_CHACHA20\_PARAMS\_PTR

13891 **CK\_CHACHA20\_PARAMS** provides the parameters to the **CKM\_CHACHA20** mechanism. It is defined  
13892 as follows:

```
13893     typedef struct CK_CHACHA20_PARAMS {  
13894         CK_BYTE_PTR    pBlockCounter;  
13895         CK ULONG      blockCounterBits;  
13896         CK_BYTE_PTR    pNonce;  
13897         CK ULONG      ulNonceBits;  
13898     } CK_CHACHA20_PARAMS;
```

13899 The fields of the structure have the following meanings:

|       |                           |                                                                                                          |
|-------|---------------------------|----------------------------------------------------------------------------------------------------------|
| 13900 | <i>pBlockCounter</i>      | <i>pointer to block counter</i>                                                                          |
| 13901 | <i>ulblockCounterBits</i> | <i>length of block counter in bits (can be either 32 or 64)</i>                                          |
| 13902 | <i>pNonce</i>             | <i>nonce (This should be never re-used with the same key.)</i>                                           |
| 13903 | <i>ulNonceBits</i>        | <i>length of nonce in bits (is 64 for original, 96 for IETF and 192 for<br/>13904 xchacha20 variant)</i> |

13905 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)  
13906 it is necessary to address these blocks in random order, thus this counter is exposed here.

13907 **CK\_CHACHA20\_PARAMS\_PTR** is a pointer to **CK\_CHACHA20\_PARAMS**.

### 13908 6.58.4 ChaCha20 key generation

13909 The ChaCha20 key generation mechanism, denoted **CKM\_CHACHA20\_KEY\_GEN**, is a key generation  
13910 mechanism for ChaCha20.

13911 It does not have a parameter.

13912 The mechanism generates ChaCha20 keys of 256 bits.

13913 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
13914 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key  
13915 supports) may be specified in the template for the key, or else are assigned default initial values.

13916 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13917 specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is  
13918 fixed at 256 bits.

13919

### 13920 6.58.5 ChaCha20 mechanism

13921 ChaCha20, denoted **CKM\_CHACHA20**, is a mechanism for single and multiple-part encryption and  
13922 decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and  
13923 handling of their nonces, affecting the safety of using random nonces and the maximum size that can be  
13924 encrypted safely.

13925 Chacha20 has a parameter, **CK\_CHACHA20\_PARAMS**, which indicates the nonce and initial block  
 13926 counter value.  
 13927 Constraints on key types and the length of input and output data are summarized in the following table:  
 13928 *Table 250, ChaCha20: Key and Data Length*

| <b>Function</b> | <b>Key type</b> | <b>Input length</b>                             | <b>Output length</b> | <b>Comments</b> |
|-----------------|-----------------|-------------------------------------------------|----------------------|-----------------|
| C_Encrypt       | ChaCha20        | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part   |
| C_Decrypt       | ChaCha20        | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part   |

13929 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
 13930 specify the supported range of ChaCha20 key sizes, in bits.  
 13931 *Table 251, ChaCha20: Nonce and block counter lengths*

| <b>Variant</b> | <b>Nonce</b> | <b>Block counter</b> | <b>Maximum message</b> | <b>Nonce generation</b>                                                                               |
|----------------|--------------|----------------------|------------------------|-------------------------------------------------------------------------------------------------------|
| original       | 64 bit       | 64 bit               | Virtually unlimited    | 1 <sup>st</sup> msg:<br>nonce <sub>0</sub> =random<br>n <sup>th</sup> msg:<br>nonce <sub>n-1</sub> ++ |
| IETF           | 96 bit       | 32 bit               | Max ~256 GB            | 1 <sup>st</sup> msg:<br>nonce <sub>0</sub> =random<br>n <sup>th</sup> msg:<br>nonce <sub>n-1</sub> ++ |
| XChaCha20      | 192 bit      | 64 bit               | Virtually unlimited    | Each nonce can be randomly generated.                                                                 |

13932 Nonces must not ever be reused with the same key. However due to the birthday paradox the first two  
 13933 variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended  
 13934 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.  
 13935 Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with  
 13936 randomly generated nonces the birthday paradox.

## 13937 **6.59 Salsa20**

13938 Salsa20 is a secret-key stream cipher described in [SALSA].

13939 *Table 252, Salsa20 Mechanisms vs. Functions*

| Mechanism           | Functions         |               |                      |        |                    |               |        |
|---------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
|                     | Encrypt & Decrypt | Sign & Verify | SR & VR <sup>1</sup> | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_SALSA20_KEY_GEN |                   |               |                      |        | ✓                  |               |        |
| CKM_SALSA20         | ✓                 |               |                      |        |                    | ✓             |        |

13940

## 6.59.1 Definitions

13942 This section defines the key type “CKK\_SALSA20” and “CKK\_SALSA20” for type CK\_KEY\_TYPE as  
 13943 used in the CKA\_KEY\_TYPE attribute of key objects.

13944 Mechanisms:

13945       CKM\_SALSA20\_KEY\_GEN

13946       CKM\_SALSA20

## 6.59.2 Salsa20 secret key objects

13948 Salsa20 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_SALSA20) hold Salsa20  
 13949 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common  
 13950 attributes defined for this object class:

13951 *Table 253, ChaCha20 Secret Key Object*

| Attribute                    | Data type  | Meaning                                                                    |
|------------------------------|------------|----------------------------------------------------------------------------|
| CKA_VALUE <sup>1,4,6,7</sup> | Byte array | Key length is fixed at 256 bits.<br>Bit length restricted to a byte array. |
| CKA_VALUE_LEN <sup>2,3</sup> | CK ULONG   | Length in bytes of key value                                               |

13952 The following is a sample template for creating a Salsa20 secret key object:

```
13953     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
13954     CK_KEY_TYPE keyType = CKK_SALSA20;
13955     CK_UTF8CHAR label[] = "A Salsa20 secret key object";
13956     CK_BYTE value[32] = {...};
13957     CK_BBOOL true = CK_TRUE;
13958     CK_ATTRIBUTE template[] = {
13959         {CKA_CLASS, &class, sizeof(class)},
13960         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13961         {CKA_TOKEN, &true, sizeof(true)},
13962         {CKA_LABEL, label, sizeof(label)-1},
13963         {CKA_ENCRYPT, &true, sizeof(true)},
13964         {CKA_VALUE, value, sizeof(value)}}
13965     };
```

13966 CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first  
 13967 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA\_VALUE attribute.

13968 **6.59.3 Salsa20 mechanism parameters**

13969 ◆ **CK\_SALSA20\_PARAMS; CK\_SALSA20\_PARAMS\_PTR**

13970 **CK\_SALSA20\_PARAMS** provides the parameters to the **CKM\_SALSA20** mechanism. It is defined as  
13971 follows:

```
13972     typedef struct CK_SALSA20_PARAMS {  
13973         CK_BYTE_PTR    pBlockCounter;  
13974         CK_BYTE_PTR    pNonce;  
13975         CK ULONG       ulNonceBits;  
13976     } CK_SALSA20_PARAMS;
```

13977 The fields of the structure have the following meanings:

13979        *pBlockCounter*     pointer to block counter (64 bits)

13980        *pNonce*          nonce

13981        *ulNonceBits*     size of the nonce in bits (64 for classic and 192 for XSalsa20)

13982 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)  
13983 it is necessary to address these blocks in random order, thus this counter is exposed here.

13984 **CK\_SALSA20\_PARAMS\_PTR** is a pointer to **CK\_SALSA20\_PARAMS**.

13985 **6.59.4 Salsa20 key generation**

13986 The Salsa20 key generation mechanism, denoted **CKM\_SALSA20\_KEY\_GEN**, is a key generation  
13987 mechanism for Salsa20.

13988 It does not have a parameter.

13989 The mechanism generates Salsa20 keys of 256 bits.

13990 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new  
13991 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key  
13992 supports) may be specified in the template for the key, or else are assigned default initial values.

13993 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
13994 specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed  
13995 at 256 bits.

13996 **6.59.5 Salsa20 mechanism**

13997 Salsa20, denoted **CKM\_SALSA20**, is a mechanism for single and multiple-part encryption and decryption  
13998 based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and  
13999 handling of their nonces, affecting the safety of using random nonces.

14000 Salsa20 has a parameter, **CK\_SALSA20\_PARAMS**, which indicates the nonce and initial block counter  
14001 value.

14002 Constraints on key types and the length of input and output data are summarized in the following table:

14003 *Table 254, Salsa20: Key and Data Length*

| Function  | Key type | Input length | Output length        | Comments      |
|-----------|----------|--------------|----------------------|---------------|
| C_Encrypt | Salsa20  | Any          | Same as input length | No final part |
| C_Decrypt | Salsa20  | Any          | Same as input length | No final part |

14004 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
14005 specify the supported range of ChaCha20 key sizes, in bits.

14006 *Table 255, Salsa20: Nonce sizes*

| Variant  | Nonce   | Maximum message     | Nonce generation                                                                                      |
|----------|---------|---------------------|-------------------------------------------------------------------------------------------------------|
| original | 64 bit  | Virtually unlimited | 1 <sup>st</sup> msg:<br>nonce <sub>0</sub> =random<br>n <sup>th</sup> msg:<br>nonce <sub>n-1</sub> ++ |
| XSalsa20 | 192 bit | Virtually unlimited | Each nonce can be randomly generated.                                                                 |

14007 Nonces must not ever be reused with the same key. However due to the birthday paradox the original  
14008 variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended  
14009 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.  
14010 Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly  
14011 generated nonces the birthday paradox.

## 14012 **6.60 Poly1305**

14013 Poly1305 is a message authentication code designed by D.J Bernstein [**POLY1305**]. Poly1305 takes a  
14014 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

14015 *Table 256, Poly1305 Mechanisms vs. Functions*

| Mechanism            | Functions         |               |                      |        |                   |               |        |
|----------------------|-------------------|---------------|----------------------|--------|-------------------|---------------|--------|
|                      | Encrypt & Decrypt | Sign & Verify | SR & VR <sup>1</sup> | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_POLY1305_KEY_GEN |                   |               |                      |        | ✓                 |               |        |
| CKM_POLY1305         |                   | ✓             |                      |        |                   |               |        |

### 14016 **6.60.1 Definitions**

14017 This section defines the key type “CKK\_POLY1305” for type CK\_KEY\_TYPE as used in the  
14018 CKA\_KEY\_TYPE attribute of key objects.

14019 Mechanisms:

14020 CKM\_POLY1305\_KEY\_GEN

14021 CKM\_POLY1305

### 14022 **6.60.2 Poly1305 secret key objects**

14023 Poly1305 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_POLY1305) hold  
14024 Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the  
14025 common attributes defined for this object class:

14026 *Table 257, Poly1305 Secret Key Object*

| <b>Attribute</b>             | <b>Data type</b> | <b>Meaning</b>                                                             |
|------------------------------|------------------|----------------------------------------------------------------------------|
| CKA_VALUE <sup>1,4,6,7</sup> | Byte array       | Key length is fixed at 256 bits.<br>Bit length restricted to a byte array. |
| CKA_VALUE_LEN <sup>2,3</sup> | CK ULONG         | Length in bytes of key value                                               |

14027 The following is a sample template for creating a Poly1305 secret key object:

```

14028     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
14029     CK_KEY_TYPE keyType = CKK_POLY1305;
14030     CK_UTF8CHAR label[] = "A Poly1305 secret key object";
14031     CK_BYTE value[32] = {...};
14032     CK_BBOOL true = CK_TRUE;
14033     CK_ATTRIBUTE template[] = {
14034         {CKA_CLASS, &class, sizeof(class)},
14035         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14036         {CKA_TOKEN, &true, sizeof(true)},
14037         {CKA_LABEL, label, sizeof(label)-1},
14038         {CKA_SIGN, &true, sizeof(true)},
14039         {CKA_VALUE, value, sizeof(value)}}
14040     };
```

14041

### 14042 6.60.3 Poly1305 mechanism

14043 Poly1305, denoted **CKM\_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key  
 14044 and arbitrary length input.

14045 It has no parameters.

14046 Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

14047 *Table 258, Poly1305: Key and Data Length*

| <b>Function</b> | <b>Key type</b> | <b>Data length</b> | <b>Signature Length</b> |
|-----------------|-----------------|--------------------|-------------------------|
| C_Sign          | Poly1305        | Any                | 128 bits                |
| C_Verify        | Poly1305        | Any                | 128 bits                |

### 14048 6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated 14049 Encryption / Decryption

14050 The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305  
 14051 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data  
 14052 (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

14053 *Table 259, Poly1305 Mechanisms vs. Functions*

| <b>Mechanism</b>      | <b>Functions</b>             |                          |                                |               |                           |                          |               |
|-----------------------|------------------------------|--------------------------|--------------------------------|---------------|---------------------------|--------------------------|---------------|
|                       | <b>Encrypt &amp; Decrypt</b> | <b>Sign &amp; Verify</b> | <b>SR &amp; VR<sup>1</sup></b> | <b>Digest</b> | <b>Gen. Key/ Key Pair</b> | <b>Wrap &amp; Unwrap</b> | <b>Derive</b> |
| CKM_CHACHA20_POLY1305 | ✓                            |                          |                                |               |                           |                          |               |
| CKM_SALSA20_POLY1305  | ✓                            |                          |                                |               |                           |                          |               |

## 14054 6.61.1 Definitions

14055 Mechanisms:

14056 CKM\_CHACHA20\_POLY1305

14057 CKM\_SALSA20\_POLY1305

## 14058 6.61.2 Usage

14059 Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].  
14060 To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305  
14061 and Salsa20/Poly1305 both use CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS for Encrypt, Decrypt  
14062 and CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS for MessageEncrypt, and MessageDecrypt.

14063 Encrypt:

- 14064 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20  
14065 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14066 • Set the Nonce data *pNonce* in the parameter block.
- 14067 • Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if  
14068 *uAADLen* is 0.
- 14069 • Call `C_EncryptInit()` for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305**  
14070 mechanism with parameters and key *K*.
- 14071 • Call `C_Encrypt()`, or `C_EncryptUpdate()`\*<sup>10</sup> `C_EncryptFinal()`, for the plaintext obtaining ciphertext  
14072 and authentication tag output.

14073 Decrypt:

- 14074 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20  
14075 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14076 • Set the Nonce data *pNonce* in the parameter block.
- 14077 • Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if  
14078 *uAADLen* is 0.
- 14079 • Call `C_DecryptInit()` for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305**  
14080 mechanism with parameters and key *K*.
- 14081 • Call `C_Decrypt()`, or `C_DecryptUpdate()`\*<sup>1</sup> `C_DecryptFinal()`, for the ciphertext, including the  
14082 appended tag, obtaining plaintext output. Note: since **CKM\_CHACHA20\_POLY1305** and  
14083 **CKM\_SALSA20\_POLY1305** are AEAD ciphers, no data should be returned until `C_Decrypt()` or  
14084 `C_DecryptFinal()`.

14085 MessageEncrypt::

- 14086 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20  
14087 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14088 • Set the Nonce data *pNonce* in the parameter block.
- 14089 • Set *pTag* to hold the tag data returned from `C_EncryptMessage()` or the final  
14090 `C_EncryptMessageNext()`.
- 14091 • Call `C_MessageEncryptInit()` for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305**  
14092 mechanism with key *K*.

---

10 “\*” indicates 0 or more calls may be made as required

- 14093     • Call C\_EncryptMessage(), or C\_EncryptMessageBegin followed by C\_EncryptMessageNext()<sup>\*11</sup>.  
 14094       The mechanism parameter is passed to all three of these functions.
- 14095     • Call C\_MessageEncryptFinal() to close the message decryption.

14096 MessageDecrypt:

- 14097     • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of ChaCha20  
 14098       will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14099     • Set the Nonce data *pNonce* in the parameter block.
- 14100     • Set the tag data *pTag* in the parameter block before C\_DecryptMessage or the final  
 14101       C\_DecryptMessageNext()
- 14102     • Call C\_MessageDecryptInit() for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305**  
 14103       mechanism with key *K*.
- 14104     • Call C\_DecryptMessage(), or C\_DecryptMessageBegin followed by C\_DecryptMessageNext()<sup>\*12</sup>.  
 14105       The mechanism parameter is passed to all three of these functions.
- 14106     • Call C\_MessageDecryptFinal() to close the message decryption

14107

14108 *ulNonceLen* is the length of the nonce in bits.

14109 In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in  
 14110 the *pTag* field of CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS. In MessageDecrypt the tag is  
 14111 provided by the *pTag* field of CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS. The application  
 14112 must provide 16 bytes of space for the tag.

14113 The key type for *K* must be compatible with **CKM\_CHACHA20** or **CKM\_SALSA20** respectively and the  
 14114 C\_EncryptInit/C\_DecryptInit calls shall behave, with respect to *K*, as if they were called directly with  
 14115 **CKM\_CHACHA20** or **CKM\_SALSA20**, *K* and NULL parameters.

14116 Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose  
 14117 the block counter, as the AEAD construction is based on a message metaphor in which random access is  
 14118 not needed.

### 14119 **6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters**

- 14120     ◆ **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS;**  
 14121       **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS\_PTR**

14122 **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS** is a structure that provides the parameters to the  
 14123 **CKM\_CHACHA20\_POLY1305** and **CKM\_SALSA20\_POLY1305** mechanisms. It is defined as follows:

```
14124     typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
14125       CK_BYTE_PTR pNonce;
14126       CK ULONG ulNonceLen;
14127       CK_BYTE_PTR pAAD;
14128       CK ULONG ulAADLen;
14129     } CK_SALSA20_CHACHA20_POLY1305_PARAMS;
```

14130 The fields of the structure have the following meanings:

11 “\*” indicates 0 or more calls may be made as required

12 “\*” indicates 0 or more calls may be made as required

14131                    *pNonce*        *nonce (This should be never re-used with the same key.)*

14132                    *ulNonceLen*     *length of nonce in bits (is 64 for original, 96 for IETF (only for chacha20) and 192 for xchacha20/xsalsa20 variant)*

14134                    *pAAD*         *pointer to additional authentication data. This data is authenticated but not encrypted.*

14135

14136                    *ulAADLen*     *length of pAAD in bytes.*

14137   **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS\_PTR** is a pointer to a  
 14138   **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS**.

14139   ◆ **CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS;**  
 14140                    **CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS\_PTR**

14141   **CK\_CHACHA20POLY1305\_PARAMS** is a structure that provides the parameters to the CKM\_  
 14142   CHACHA20\_POLY1305 mechanism. It is defined as follows:

```
14143   typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
14144       CK_BYTE_PTR    pNonce;
14145       CK ULONG      ulNonceLen;
14146       CK_BYTE_PTR    pTag;
14147 } CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
```

14148   The fields of the structure have the following meanings:

14149                    *pNonce*        *pointer to nonce*

14150                    *ulNonceLen*     *length of nonce in bits. The length of the influences which variant of the ChaCha20 will be used (64 original, 96 IETF(only for ChaCha20), 192 XChaCha20/XSalsa20)*

14153                    *pTag*         *location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.*

14154

14155   **CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS\_PTR** is a pointer to a  
 14156   **CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS**.

## 6.62 HKDF Mechanisms

14158   Details for HKDF key derivation mechanisms can be found in [RFC 5869].

14159

14160   Table 260, HKDF Mechanisms vs. Functions

| Mechanism        | Functions         |               |                      |        |                    |               |        |
|------------------|-------------------|---------------|----------------------|--------|--------------------|---------------|--------|
|                  | Encrypt & Decrypt | Sign & Verify | SR & VR <sup>1</sup> | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_HKDF_DERIVE  |                   |               |                      |        |                    |               | ✓      |
| CKM_HKDF_DATA    |                   |               |                      |        |                    |               | ✓      |
| CKM_HKDF_KEY_GEN |                   |               |                      |        | ✓                  |               |        |

## 14161 6.62.1 Definitions

14162 Mechanisms:

14163 CKM\_HKDF\_DERIVE  
14164 CKM\_HKDF\_DATA  
14165 CKM\_HKDF\_KEY\_GEN

14166

14167 Key Types:

14168 CKK\_HKDF

## 14169 6.62.2 HKDF mechanism parameters

### 14170 ◆ CK\_HKDF\_PARAMS; CK\_HKDF\_PARAMS\_PTR

14171 CK\_HKDF\_PARAMS is a structure that provides the parameters to the CKM\_HKDF\_DERIVE and  
14172 CKM\_HKDF\_DATA mechanisms. It is defined as follows:

```
14173     typedef struct CK_HKDF_PARAMS {  
14174         CK_BBOOL bExtract;  
14175         CK_BBOOL bExpand;  
14176         CK_MECHANISM_TYPE prfHashMechanism;  
14177         CK ULONG ulSaltType;  
14178         CK_BYTE_PTR pSalt;  
14179         CK ULONG ulSaltLen;  
14180         CK_OBJECT_HANDLE hSaltKey;  
14181         CK_BYTE_PTR pInfo;  
14182         CK ULONG ulInfoLen;  
14183     } CK_HKDF_PARAMS;
```

14184

14185 The fields of the structure have the following meanings:

|       |                  |                                                                                                                                                                                                                                                                |
|-------|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 14186 | bExtract         | execute the extract portion of HKDF.                                                                                                                                                                                                                           |
| 14187 | bExpand          | execute the expand portion of HKDF.                                                                                                                                                                                                                            |
| 14188 | prfHashMechanism | base hash used for the HMAC in the underlying HKDF operation.                                                                                                                                                                                                  |
| 14189 | ulSaltType       | specifies how the salt for the extract portion of the KDF is supplied.<br>CKF_HKDF_SALT_NULL no salt is supplied.<br>CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with length ulSaltLen.<br>CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey. |
| 14190 | pSalt            | pointer to the salt.                                                                                                                                                                                                                                           |
| 14191 | ulSaltLen        | length of the salt pointed to in pSalt.                                                                                                                                                                                                                        |
| 14192 | hSaltKey         | object handle to the salt key.                                                                                                                                                                                                                                 |
| 14193 | pInfo            | info string for the expand stage.                                                                                                                                                                                                                              |
| 14194 | ulInfoLen        | length of the info string for the expand stage.                                                                                                                                                                                                                |

14195

14200 CK\_HKDF\_PARAMS\_PTR is a pointer to a CK\_HKDF\_PARAMS.

### 6.62.3 HKDF derive

HKDF derivation implements the HKDF as specified in [RFC 5869]. The two booleans bExtract and bExpand control whether the extract section of the HKDF or the expand section of the HKDF is in use. It has a parameter, a **CK\_HKDF\_PARAMS** structure, which allows for the passing of the salt and or the expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract or expand portions of the HKDF is to be used. This structure is defined in Section 6.62.2.

The input key must be of type **CKK\_HKDF** or **CKK\_GENERIC\_SECRET** and the length must be the size of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has the same size as the underlying hash function, and which may be supplied as an input key. In this case *bExtract* should be true and non-null salt should be supplied.

Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then expanded. The salt is used in the extraction stage. If *bExtract* is set to true and no salt is given, a 'zero' salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified by the RFC. If *bExpand* is set to true, **CKA\_VALUE\_LEN** should be set to the desired key length. If it is false **CKA\_VALUE\_LEN** may be set to the length of the hash, but that is not necessary as the mechanism will supply this value. The salt should be ignored if *bExtract* is false. The *pInfo* should be ignored if *bExpand* is set to false.

The mechanism also contributes the **CKA\_CLASS**, and **CKA\_VALUE** attributes to the new key. Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

### 6.62.4 HKDF Data

HKDF Data derive mechanism, denoted **CKM\_HKDF\_DATA**, is identical to HKDF Derive except the output is a **CKO\_DATA** object whose value is the result to the derive operation. Some tokens may restrict what data may be successfully derived based on the *pInfo* portion of the **CK\_HKDF\_PARAMS**. Tokens may reject requests based on the *pInfo* values. Allowed *pInfo* values are specified in the profile document and applications could then query the appropriate profile before depending on the mechanism.

### 6.62.5 HKDF Key gen

HKDF key gen, denoted **CKM\_HKDF\_KEY\_GEN** generates a new random HKDF key. **CKA\_VALUE\_LEN** must be set in the template.

## 6.63 NULL Mechanism

**CKM\_NULL** is a mechanism used to implement the trivial pass-through function.

14247 *Table 261, CKM\_NULL Mechanisms vs. Functions*

| <b>Mechanism</b> | <b>Functions</b>             |                          |                                |               |                           |                          |               |
|------------------|------------------------------|--------------------------|--------------------------------|---------------|---------------------------|--------------------------|---------------|
|                  | <b>Encrypt &amp; Decrypt</b> | <b>Sign &amp; Verify</b> | <b>SR &amp; VR<sup>1</sup></b> | <b>Digest</b> | <b>Gen. Key/ Key Pair</b> | <b>Wrap &amp; Unwrap</b> | <b>Derive</b> |
| CKM_NULL         | ✓                            | ✓                        | ✓                              | ✓             |                           | ✓                        | ✓             |

<sup>1</sup>SR = SignRecover, VR = VerifyRecover

14248

### 14249 **6.63.1 Definitions**

14250 Mechanisms:

14251 CKM\_NULL

### 14252 **6.63.2 CKM\_NULL mechanism parameters**

14253 CKM\_NULL does not have a parameter.

14254

14255 When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

14256 When used for signing, the input data is copied to the signature. When used for signature verification, it compares the input data and the signature, and returns CKR\_OK (indicating that both are identical) or CKR\_SIGNATURE\_INVALID.

14259 When used for digesting data, the input data is copied to the message digest.

14260 When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be created.

14263 When used for deriving a key, the derived key has the same value as the base key.

14264

### 14265 **6.64 IKE Mechanisms**

14266

14267 *Table 262, IKE Mechanisms vs. Functions*

| <b>Mechanism</b>         | <b>Functions</b>             |                          |                                |               |                           |                          |               |
|--------------------------|------------------------------|--------------------------|--------------------------------|---------------|---------------------------|--------------------------|---------------|
|                          | <b>Encrypt &amp; Decrypt</b> | <b>Sign &amp; Verify</b> | <b>SR &amp; VR<sup>1</sup></b> | <b>Digest</b> | <b>Gen. Key/ Key Pair</b> | <b>Wrap &amp; Unwrap</b> | <b>Derive</b> |
| CKM_IKE2_PRF_PLUS_DERIVE |                              |                          |                                |               |                           |                          | ✓             |
| CKM_IKE_PRF_DERIVE       |                              |                          |                                |               |                           |                          | ✓             |
| CKM_IKE1_PRF_DERIVE      |                              |                          |                                |               |                           |                          | ✓             |
| CKM_IKE1_EXTENDED_DERIVE |                              |                          |                                |               |                           |                          | ✓             |

14268

### 14269 **6.64.1 Definitions**

14270 Mechanisms:

```
14271      CKM_IKE2_PRF_PLUS_DERIVE  
14272      CKM_IKE_PRF_DERIVE  
14273      CKM_IKE1_PRF_DERIVE  
14274      CKM_IKE1_EXTENDED_DERIVE  
14275
```

## 14276 6.64.2 IKE mechanism parameters

- 14277 ◆ **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS;**
- 14278 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR**

14279 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
14280 **CKM\_IKE2\_PRF\_PLUS\_DERIVE** mechanism. It is defined as follows:

```
14281     typedef struct CK_IKE2_PRF_PLUS_DERIVE_PARAMS {  
14282         CK_MECHANISM_TYPE prfMechanism;  
14283         CK_BBOOL bHasSeedKey;  
14284         CK_OBJECT_HANDLE hSeedKey;  
14285         CK_BYTE_PTR pSeedData;  
14286         CK ULONG ulSeedDataLen;  
14287     } CK_IKE2_PRF_PLUS_DERIVE_PARAMS;
```

14288 The fields of the structure have the following meanings:

|       |               |                                                                                   |
|-------|---------------|-----------------------------------------------------------------------------------|
| 14290 | prfMechanism  | underlying MAC mechanism used to generate the prf                                 |
| 14291 | bHasSeedKey   | hSeed key is present                                                              |
| 14292 | hSeedKey      | optional seed from key                                                            |
| 14293 | pSeedData     | optional seed from data                                                           |
| 14294 | ulSeedDataLen | length of optional seed data. If no seed data is present this value is<br>14295 0 |

14296 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR** is a pointer to a  
14297 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS**.

14298

- 14299 ◆ **CK\_IKE\_PRF\_DERIVE\_PARAMS; CK\_IKE\_PRF\_DERIVE\_PARAMS\_PTR**

14300 **CK\_IKE\_PRF\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
14301 **CKM\_IKE\_PRF\_DERIVE** mechanism. It is defined as follows:

```
14302  
14303     typedef struct CK_IKE_PRF_DERIVE_PARAMS {  
14304         CK_MECHANISM_TYPE prfMechanism;  
14305         CK_BBOOL bDataAsKey;  
14306         CK_BBOOL bRekey;  
14307         CK_BYTE_PTR pNi;  
14308         CK ULONG ulNiLen;  
14309         CK_BYTE_PTR pNr;  
14310         CK ULONG ulNrLen;  
14311         CK_OBJECT_HANDLE hNewKey;  
14312     } CK_IKE_PRF_DERIVE_PARAMS;
```

14313

14314 The fields of the structure have the following meanings:

|       |              |                                                           |
|-------|--------------|-----------------------------------------------------------|
| 14315 | prfMechanism | underlying MAC mechanism used to generate the prf         |
| 14316 | bDataAsKey   | Ni  Nr is used as the key for the prf rather than baseKey |
| 14317 | bRekey       | rekey operation. hNewKey must be present                  |
| 14318 | pNi          | Ni value                                                  |
| 14319 | ulNiLen      | length of Ni                                              |
| 14320 | pNr          | Nr value                                                  |
| 14321 | ulNrLen      | length of Nr                                              |
| 14322 | hNewKey      | New key value to drive the rekey.                         |

14323 **CK\_IKE\_PRF\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE\_PRF\_DERIVE\_PARAMS**.

14324

14325 ◆ **CK\_IKE1\_PRF\_DERIVE\_PARAMS; CK\_IKE1\_PRF\_DERIVE\_PARAMS\_PTR**

14326 **CK\_IKE1\_PRF\_DERIVE\_PARAMS** is a structure that provides the parameters to the

14327 **CKM\_IKE1\_PRF\_DERIVE** mechanism. It is defined as follows:

```
14328     typedef struct CK_IKE1_PRF_DERIVE_PARAMS {
14329         CK_MECHANISM_TYPE    prfMechanism;
14330         CK_BBOOL              bHasPrevKey;
14331         CK_OBJECT_HANDLE      hKeygxy;
14332         CK_OBJECT_HANDLE      hPrevKey;
14333         CK_BYTE_PTR           pCKYi;
14334         CK ULONG               ulCKYiLen;
14335         CK_BYTE_PTR           pCKYr;
14336         CK ULONG               ulCKYrLen;
14337         CK_BYTE               keyNumber;
14338     } CK_IKE1_PRF_DERIVE_PARAMS;
```

14339

14340 The fields of the structure have the following meanings:

|       |              |                                                   |
|-------|--------------|---------------------------------------------------|
| 14341 | prfMechanism | underlying MAC mechanism used to generate the prf |
| 14342 | bHasPrevkey  | hPrevKey is present                               |
| 14343 | hKeygxy      | handle to the exchanged g^xy key                  |
| 14344 | hPrevKey     | handle to the previously derived key              |
| 14345 | pCKYi        | CKYi value                                        |
| 14346 | ulCKYiLen    | length of CKYi                                    |
| 14347 | pCKYr        | CKYr value                                        |
| 14348 | ulCKYrLen    | length of CKYr                                    |
| 14349 | keyNumber    | unique number for this key derivation             |

14350 **CK\_IKE1\_PRF\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE1\_PRF\_DERIVE\_PARAMS**.

14351

14352 ◆ **CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS;**  
14353 **CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS\_PTR**

14354 **CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS** is a structure that provides the parameters to the  
14355 **CKM\_IKE1\_EXTENDED\_DERIVE** mechanism. It is defined as follows:

```
14356
14357     typedef struct CK_IKE1_EXTENDED_DERIVE_PARAMS {
14358         CK_MECHANISM_TYPE prfMechanism;
14359         CK_BBOOL bHasKeygxy;
14360         CK_OBJECT_HANDLE hKeygxy;
14361         CK_BYTE_PTR pExtraData;
14362         CK ULONG ulExtraDataLen;
14363     } CK_IKE1_EXTENDED_DERIVE_PARAMS;
```

14364 The fields of the structure have the following meanings:

|       |                |                                                                          |
|-------|----------------|--------------------------------------------------------------------------|
| 14365 | prfMechanism   | underlying MAC mechanism used to generate the prf                        |
| 14366 | bHasKeygxy     | hKeygxy key is present                                                   |
| 14367 | hKeygxy        | optional key $g^x y$                                                     |
| 14368 | pExtraData     | optional extra data                                                      |
| 14369 | ulExtraDataLen | length of optional extra data. If no extra data is present this value is |
| 14370 |                | 0                                                                        |

14371 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR** is a pointer to a  
14372 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS**.

14373

### 14374 6.64.3 IKE PRF DERIVE

14375 The IKE PRF Derive mechanism denoted **CKM\_IKE\_PRF\_DERIVE** is used in IPSEC both IKEv1 and  
14376 IKEv2 to generate an initial key that is used to generate additional keys. It takes a  
14377 **CK\_IKE\_PRF\_DERIVE\_PARAMS** as a mechanism parameter. *baseKey* is the base key passed into  
14378 **C\_DeriveKey**. *baseKey* must be of type **CKK\_GENERIC\_SECRET** if *bDataAsKey* is TRUE and the key  
14379 type of the underlying prf if *bDataAsKey* is FALSE. *hNewKey* must be of type **CKK\_GENERIC\_SECRET**.  
14380 Depending on the parameter settings, it generates keys with a **CKA\_VALUE** of:

- 14381 1. prf(pNijpNr, baseKey); (*bDataAsKey*=TRUE, *bRekey*=FALSE)
- 14382 2. prf(baseKey, pNijpNr); (*bDataAsKey*=FALSE, *bRekey*=FALSE)
- 14383 3. prf(baseKey, ValueOf(hNewKey) | pNi | pNr); (*bDataAsKey*=FALSE, *bRekey*=TRUE)

14384 The resulting output key is always the length of the underlying prf. The combination of  
14385 *bDataAsKey*=TRUE and *bRekey*=TRUE is not allowed. If both are set, **CKR\_ARGUMENTS\_BAD** is  
14386 returned.

14387 Case 1 is used in

- 14388 a. ikev2 (RFC 5996) *baseKey* is called  $g^{\text{ir}}$ , the output is called SKEYSEED
- 14389 b. ikev1 (RFC 2409) *baseKey* is called  $g^{\text{ir}}$ , the output is called SKEYID

14390 Case 2 is used in ikev1 (RFC 2409) inkey is called pre-shared-key, output is called SKEYID

14391 Case 3 is used in ikev2 (RFC 5996) rekey case, *baseKey* is SK\_d, *hNewKey* is  $g^{\text{ir}}$  (new), the output is  
14392 called SKEYSEED. The derived key will have a length of the length of the underlying prf. If  
14393 **CKA\_VALUE\_LEN** is specified, it must equal the underlying prf or **CKR\_KEY\_SIZE\_RANGE** is returned.  
14394 If **CKA\_KEY\_TYPE** is not specified in the template, it will be the underlying key type of the prf.

14396

#### 14397 6.64.4 IKEv1 PRF DERIVE

14398 The IKEv1 PRF Derive mechanism denoted **CKM\_IKE1\_PRF\_DERIVE** is used in IPSEC IKEv1 to  
14399 generate various additional keys from the initial SKEYID. It takes a **CK\_IKE1\_PRF\_DERIVE\_PARAMS**  
14400 as a mechanism parameter. SKEYID is the base key passed into **C\_DeriveKey**.

14401

14402 This mechanism derives a key with **CKA\_VALUE** set to either:

14403     prf(baseKey, ValueOf(hKeygxy) || pCKYi || pCKYr || key\_number)

14404 or

14405     prf(baseKey, ValueOf(hPrevKey) || ValueOf(hKeygxy) || pCKYi || pCKYr || key\_number)

14406 depending on the state of *bHasPrevKey*.

14407 The key type of *baseKey* must be the key type of the prf, and the key type of *hKeygxy* must be  
14408 **CKK\_GENERIC\_SECRET**. The key type of *hPrevKey* can be any key type.

14409

14410 This is defined in RFC 2409. For each of the following keys.

14411     *baseKey* is SKEYID, *hKeygxy* is  $g^x y$

14412     for *outKey* = SKEYID\_d, *bHasPrevKey* = false, *key\_number* = 0

14413     for *outKey* = SKEYID\_a, *hPrevKey* = SKEYID\_d, *key\_number* = 1

14414     for *outKey* = SKEYID\_e, *hPrevKey* = SKEYID\_a, *key\_number* = 2

14415 If **CKA\_VALUE\_LEN** is not specified, the resulting key will be the length of the prf. If **CKA\_VALUE\_LEN**  
14416 is greater than the prf, **CKR\_KEY\_SIZE\_RANGE** is returned. If it is less the key is truncated taking the  
14417 left most bytes. The value **CKA\_KEY\_TYPE** must be specified in the template or  
14418 **CKR\_TEMPLATE\_INCOMPLETE** is returned.

14419

#### 14420 6.64.5 IKEv2 PRF PLUS DERIVE

14421 The IKEv2 PRF PLUS Derive mechanism denoted **CKM\_IKE2\_PRF\_PLUS\_DERIVE** is used in IPSEC  
14422 IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a  
14423 **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS** as a mechanism parameter. SKEYSEED is the base key  
14424 passed into **C\_DeriveKey**. The key type of *baseKey* must be the key type of the underlying prf. This  
14425 mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient  
14426 bytes to cover all additional keys. The application will then use **CKM\_EXTRACT\_KEY\_FROM\_KEY**  
14427 several times to pull out the various keys. **CKA\_VALUE\_LEN** must be set in the template and its value  
14428 must not be bigger than 255 times the size of the prf function output or **CKR\_KEY\_SIZE\_RANGE** will be  
14429 returned. If **CKA\_KEY\_TYPE** is not specified, the output key type will be **CKK\_GENERIC\_SECRET**.

14430

14431 This mechanism derives a key with a **CKA\_VALUE** of (from RFC 5996):

14432

14433  $\text{prfplus} = T_1 | T_2 | T_3 | T_4 | \dots | T_n$

14434 where:

14435      $T_1 = \text{prf}(K, S | 0x01)$

14436      $T_2 = \text{prf}(K, T_1 | S | 0x02)$

14437      $T_3 = \text{prf}(K, T_2 | S | 0x03)$

14438      $T_4 = \text{prf}(K, T_3 | S | 0x04)$

14439

14440  $T_n = \text{prf}(K, T(n-1) | n)$   
14441  $K = \text{baseKey}, S = \text{valueOf}(h\text{SeedKey}) | p\text{SeedData}$   
14442

## 14443 6.64.6 IKEv1 Extended Derive

14444 The IKE Extended Derive mechanism denoted **CKM\_IKE1\_EXTENDED\_DERIVE** is used in IPSEC  
14445 IKEv1 to derive longer keys than **CKM\_IKE1\_EXTENDED\_DERIVE** can from the initial SKEYID. It is  
14446 used to support RFC 2409 appendix B and RFC 2409 section 5.5 (Quick Mode). It takes a  
14447 **CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS** as a mechanism parameter. SKEYID is the base key passed  
14448 into **C\_DeriveKey**. **CKA\_VALUE\_LEN** must be set in the template and its value must not be bigger than  
14449 255 times the size of the prf function output or **CKR\_KEY\_SIZE\_RANGE** will be returned. If  
14450 **CKA\_KEY\_TYPE** is not specified, the output key type will be **CKK\_GENERIC\_SECRET**. The key type of  
14451 SKEYID must be the key type of the prf, and the key type of *hKeygxy* (if present) must be  
14452 **CKK\_GENERIC\_SECRET**.

14453  
14454 This mechanism derives a key with **CKA\_VALUE** (from RFC 2409 appendix B and section 5.5):  
14455  $K_a = K_1 | K_2 | K_3 | K_4 | \dots | K_n$   
14456 where:  
14457  $K_1 = \text{prf}(K, \text{valueOf}(h\text{Keygxy}) | p\text{ExtraData})$  or  $\text{prf}(K, 0x00)$  if *bHashKeygxy* is FALSE and *ulExtraData*  
14458 is 0  
14459  $K_2 = \text{prf}(K, K_1 | \text{valueOf}(h\text{Keygxy}) | p\text{ExtraData})$   
14460  $K_3 = \text{prf}(K, K_2 | \text{valueOf}(h\text{Keygxy}) | p\text{ExtraData})$   
14461  $K_4 = \text{prf}(K, K_3 | \text{valueOf}(h\text{Keygxy}) | p\text{ExtraData})$   
14462 .  
14463  $K_n = \text{prf}(K, K_{(n-1)} | \text{valueOf}(h\text{Keygxy}) | p\text{ExtraData})$   
14464  $K = \text{baseKey}$   
14465  
14466 If **CKA\_VALUE\_LEN** is less than or equal to the prf length and *bHasKeygxy* is CK\_FALSE, then the new  
14467 key is simply the base key truncated to **CKA\_VALUE\_LEN** (specified in RFC 2409 appendix B).  
14468 Otherwise the prf is executed and the derived keys value is **CKA\_VALUE\_LEN** bytes of the resulting prf.

## 14469 6.65 HSS

14470 HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm  
14471 defined in [RFC 8554] and [NIST 802-208].

14472

14473 *Table 263, HSS Mechanisms vs. Functions*

| Mechanism            | Functions         |                |         |        |                    |               |        |
|----------------------|-------------------|----------------|---------|--------|--------------------|---------------|--------|
|                      | Encrypt & Decrypt | Sign & Verify  | SR & VR | Digest | Gen. Key/ Key Pair | Wrap & Unwrap | Derive |
| CKM_HSS_KEY_PAIR_GEN |                   |                |         |        | ✓                  |               |        |
| CKM_HSS              |                   | ✓ <sup>1</sup> |         |        |                    |               |        |

14474 1 Single-part operations only

14475

## 14476 6.65.1 Definitions

14477 This section defines the key type **CKK\_HSS** for type **CK\_KEY\_TYPE** as used in the **CKA\_KEY\_TYPE**  
14478 attribute of key objects and domain parameter objects.

14479 Mechanisms:

14480 CKM\_HSS\_KEY\_PAIR\_GEN

14481 CKM\_HSS

## 14482 6.65.2 HSS public key objects

14483 HSS public key objects (object class **CKO\_PUBLIC\_KEY**, key type **CKK\_HSS**) hold HSS public keys.

14484 The following table defines the HSS public key object attributes, in addition to the common attributes  
14485 defined for this object class:

14486 Table 264, *HSS Public Key Object Attributes*

| Attribute                         | Data Type  | Meaning                                                                                               |
|-----------------------------------|------------|-------------------------------------------------------------------------------------------------------|
| CKA_HSS_LEVELS <sup>2,4</sup>     | CK ULONG   | The number of levels in the HSS scheme.                                                               |
| CKA_HSS_LMS_TYPE <sup>2,4</sup>   | CK ULONG   | The encoding for the Merkle tree heights of the top level LMS tree in the hierarchy.                  |
| CKA_HSS_LMOTS_TYPE <sup>2,4</sup> | CK ULONG   | The encoding for the Winternitz parameter of the one-time-signature scheme of the top level LMS tree. |
| CKA_VALUE <sup>1,4</sup>          | Byte array | XDR-encoded public key as defined in [RFC8554].                                                       |

14487 - Refer to Table 11 for footnotes

14488

14489 The following is a sample template for creating an HSS public key object:

14490

```
14491     CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
14492     CK_KEY_TYPE keyType = CKK_HSS;
14493     CK_UTF8CHAR label[] = "An HSS public key object";
14494     CK_BYTE value[] = {...};
14495     CK_BBOOL true = CK_TRUE;
14496     CK_BBOOL false = CK_FALSE;
14497
14498     CK_ATTRIBUTE template[] = {
14499         {CKA_CLASS, &keyClass, sizeof(keyClass)},
14500         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14501         {CKA_TOKEN, &false, sizeof(false)},
14502         {CKA_LABEL, label, sizeof(label)-1},
14503         {CKA_VALUE, value, sizeof(value)},
14504         {CKA_VERIFY, &true, sizeof(true)}  
14505     };
```

### 6.65.3 HSS private key objects

HSS private key objects (object class **CKO\_PRIVATE\_KEY**, key type **CKK\_HSS**) hold HSS private keys.

The following table defines the HSS private key object attributes, in addition to the common attributes defined for this object class:

*Table 265, HSS Private Key Object Attributes*

| Attribute                             | Data Type    | Meaning                                                                                                                                                                                                                                                                                                               |
|---------------------------------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CKA_HSS_LEVELS <sup>1,3</sup>         | CK ULONG     | The number of levels in the HSS scheme.                                                                                                                                                                                                                                                                               |
| CKA_HSS_LMS_TYPES <sup>1,3</sup>      | CK ULONG_PTR | A list of encodings for the Merkle tree heights of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK ULONG. This number must match the CKA_HSS_LEVELS attribute value.                                   |
| CKA_HSS_LMOTS_TYPES <sup>1,3</sup>    | CK ULONG_PTR | A list of encodings for the Winternitz parameter of the one-time-signature scheme of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK ULONG. This number must match the CKA_HSS_LEVELS attribute value. |
| CKA_VALUE <sup>1,4,6,7</sup>          | Byte array   | Vendor defined, must include state information.<br>Note that exporting this value is dangerous as it would allow key reuse.                                                                                                                                                                                           |
| CKA_HSS_KEYS_REMAINING <sup>2,4</sup> | CK ULONG     | The minimum of the following two values:<br>1) The number of one-time private keys remaining; 2) 2^32-1                                                                                                                                                                                                               |

- Refer to Table 11 for footnotes

The encodings for CKA\_HSS\_LMOTS\_TYPES and CKA\_HSS\_LMS\_TYPES are defined in [RFC 8554] and [NIST 802-208].

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

```
CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;  
CK_KEY_TYPE keyType = CKK_HSS;  
CK_UTF8CHAR label[] = "An HSS private key object";  
CK ULONG hssLevels = 123;  
CK ULONG lmsTypes[] = {123, ...};
```

```

14524     CK ULONG lmotsTypes[] = {123,...};
14525     CK_BYTE value[] = {...};
14526     CK_BBOOL true = CK_TRUE;
14527     CK_BBOOL false = CK_FALSE;
14528     CK_ATTRIBUTE template[] = {
14529         {CKA_CLASS, &keyClass, sizeof(keyClass)},
14530         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14531         {CKA_TOKEN, &true, sizeof(true)},
14532         {CKA_LABEL, label, sizeof(label)-1},
14533         {CKA_SENSITIVE, &true, sizeof(true)},
14534         {CKA_EXTRACTABLE, &false, sizeof(true)},
14535         {CKA_HSS_LEVELS, &hssLevels, sizeof(hssLevels)},
14536         {CKA_HSS_LMS_TYPES, lmsTypes, sizeof(lmsTypes)},
14537         {CKA_HSS_LMOTS_TYPES, lmotsTypes, sizeof(lmotsTypes)},
14538         {CKA_VALUE, value, sizeof(value)},
14539         {CKA_SIGN, &true, sizeof(true)}}
14540     };
14541
14542 CKA_SENSITIVE MUST be true, CKA_EXTRACTABLE MUST be false, and CKA_COPYABLE MUST
14543 be false for this key.

```

#### 14544 6.65.4 HSS key pair generation

14545 The HSS key pair generation mechanism, denoted **CKM\_HSS\_KEY\_PAIR\_GEN**, is a key pair generation  
14546 mechanism for HSS.

14547 This mechanism does not have a parameter.

14548 The mechanism generates HSS public/private key pairs for the scheme specified by the  
14549 **CKA\_HSS\_LEVELS**, **CKA\_HSS\_LMS\_TYPES**, and **CKA\_HSS\_LMOTS\_TYPES** attributes of the  
14550 template for the private key.

14551 The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_HSS\_LEVELS**,  
14552 **CKA\_HSS\_LMS\_TYPE**, **CKA\_HSS\_LMOTS\_TYPE**, and **CKA\_VALUE** attributes to the new public key  
14553 and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE**, and **CKA\_HSS\_KEYS\_REMAINING** attributes  
14554 to the new private key.

14555 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
14556 are not used and must be set to 0.

#### 14557 6.65.5 HSS without hashing

14558 The HSS without hashing mechanism, denoted **CKM\_HSS**, is a mechanism for single-part signatures and  
14559 verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value,  
14560 which may be of any length; it does not compute the hash value.)

14561 This mechanism does not have a parameter.

14562 For the purposes of these mechanisms, an HSS signature is a byte string with length depending on  
14563 **CKA\_HSS\_LEVELS**, **CKA\_HSS\_LMS\_TYPES**, **CKA\_HSS\_LMOTS\_TYPES** as described in the  
14564 following table.

14565 Table 266, *HSS without hashing: Key and Data Length*

| Function              | Key type        | Input length                 | Output length           |
|-----------------------|-----------------|------------------------------|-------------------------|
| C_Sign <sup>1</sup>   | HSS Private Key | any                          | 1296-74988 <sup>2</sup> |
| C_Verify <sup>1</sup> | HSS Public Key  | any, 1296-74988 <sup>2</sup> | N/A                     |

14566 <sup>1</sup> Single-part operations only.

14567 <sup>2</sup>  $4 + (\text{levels}-1)*56 + \text{levels}*(8 + (36+32*p) + h*32)$  where  $p$  has values (265, 133, 67, 34) for Imots type (W1, W2, W4, W8) and  $h$  is the number of levels in the LMS Merkle trees.

14569 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure  
14570 are not used and must be set to 0.

14571 If the number of signatures is exhausted, CKR\_KEY\_EXHAUSTED will be returned.

---

## 14572 7 PKCS #11 Implementation Conformance

### 14573 7.1 PKCS#11 Consumer Implementation Conformance

14574 An implementation is a conforming PKCS#11 Consumer if the implementation meets the conditions  
14575 specified in one or more consumer profiles specified in [PKCS11-Prof].

14576 A PKCS#11 consumer implementation SHALL be a conforming PKCS#11 Consumer.

14577 If a PKCS#11 consumer implementation claims support for a particular consumer profile, then the  
14578 implementation SHALL conform to all normative statements within the clauses specified for that profile  
14579 and for any subclauses to each of those clauses.

### 14580 7.2 PKCS#11 Provider Implementation Conformance

14581 An implementation is a conforming PKCS#11 Provider if the implementation meets the conditions  
14582 specified in one or more provider profiles specified in [PKCS11-Prof].

14583 A PKCS#11 provider implementation SHALL be a conforming PKCS#11 Provider.

14584

## Appendix A. Acknowledgments

14585

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

14587

### Participants:

| Salutation | First Name | Last Name      | Company                          |
|------------|------------|----------------|----------------------------------|
| Dr.        | Warren     | Armstrong      | QuintessenceLabs Pty Ltd.        |
|            | Anthony    | Berglas        | Cryptsoft Pty Ltd.               |
| Mr.        | Dieter     | Bong           | Utimaco IS GmbH                  |
| Mr.        | Roland     | Bramm          | PrimeKey Solutions AB            |
|            | Andrew     | Byrne          | Dell                             |
|            | Hamish     | Cameron        | nCipher                          |
|            | Kenli      | Chong          | QuintessenceLabs Pty Ltd.        |
| Mr.        | Justin     | Corlett        | Cryptsoft Pty Ltd.               |
|            | Xuelei     | Fan            | Oracle                           |
| Mr.        | Jan        | Friedel        | Oracle                           |
| Ms.        | Susan      | Gleeson        | Oracle                           |
| Mr.        | Thomas     | Hardjono       | M.I.T.                           |
| Mrs.       | Jane       | Harnad         | OASIS                            |
|            | David      | Horton         | Dell                             |
|            | Tim        | Hudson         | Cryptsoft Pty Ltd.               |
| Mr.        | Gershon    | Janssen        | Individual                       |
| Mr.        | Jakub      | Jelen          | Red Hat                          |
| Dr.        | Mark       | Joseph         | P6R, Inc                         |
| Mr.        | Paul       | King           | nCipher                          |
| Ms.        | Dina       | Kurktchi-Nimeh | Oracle                           |
|            | John       | Leiseboer      | QuintessenceLabs Pty Ltd.        |
| Mr.        | John       | Leser          | Oracle                           |
|            | Chris      | Malafis        | Red Hat                          |
| Dr.        | Michael    | Markowitz      | Information Security Corporation |
| Mr.        | Scott      | Marshall       | Cryptsoft Pty Ltd.               |

| Salutation | First Name | Last Name      | Company                          |
|------------|------------|----------------|----------------------------------|
| Mr.        | Chris      | Meyer          | Utimaco IS GmbH                  |
| Mr.        | Darren     | Moffat         | Oracle                           |
| Dr.        | Florian    | Poppa          | QuintessenceLabs Pty Ltd.        |
|            | Roland     | Reichenberg    | Utimaco IS GmbH                  |
| Mr.        | Robert     | Relyea         | Red Hat                          |
| Mr.        | Jonathan   | Schulze-Hewett | Information Security Corporation |
| Mr.        | Greg       | Scott          | Cryptsoft Pty Ltd.               |
| Mr.        | Martin     | Shannon        | QuintessenceLabs Pty Ltd.        |
| Mr.        | Oscar      | So             | Individual                       |
|            | Patrick    | Steuer         | IBM                              |
| Mr.        | Gerald     | Stueve         | Fornetix                         |
|            | Jim        | Susoy          | P6R, Inc                         |
| Mr.        | Sander     | Temme          | nCipher                          |
| Mr.        | Manish     | Upasani        | Utimaco IS GmbH                  |
| Mr.        | Charles    | White          | Fornetix                         |
| Ms.        | Magda      | Zdunkiewicz    | Cryptsoft Pty Ltd.               |

14588

---

## 14589 **Appendix B. Manifest constants**

14590 The definitions for manifest constants specified in this document can be found in the following normative  
14591 computer language definition files:

- 14592 • [pkcs11.h : https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11.h](https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11.h)
- 14593 • [pkcs11f.h : https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11f.h](https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11f.h)
- 14594 • [pkcs11t.h : https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11t.h](https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11t.h)

14596

## Appendix C. Revision History

14597

| Revision | Date             | Editor                      | Changes Made                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------|------------------|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WD01     | 02 December 2020 | Dieter Bong & Tony Cox      | <ul style="list-style-type: none"> <li>- Merged Base Specification &amp; Current Mechanisms forming new "PKCS#11 Specification v3.1"</li> <li>- Added CKA_DERIVE_TEMPLATE</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                       |
| WD02     | 04 December 2020 | Dieter Bong & Tony Cox      | <ul style="list-style-type: none"> <li>- Removed section 4.9.1 (covered in 6.1.3)</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| WD03     | 4 March 2021     | Dieter Bong & Tony Cox      | <ul style="list-style-type: none"> <li>- Section 6.3.8 2<sup>nd</sup> paragraph replace "Edwards" by "Montgomery"</li> <li>- Revised Note in § 5.2</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                              |
| WD04     | 1 June 2021      | Daniel Minder & Dieter Bong | <ul style="list-style-type: none"> <li>- Fixed several references and typos</li> <li>- Moved CKM_SHA224_RSA_PKCS and CKM_SHA224_RSA_PKCS_PSS from table 137 to table 32</li> <li>- Fixed the typo and added the wording wrt. CKA_VALUE_LEN in sections 6.64.2 and 6.64.6</li> <li>- Section 4.9 Private key objects: replaced "&lt;this version&gt;" by "PKCS #11 V2.40"</li> <li>- Section 6.65 updated to HSS proposal dd. 12 May 2021</li> <li>- Section 6.3: deprecation notice for CKM_ECDH_AES_KEY_WRAP</li> </ul>                                                   |
| WD05     | 15 July 2021     | Dieter Bong & Tony Cox      | <ul style="list-style-type: none"> <li>- Section 6.64: change the non-existing error CKR_KEY_RANGE_ERROR to CKR_KEY_SIZE_RANGE</li> <li>- Section 6.64.2: typo corrected: CK_IKE_PRF_PARAMS -&gt; CK_IKE_PRF_DERIVE_PARAMS</li> <li>- Section 6.64.5: improved wording for IKE v2 key derivation</li> <li>- Section 6.64 and 6.65: formatting updated</li> <li>- Section 6.65: removed timeout error code and description</li> <li>- Section 5.9.5: Reported by Mostafa ADILI: C_EncryptMessageNext should be C_MessageEncryptFinal in the function declaration</li> </ul> |
| WD06     | 14 October 2021  | Dieter Bong & Tony Cox      | <ul style="list-style-type: none"> <li>- Added clarifying text to 6.64.6</li> <li>- Clarified deprecation statement for CKM_ECDH_AES_KEY_WRAP</li> <li>- Updated [PKCS11-Prof] Reference</li> <li>- Clarified encodings in sections 6.3.5, 6.3.6, 6.3.7, 6.3.8, 6.3.16 &amp; 6.3.17</li> </ul>                                                                                                                                                                                                                                                                             |

| <b>Revision</b> | <b>Date</b>      | <b>Editor</b>          | <b>Changes Made</b>                                                                                                                                                                                                                                                                   |
|-----------------|------------------|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WD07            | 23 November 2021 | Dieter Bong & Tony Cox | <ul style="list-style-type: none"> <li>- Further clarification for CKM_ECDH_AES_KEY_WRAP deprecation notice</li> <li>- Clarified multiple EC key references in §6 (insertion of short Weierstrass descriptor)</li> <li>- Correction to description of CK_RSA_PKCS_MGF_TYPE</li> </ul> |
| WD08            | 9 December 2021  | Dieter Bong            | <ul style="list-style-type: none"> <li>- Clarified a few more EC key references in §6 (insertion of short Weierstrass descriptor and/or key type CKK_EC)</li> </ul>                                                                                                                   |
| WD09            | 14 December 2021 | Dieter Bong            | <ul style="list-style-type: none"> <li>- Updated deprecation notice for CKM_ECDH_AES_KEY_WRAP in section 6.3.20</li> </ul>                                                                                                                                                            |
| WD10            | 21 January 2022  | Dieter Bong            | Removed deprecation notice for CKM_ECDH_AES_KEY_WRAP in section 6.3.20 as per TC meeting 12-January-2022                                                                                                                                                                              |
| WD11            | 31 January 2022  | Dieter Bong            | Appendix B: include names of, and references to, computer language definition files                                                                                                                                                                                                   |

14598

---

## 14599 Appendix D. Notices

14600 Copyright © OASIS Open 2022. All Rights Reserved.

14601 All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual  
14602 Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](https://www.oasis-open.org/policies-guidelines/ipr/) may be found at the OASIS website:  
14603 [<https://www.oasis-open.org/policies-guidelines/ipr/>].

14604 This document and translations of it may be copied and furnished to others, and derivative works that  
14605 comment on or otherwise explain it or assist in its implementation may be prepared, copied, published,  
14606 and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice  
14607 and this section are included on all such copies and derivative works. However, this document itself may  
14608 not be modified in any way, including by removing the copyright notice or references to OASIS, except as  
14609 needed for the purpose of developing any document or deliverable produced by an OASIS Technical  
14610 Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must  
14611 be followed) or as required to translate it into languages other than English.

14612 The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors  
14613 or assigns.

14614 This document and the information contained herein is provided on an "AS IS" basis and OASIS  
14615 DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY  
14616 WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY  
14617 OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A  
14618 PARTICULAR PURPOSE. OASIS AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT,  
14619 INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS  
14620 DOCUMENT OR ANY PART THEREOF.

14621 As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards  
14622 Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

14623 [OASIS requests that any OASIS Party or any other party that believes it has patent claims that would  
14624 necessarily be infringed by implementations of this OASIS Standards Final Deliverable, to notify OASIS  
14625 TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims  
14626 in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this  
14627 deliverable.]

14628 [OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of  
14629 any patent claims that would necessarily be infringed by implementations of this OASIS Standards Final  
14630 Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner  
14631 consistent with the IPR Mode of the OASIS Technical Committee that produced this OASIS Standards  
14632 Final Deliverable. OASIS may include such claims on its website, but disclaims any obligation to do so.]

14633 [OASIS takes no position regarding the validity or scope of any intellectual property or other rights that  
14634 might be claimed to pertain to the implementation or use of the technology described in this OASIS  
14635 Standards Final Deliverable or the extent to which any license under such rights might or might not be  
14636 available; neither does it represent that it has made any effort to identify any such rights. Information on  
14637 OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS  
14638 Technical Committee can be found on the OASIS website. Copies of claims of rights made available for  
14639 publication and any assurances of licenses to be made available, or the result of an attempt made to  
14640 obtain a general license or permission for the use of such proprietary rights by implementers or users of  
14641 this OASIS Standards Final Deliverable, can be obtained from the OASIS TC Administrator. OASIS  
14642 makes no representation that any information or list of intellectual property rights will at any time be  
14643 complete, or that any claims in such list are, in fact, Essential Claims.]

14644 The name "OASIS" is a trademark of [OASIS](https://www.oasis-open.org/policies-guidelines/trademark/), the owner and developer of this document, and should be  
14645 used only to refer to the organization and its official outputs. OASIS welcomes reference to, and  
14646 implementation and use of, documents, while reserving the right to enforce its marks against misleading  
14647 uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark/> for above guidance.