This stage:
https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.pdf (Authoritative)
https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.html
https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.docx

Previous 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.html
https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.docx

Technical Committee:
OASIS PKCS 11 TC

Chairs:
Robert Relyea (rrelyea@redhat.com), Red Hat
Greg Scott (greg.scott@cryptsoft.com), Cryptsoft Pty Ltd

Editors:
Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH
Tony Cox (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:
• pkcs11.h: https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/include/pkcs11-v3.1/pkcs11.h
• pkcs11f.h: https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/include/pkcs11-v3.1/pkcs11f.h

Related work:
This specification replaces or supersedes:

This specification is related to:

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

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

TC members should send comments on this document to the TC’s email list. Others should send comments to the TC’s public comment list, after subscribing to it by following the instructions at the “Send A Comment” button on the TC’s web page at https://www.oasis-open.org/committees/pkcs11/.

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

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

**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]


**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/](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 CGetWidth ................................................................................. 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6.1</td>
<td>C_OpenSession</td>
</tr>
<tr>
<td>5.6.2</td>
<td>C_CloseSession</td>
</tr>
<tr>
<td>5.6.3</td>
<td>C_CloseAllSessions</td>
</tr>
<tr>
<td>5.6.4</td>
<td>C_GetSessionInfo</td>
</tr>
<tr>
<td>5.6.5</td>
<td>C_SessionCancel</td>
</tr>
<tr>
<td>5.6.6</td>
<td>C_GetOperationState</td>
</tr>
<tr>
<td>5.6.7</td>
<td>C_SetOperationState</td>
</tr>
<tr>
<td>5.6.8</td>
<td>C_Login</td>
</tr>
<tr>
<td>5.6.9</td>
<td>C_LoginUser</td>
</tr>
<tr>
<td>5.6.10</td>
<td>C_Logout</td>
</tr>
<tr>
<td>5.7</td>
<td>Object management functions</td>
</tr>
<tr>
<td>5.7.1</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>5.7.2</td>
<td>C_CopyObject</td>
</tr>
<tr>
<td>5.7.3</td>
<td>C_DestroyObject</td>
</tr>
<tr>
<td>5.7.4</td>
<td>C_GetObjectSize</td>
</tr>
<tr>
<td>5.7.5</td>
<td>C_GetAttributeValue</td>
</tr>
<tr>
<td>5.7.6</td>
<td>C_SetAttributeValue</td>
</tr>
<tr>
<td>5.7.7</td>
<td>C_FindObjectsInit</td>
</tr>
<tr>
<td>5.7.8</td>
<td>C_FindObjects</td>
</tr>
<tr>
<td>5.7.9</td>
<td>C_FindObjectsFinal</td>
</tr>
<tr>
<td>5.8</td>
<td>Encryption functions</td>
</tr>
<tr>
<td>5.8.1</td>
<td>C_EncryptInit</td>
</tr>
<tr>
<td>5.8.2</td>
<td>C_Encrypt</td>
</tr>
<tr>
<td>5.8.3</td>
<td>C_EncryptUpdate</td>
</tr>
<tr>
<td>5.8.4</td>
<td>C_EncryptFinal</td>
</tr>
<tr>
<td>5.9</td>
<td>Message-based encryption functions</td>
</tr>
<tr>
<td>5.9.1</td>
<td>C_MessageEncryptInit</td>
</tr>
<tr>
<td>5.9.2</td>
<td>C_EncryptMessage</td>
</tr>
<tr>
<td>5.9.3</td>
<td>C_EncryptMessageBegin</td>
</tr>
<tr>
<td>5.9.4</td>
<td>C_EncryptMessageNext</td>
</tr>
<tr>
<td>5.9.5</td>
<td>C_MessageEncryptFinal</td>
</tr>
<tr>
<td>5.10</td>
<td>Decryption functions</td>
</tr>
<tr>
<td>5.10.1</td>
<td>C_DecryptInit</td>
</tr>
<tr>
<td>5.10.2</td>
<td>C_Decrypt</td>
</tr>
<tr>
<td>5.10.3</td>
<td>C_DecryptUpdate</td>
</tr>
<tr>
<td>5.10.4</td>
<td>C_DecryptFinal</td>
</tr>
<tr>
<td>5.11</td>
<td>Message-based decryption functions</td>
</tr>
<tr>
<td>5.11.1</td>
<td>C_MessageDecryptInit</td>
</tr>
<tr>
<td>5.11.2</td>
<td>C_DecryptMessage</td>
</tr>
<tr>
<td>5.11.3</td>
<td>C_DecryptMessageBegin</td>
</tr>
<tr>
<td>5.11.4</td>
<td>C_DecryptMessageNext</td>
</tr>
<tr>
<td>5.11.5</td>
<td>C_MessageDecryptFinal</td>
</tr>
<tr>
<td>5.12</td>
<td>Message digesting functions</td>
</tr>
<tr>
<td>5.12.1</td>
<td>C_DIGESTInit</td>
</tr>
<tr>
<td>5.12.2</td>
<td>C_Digest</td>
</tr>
<tr>
<td>5.12.3</td>
<td>C_DigestUpdate</td>
</tr>
</tbody>
</table>

11 August 2022
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
6 RSA

6.1 Definitions ................................................................. 175
6.2 RSA public key objects .................................................... 176
6.3 RSA private key objects .................................................... 177
6.4 PKCS #1 RSA key pair generation ........................................ 179
6.5 X9.31 RSA key pair generation ........................................... 179
6.6 PKCS #1 v1.5 RSA ........................................................... 179
6.7 PKCS #1 RSA OAEP mechanism parameters .......................... 180
6.8 PKCS #1 RSA OAEP .......................................................... 182
6.9 PKCS #1 RSA PSS mechanism parameters ............................ 182
6.10 PKCS #1 RSA PSS ............................................................ 183
6.11 ISO/IEC 9796 RSA ......................................................... 183
6.12 X.509 (raw) RSA ............................................................. 184
6.13 ANSI X9.31 RSA ............................................................ 185
6.14 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
6.15 PKCS #1 v1.5 RSA signature with SHA-224 .......................... 186
6.16 PKCS #1 RSA PSS signature with SHA-224 .......................... 186
6.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512 ...................................................... 187
6.18 PKCS #1 v1.5 RSA signature with SHA3 ............................... 187
6.19 PKCS #1 RSA PSS signature with SHA3 ................................ 187
6.20 ANSI X9.31 RSA signature with SHA-1 ................................ 187
6.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA ......................... 188
6.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP ....................... 188
6.23 RSA AES KEY WRAP ..................................................... 189
6.24 RSA AES KEY WRAP mechanism parameters .................... 190
6.25 FIPS 186-4 ................................................................. 191

6.2 DSA

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 ........................................................ 198
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-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-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
  6.9.2 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 .................................................................................................................. 263
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 ...................................................................................... 266
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 ............................................................................ 268
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 .............................................................................................. 270
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
6.28.5 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 .......................................................... 296
  6.34.5 BLAKE2B-256 key derivation ............................................... 296
  6.34.6 BLAKE2B-256 HMAC key generation .................................. 296
6.35 BLAKE2B-384 .............................................................................. 296
  6.35.1 Definitions ............................................................................. 297
  6.35.2 BLAKE2B-384 digest ............................................................. 297
  6.35.3 General-length BLAKE2B-384-HMAC .............................. 297
  6.35.4 BLAKE2B-384-HMAC .......................................................... 298
  6.35.5 BLAKE2B-384 key derivation ............................................... 298
6.35.6 BLAKE2B-384 HMAC key generation ......................................................... 298
6.36 BLAKE2B-512 ......................................................................................... 298
  6.36.1 Definitions ......................................................................................... 299
  6.36.2 BLAKE2B-512 digest ......................................................................... 299
  6.36.3 General-length BLAKE2B-512-HMAC .............................................. 299
  6.36.4 BLAKE2B-512-HMAC ................................................................. 300
  6.36.5 BLAKE2B-512 key derivation ......................................................... 300
  6.36.6 BLAKE2B-512 HMAC key generation ............................................ 300
6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE) ...................... 300
  6.37.1 Definitions ......................................................................................... 301
  6.37.2 Password-based encryption/authentication mechanism parameters .......... 301
  6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters .................. 302
  6.37.4 PKCS #5 PBKDF2 key generation ..................................................... 304
6.38 PKCS #12 password-based encryption/authentication mechanisms .................. 304
  6.38.1 SHA-1-PBE for 3-key triple-DES-CBC ............................................. 305
  6.38.2 SHA-1-PBE for 2-key triple-DES-CBC ............................................. 305
  6.38.3 SHA-1-PBA for SHA-1-HMAC ....................................................... 305
6.39 SSL ........................................................................................................ 305
  6.39.1 Definitions ......................................................................................... 306
  6.39.2 SSL mechanism parameters ............................................................. 306
  6.39.3 Pre-master key generation ................................................................. 308
  6.39.4 Master key derivation ....................................................................... 308
  6.39.5 Master key derivation for Diffie-Hellman ........................................ 309
  6.39.6 Key and MAC derivation ............................................................... 310
  6.39.7 MD5 MACing in SSL 3.0 ............................................................... 311
  6.39.8 SHA-1 MACing in SSL 3.0 .............................................................. 311
6.40 TLS 1.2 Mechanisms ............................................................................ 311
  6.40.1 Definitions ......................................................................................... 312
  6.40.2 TLS 1.2 mechanism parameters ....................................................... 312
  6.40.3 TLS MAC ......................................................................................... 315
  6.40.4 Master key derivation ....................................................................... 315
  6.40.5 Master key derivation for Diffie-Hellman ........................................ 316
  6.40.6 Key and MAC derivation ............................................................... 317
  6.40.7 CKM_TLS12_KEY_SAFE_DERIVE .............................................. 317
  6.40.8 Generic Key Derivation using the TLS PRF ...................................... 318
  6.40.9 Generic Key Derivation using the TLS12 PRF .................................... 318
6.41 WTLS ..................................................................................................... 319
  6.41.1 Definitions ......................................................................................... 319
  6.41.2 WTLS mechanism parameters ....................................................... 320
  6.41.3 Pre master secret key generation for RSA key exchange suite ........ ..... 322
  6.41.4 Master secret key derivation ............................................................ 323
  6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography .... 323
  6.41.6 WTLS PRF (pseudorandom function) ............................................ 324
  6.41.7 Server Key and MAC derivation ..................................................... 324
  6.41.8 Client key and MAC derivation ...................................................... 325
6.42 SP 800-108 Key Derivation ................................................................ 326
6.42.1 Definitions .................................................................................................................. 326
6.42.2 Mechanism Parameters .............................................................................................. 327
6.42.3 Counter Mode KDF .................................................................................................... 327
6.42.4 Feedback Mode KDF .................................................................................................. 332
6.42.5 Double Pipeline Mode KDF ....................................................................................... 333
6.42.6 Deriving Additional Keys .......................................................................................... 334
6.42.7 Key Derivation Attribute Rules .................................................................................. 335
6.42.8 Constructing PRF Input Data ...................................................................................... 335
  6.42.8.1 Sample Counter Mode KDF .................................................................................. 335
  6.42.8.2 Sample SCP03 Counter Mode KDF .................................................................... 336
  6.42.8.3 Sample Feedback Mode KDF .............................................................................. 337
  6.42.8.4 Sample Double-Pipeline Mode KDF .................................................................. 339
6.43 Miscellaneous simple key derivation mechanisms ....................................................... 340
  6.43.1 Definitions ............................................................................................................... 340
  6.43.2 Parameters for miscellaneous simple key derivation mechanisms ......................... 340
  6.43.3 Concatenation of a base key and another key ......................................................... 341
  6.43.4 Concatenation of a base key and data .................................................................... 341
  6.43.5 Concatenation of data and a base key ..................................................................... 342
  6.43.6 XORing of a key and data ...................................................................................... 343
  6.43.7 Extraction of one key from another key .................................................................. 344
6.44 CMS .............................................................................................................................. 344
  6.44.1 Definitions ............................................................................................................... 345
  6.44.2 CMS Signature Mechanism Objects ...................................................................... 345
  6.44.3 CMS mechanism parameters .................................................................................. 345
  6.44.4 CMS signatures ...................................................................................................... 346
6.45 Blowfish ......................................................................................................................... 347
  6.45.1 Definitions ............................................................................................................... 348
  6.45.2 BLOWFISH secret key objects .............................................................................. 348
  6.45.3 Blowfish key generation .......................................................................................... 349
  6.45.4 Blowfish-CBC ........................................................................................................ 349
  6.45.5 Blowfish-CBC with PKCS padding ....................................................................... 349
6.46 Twofish .......................................................................................................................... 350
  6.46.1 Definitions ............................................................................................................... 350
  6.46.2 Twofish secret key objects ...................................................................................... 350
  6.46.3 Twofish key generation ........................................................................................... 351
  6.46.4 Twofish -CBC ......................................................................................................... 351
  6.46.5 Twofish-CBC with PKCS padding ....................................................................... 351
6.47 CAMELLIA ...................................................................................................................... 351
  6.47.1 Definitions ............................................................................................................... 352
  6.47.2 Camellia secret key objects ..................................................................................... 352
  6.47.3 Camellia key generation ......................................................................................... 353
  6.47.4 Camellia-ECB ......................................................................................................... 353
  6.47.5 Camellia-CBC ........................................................................................................ 354
  6.47.6 Camellia-CBC with PKCS padding ....................................................................... 354
  6.47.7 CAMELLIA with Counter mechanism parameters ............................................... 355
  6.47.8 General-length Camellia-MAC .............................................................................. 356
  6.47.9 Camellia-MAC ...................................................................................................... 356
6.48 Key derivation by data encryption - Camellia ................................................. 357
  6.48.1 Definitions ........................................................................................................ 357
  6.48.2 Mechanism Parameters ................................................................................. 357
6.49 ARIA ..................................................................................................................... 357
  6.49.1 Definitions ........................................................................................................ 358
  6.49.2 Aria secret key objects ..................................................................................... 358
  6.49.3 ARIA key generation ......................................................................................... 359
  6.49.4 ARIA-ECB ....................................................................................................... 359
  6.49.5 ARIA-CBC ...................................................................................................... 359
  6.49.6 ARIA-CBC with PKCS padding ...................................................................... 360
  6.49.7 General-length ARIA-MAC ........................................................................... 361
  6.49.8 ARIA-MAC ..................................................................................................... 361
6.50 Key derivation by data encryption - ARIA ......................................................... 361
  6.50.1 Definitions ........................................................................................................ 362
  6.50.2 Mechanism Parameters ................................................................................. 362
6.51 SEED .................................................................................................................. 362
  6.51.1 Definitions ........................................................................................................ 363
  6.51.2 SEED secret key objects .................................................................................. 363
  6.51.3 SEED key generation ....................................................................................... 364
  6.51.4 SEED-ECB ...................................................................................................... 364
  6.51.5 SEED-CBC .................................................................................................... 364
  6.51.6 SEED-CBC with PKCS padding ................................................................... 364
  6.51.7 General-length SEED-MAC .......................................................................... 365
  6.51.8 SEED-MAC ..................................................................................................... 365
6.52 Key derivation by data encryption - SEED ......................................................... 365
  6.52.1 Definitions ........................................................................................................ 365
  6.52.2 Mechanism Parameters ................................................................................. 365
6.53 OTP .................................................................................................................... 366
  6.53.1 Usage overview .............................................................................................. 366
  6.53.2 Case 1: Generation of OTP values ................................................................. 366
  6.53.3 Case 2: Verification of provided OTP values ................................................. 367
  6.53.4 Case 3: Generation of OTP keys ..................................................................... 367
  6.53.5 OTP objects .................................................................................................... 368
    6.53.5.1 Key objects ................................................................................................. 368
  6.53.6 OTP-related notifications .............................................................................. 371
  6.53.7 OTP mechanisms ........................................................................................... 371
    6.53.7.1 OTP mechanism parameters .................................................................... 371
  6.53.8 RSA SecurID .................................................................................................. 375
    6.53.8.1 RSA SecurID secret key objects ............................................................... 375
    6.53.8.2 RSA SecurID key generation ................................................................. 376
    6.53.8.3 SecurID OTP generation and validation .............................................. 376
    6.53.8.4 Return values ........................................................................................... 377
  6.53.9 OATH HOTP .................................................................................................. 377
    6.53.9.1 OATH HOTP secret key objects ............................................................ 377
    6.53.9.2 HOTP key generation .............................................................................. 378
    6.53.9.3 HOTP OTP generation and validation ................................................... 378
  6.53.10 Actividentity ACTI ...................................................................................... 378
6.54 CT-KIP ..................................................... 380
6.54.1 Principles of Operation .................................. 380
6.54.2 Mechanisms ............................................. 380
6.54.3 Definitions .............................................. 381
6.54.4 CT-KIP Mechanism parameters ......................... 381
6.54.5 CT-KIP key derivation .................................. 381
6.54.6 CT-KIP key wrap and key unwrap ......................... 382
6.54.7 CT-KIP signature generation ......................... 382
6.55 GOST 28147-89 ............................................. 382
6.55.1 Definitions ............................................. 382
6.55.2 GOST 28147-89 secret key objects ....................... 383
6.55.3 GOST 28147-89 domain parameter objects .............. 383
6.55.4 GOST 28147-89 key generation ........................ 384
6.55.5 GOST 28147-89-ECB .................................. 385
6.55.6 GOST 28147-89 encryption mode except ECB .......... 385
6.55.7 GOST 28147-89-MAC .................................. 386
6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 ............................................. 386
6.56 GOST R 34.11-94 ............................................ 387
6.56.1 Definitions ............................................. 387
6.56.2 GOST R 34.11-94 domain parameter objects ............ 387
6.56.3 GOST R 34.11-94 digest ................................ 388
6.56.4 GOST R 34.11-94 HMAC ................................ 389
6.57 GOST R 34.10-2001 .......................................... 389
6.57.1 Definitions ............................................. 389
6.57.2 GOST R 34.10-2001 public key objects .................. 390
6.57.3 GOST R 34.10-2001 private key objects ................ 391
6.57.4 GOST R 34.10-2001 domain parameter objects ......... 393
6.57.5 GOST R 34.10-2001 mechanism parameters ............. 394
6.57.6 GOST R 34.10-2001 key pair generation ............... 395
6.57.7 GOST R 34.10-2001 without hashing .................... 395
6.57.8 GOST R 34.10-2001 with GOST R 34.11-94 ............. 396
6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001 ............................................. 396
6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys ............................................. 396
6.58 ChaCha20 .................................................... 397
6.58.1 Definitions ............................................. 397
6.58.2 ChaCha20 secret key objects ............................ 397
6.58.3 ChaCha20 mechanism parameters ....................... 398
6.58.4 ChaCha20 key generation ................................ 398
6.58.5 ChaCha20 mechanism ................................... 398
6.59 Salsa20 ..................................................... 399
6.59.1 Definitions ............................................. 400
6.59.2 Salsa20 secret key objects ............................... 400
6.59.3 Salsa20 mechanism parameters ......................... 401
6.59.4 Salsa20 key generation ................................................................. 401
6.59.5 Salsa20 mechanism ................................................................. 401
6.60 Poly1305 ....................................................................................... 402
   6.60.1 Definitions ............................................................................. 402
   6.60.2 Poly1305 secret key objects .................................................. 402
   6.60.3 Poly1305 mechanism ............................................................. 403
6.61 ChaCha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption ......................................................... 403
   6.61.1 Definitions ............................................................................. 404
   6.61.2 Usage .................................................................................... 404
   6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters ................................................................. 406
6.62 HKDF Mechanisms ........................................................................ 407
   6.62.1 Definitions ............................................................................. 407
   6.62.2 HKDF mechanism parameters .............................................. 407
   6.62.3 HKDF derive ......................................................................... 408
   6.62.4 HKDF Data ........................................................................... 409
   6.62.5 HKDF Key gen ..................................................................... 409
6.63 NULL Mechanism ........................................................................... 409
   6.63.1 Definitions ............................................................................. 409
   6.63.2 CKM_NULL mechanism parameters ..................................... 409
6.64 IKE Mechanisms ............................................................................ 409
   6.64.1 Definitions ............................................................................. 410
   6.64.2 IKE mechanism parameters .................................................. 410
   6.64.3 IKE PRF DERIVE .................................................................. 412
   6.64.4 IKEv1 PRF DERIVE ............................................................... 413
   6.64.5 IKEv2 PRF PLUS DERIVE ...................................................... 413
   6.64.6 IKEv1 Extended Derive ........................................................... 414
6.65 HSS .............................................................................................. 414
   6.65.1 Definitions ............................................................................. 415
   6.65.2 HSS public key objects .......................................................... 415
   6.65.3 HSS private key objects ......................................................... 416
   6.65.4 HSS key pair generation ......................................................... 418
   6.65.5 HSS without hashing ............................................................. 418
7 PKCS #11 Implementation Conformance ............................................. 420
   7.1 PKCS#11 Consumer Implementation Conformance .................. 420
   7.2 PKCS#11 Provider Implementation Conformance ....................... 420
Appendix A. Acknowledgments ............................................................. 421
Appendix B. Manifest constants ............................................................ 423
Appendix C. Revision History ............................................................... 424
Appendix D. Notices ............................................................................. 426
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:

- **AES** Advanced Encryption Standard, as defined in FIPS PUB 197.
- **API** Application programming interface.
- **Application** Any computer program that calls the Cryptoki interface.
- **ASN.1** Abstract Syntax Notation One, as defined in X.680.
- **Attribute** A characteristic of an object.
- **BER** Basic Encoding Rules, as defined in X.690.
- **CAMELLIA** The Camellia encryption algorithm, as defined in RFC 3713.
- **CBC** Cipher-Block Chaining mode, as defined in FIPS PUB 81.
- **Certificate** A signed message binding a subject name and a public key, or a subject name and a set of attributes.
- **CDMF** Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
- **CMAC** Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].
- **CMS** Cryptographic Message Syntax (see RFC 5652)
Cryptographic Device: A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.

Cryptoki: The Cryptographic Token Interface defined in this standard.

Cryptoki library: A library that implements the functions specified in this standard.

CT-KIP: Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP]).

DER: Distinguished Encoding Rules, as defined in X.690.

DES: Data Encryption Standard, as defined in FIPS PUB 46-3.

DSA: Digital Signature Algorithm, as defined in FIPS PUB 186-4.

EC: Elliptic Curve

ECB: Electronic Codebook mode, as defined in FIPS PUB 81.


ECDSA: Elliptic Curve DSA, as in ANSI X9.62.

ECMQV: Elliptic Curve Menezes-Qu-Vanstone

GOST 28147-89: The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].

GOST R 34.11-94: Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].

GOST R 34.10-2001: The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].

IV: Initialization Vector.

MAC: Message Authentication Code.

Mechanism: A process for implementing a cryptographic operation.

MQV: Menezes-Qu-Vanstone

OAEP: Optimal Asymmetric Encryption Padding for RSA.

Object: An item that is stored on a token. May be data, a certificate, or a key.

PIN: Personal Identification Number.

PKCS: Public-Key Cryptography Standards.

PRF: Pseudo random function.

PTD: Personal Trusted Device, as defined in MeT-PTD

RSA: The RSA public-key cryptosystem.

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

Session: A logical connection between an application and a token.

SHA-1: The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.

SHA-224: 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.
The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.

SHA-384
The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.

SHA-512
The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.

Slot
A logical reader that potentially contains a token.

SSL
The Secure Sockets Layer 3.0 protocol.

Subject Name
The X.500 distinguished name of the entity to which a key is assigned.

SO
A Security Officer user.

TLS
Transport Layer Security.

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

User
The person using an application that interfaces to Cryptoki.

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

WTLS

1.2 Symbols and abbreviations
The following symbols are used in this standard:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>Not applicable</td>
</tr>
<tr>
<td>R/O</td>
<td>Read-only</td>
</tr>
<tr>
<td>R/W</td>
<td>Read/write</td>
</tr>
</tbody>
</table>

The following prefixes are used in this standard:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_</td>
<td>Function</td>
</tr>
<tr>
<td>CK_</td>
<td>Data type or general constant</td>
</tr>
<tr>
<td>CKA_</td>
<td>Attribute</td>
</tr>
<tr>
<td>CKC_</td>
<td>Certificate type</td>
</tr>
<tr>
<td>CKD_</td>
<td>Key derivation function</td>
</tr>
<tr>
<td>CKF_</td>
<td>Bit flag</td>
</tr>
<tr>
<td>CKG_</td>
<td>Mask generation function</td>
</tr>
<tr>
<td>CKH_</td>
<td>Hardware feature type</td>
</tr>
<tr>
<td>CKK_</td>
<td>Key type</td>
</tr>
<tr>
<td>CKM_</td>
<td>Mechanism type</td>
</tr>
<tr>
<td>CKN_</td>
<td>Notification</td>
</tr>
<tr>
<td>CKO_</td>
<td>Object class</td>
</tr>
</tbody>
</table>
Cryptoki is based on ANSI C types, and defines the following data types:

```c
/* an unsigned 8-bit value */
typedef unsigned char CK_BYTE;

/* an unsigned 8-bit character */
typedef CK_BYTE CK_CHAR;

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

/* a BYTE-sized Boolean flag */
typedef CK_BYTE CK_BBOOL;

/* an unsigned value, at least 32 bits long */
typedef unsigned long int CK_ULONG;

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

/* at least 32 bits; each bit is a Boolean flag */
typedef CK_ULONG CK_FLAGS;
```

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

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

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

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

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

```c
NULL_PTR         /* A NULL pointer */
```
It follows that many of the data and pointer types will vary somewhat from one environment to another (e.g., a CK_ULONG will sometimes be 32 bits, and sometimes perhaps 64 bits). However, these details should not affect an application, assuming it is compiled with Cryptoki header files consistent with the Cryptoki library to which the application is linked.

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

The **CK_CHAR** data type holds characters from the following table, taken from ANSI C:

*Table 3, Character Set*

<table>
<thead>
<tr>
<th>Category</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>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</td>
</tr>
<tr>
<td>Numbers</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Graphic characters</td>
<td>! &quot; # % &amp; ' ( ) + , - . / : ; &lt; = &gt; ? [ ] ^ _ { }</td>
</tr>
<tr>
<td>Blank character</td>
<td>'</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

### 1.3 Normative References


URL: https://cr.yp.to/mac/poly1305-20050329.pdf

URL: https://tools.ietf.org/html/rfc2409


URL: http://www.ietf.org/rfc/rfc2279.txt

URL: http://www.ietf.org/rfc/rfc2534.txt


URL: http://www.ietf.org/rfc/rfc5705.txt

URL: https://tools.ietf.org/html/rfc5996

URL: https://tools.ietf.org/html/rfc8554

URL: http://homes.esat.kuleuven.be/~bossela/riemd160.html

URL: http://cr.yp.to/chacha/chacha-20080128.pdf


URL: http://www.ietf.org/rfc/rfc4346.txt http://www.ietf.org/rfc/rfc4346.txt, which was superseded by [TLS12].
1.4 Non-Normative References

URL: http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html


URL: http://www.ecc-brainpool.org

URL: http://www.w3.org/TR/CCPP-struct-vocab/


URL: http://www.ietf.org/rfc/rfc4309.txt

URL: http://www.ietf.org/rfc/rfc4357.txt

URL: http://www.ietf.org/rfc/rfc4490.txt

URL: http://www.ietf.org/rfc/rfc4491.txt

URL: http://www.ietf.org/rfc/rfc4493.txt

URL: http://www.ietf.org/rfc/rfc5705.txt

URL: http://www.ietf.org/rfc/rfc5869.txt

URL: https://tools.ietf.org/rfc/rfc7539.txt

URL: https://tools.ietf.org/html/rfc7748

URL: https://tools.ietf.org/html/rfc8032


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

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

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

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

2.1 Structure packing

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

2.2 Pointer-related macros

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

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

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

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

CK_CALLBACK_FUNCTION

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

CK_CALLBACK_FUNCTION(CK_RV, myCallback) (args);

or:

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

NULL_PTR

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

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

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

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

3.1 General information

Cryptoki represents general information with the following types:

* CK_VERSION; CK_VERSION_PTR

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an SSL or TLS implementation, or the hardware or firmware version of a slot or token. It is defined as follows:

```c
typedef struct CK_VERSION {
    CK_BYTE major;
    CK_BYTE minor;
} CK_VERSION;
```

The fields of the structure have the following meanings:

- **major** major version number (the integer portion of the version)
- **minor** minor version number (the hundredths portion of the version)

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

<table>
<thead>
<tr>
<th>Version</th>
<th>major</th>
<th>minor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0x01</td>
<td>0x00</td>
</tr>
<tr>
<td>2.01</td>
<td>0x02</td>
<td>0x01</td>
</tr>
<tr>
<td>2.10</td>
<td>0x02</td>
<td>0x0a</td>
</tr>
<tr>
<td>2.11</td>
<td>0x02</td>
<td>0x0b</td>
</tr>
<tr>
<td>2.20</td>
<td>0x02</td>
<td>0x14</td>
</tr>
<tr>
<td>2.30</td>
<td>0x02</td>
<td>0x1e</td>
</tr>
<tr>
<td>2.40</td>
<td>0x02</td>
<td>0x28</td>
</tr>
<tr>
<td>3.0</td>
<td>0x03</td>
<td>0x00</td>
</tr>
</tbody>
</table>

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

CK_VERSION_PTR is a pointer to a CK_VERSION.

* CK_INFO; CK_INFO_PTR

CK_INFO provides general information about Cryptoki. It is defined as follows:
typedef struct CK_INFO {
    CK_VERSION cryptokiVersion;
    CK_UTF8CHAR manufacturerID[32];
    CK_FLAGS flags;
    CK_UTF8CHAR libraryDescription[32];
    CK_VERSION libraryVersion;
} CK_INFO;

The fields of the structure have the following meanings:

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

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

CK_INFO_PTR is a pointer to a CK_INFO.

♦ CK_NOTIFICATION

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

typedef CK_ULONG CK_NOTIFICATION;

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

CKN_SURRENDER

The notifications have the following meanings:

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

3.2 Slot and token types

Cryptoki represents slot and token information with the following types:

♦ CK_SLOT_ID; CK_SLOT_ID_PTR

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

typedef CK_ULONG CK_SLOT_ID;

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

CK_SLOT_INFO; CK_SLOT_INFO_PTR

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

```c
typedef struct CK_SLOT_INFO {
    CK_UTF8CHAR slotDescription[64];
    CK_UTF8CHAR manufacturerID[32];
    CK_FLAGS flags;
    CK_VERSION hardwareVersion;
    CK_VERSION firmwareVersion;
} CK_SLOT_INFO;
```

The fields of the structure have the following meanings:

- `slotDescription`: character-string description of the slot. MUST be padded with the blank character (\'\'). MUST NOT be null-terminated.
- `manufacturerID`: ID of the slot manufacturer. MUST be padded with the blank character (\'\'). MUST NOT be null-terminated.
- `flags`: bits flags that provide capabilities of the slot. The flags are defined below.
- `hardwareVersion`: version number of the slot’s hardware
- `firmwareVersion`: version number of the slot’s firmware

The following table defines the `flags` field:

<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_TOKEN_PRESENT</td>
<td>0x00000001</td>
<td>True if a token is present in the slot (e.g., a device is in the reader)</td>
</tr>
<tr>
<td>CKF_REMOVABLE_DEVICE</td>
<td>0x00000002</td>
<td>True if the reader supports removable devices</td>
</tr>
<tr>
<td>CKF_HW_SLOT</td>
<td>0x00000004</td>
<td>True if the slot is a hardware slot, as opposed to a software slot implementing a “soft token”</td>
</tr>
</tbody>
</table>

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

CK_SLOT_INFO_PTR is a pointer to a CK_SLOT_INFO.

CK_TOKEN_INFO; CK_TOKEN_INFO_PTR

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

```c
typedef struct CK_TOKEN_INFO {
    CK_UTF8CHAR label[32];
    CK_UTF8CHAR manufacturerID[32];
    CK_UTF8CHAR model[16];
    CK_CHAR serialNumber[16];
    CK_FLAGS flags;
    CK_ULONG ulMaxSessionCount;
    CK_ULONG ulSessionCount;
} CK_TOKEN_INFO;
```
The fields of the structure have the following meanings:

- **label**: application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
- **manufacturerID**: ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
- **model**: model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
- **serialNumber**: character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
- **flags**: bit flags indicating capabilities and status of the device as defined below
- **ulMaxSessionCount**: maximum number of sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
- **ulSessionCount**: number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
- **ulMaxRwSessionCount**: maximum number of read/write sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
- **ulRwSessionCount**: number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
- **ulMaxPinLen**: maximum length in bytes of the PIN
- **ulMinPinLen**: minimum length in bytes of the PIN
- **ulTotalPublicMemory**: the total amount of memory on the token in bytes in which public objects may be stored (see CK_TOKEN_INFO Note below)
- **ulFreePublicMemory**: the amount of free (unused) memory on the token in bytes for public objects (see CK_TOKEN_INFO Note below)
- **ulTotalPrivateMemory**: the total amount of memory on the token in bytes in which private objects may be stored (see CK_TOKEN_INFO Note below)
- **ulFreePrivateMemory**: the amount of free (unused) memory on the token in bytes for private objects (see CK_TOKEN_INFO Note below)
- **hardwareVersion**: version number of hardware
- **firmwareVersion**: version number of firmware
The following table defines the `flags` field:

**Table 6, Token Information Flags**

<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_RNG</td>
<td>0x00000001</td>
<td>True if the token has its own random number generator</td>
</tr>
<tr>
<td>CKF_WRITE_PROTECTED</td>
<td>0x00000002</td>
<td>True if the token is write-protected (see below)</td>
</tr>
<tr>
<td>CKF_LOGIN_REQUIRED</td>
<td>0x00000004</td>
<td>True if there are some cryptographic functions that a user MUST be logged in to perform</td>
</tr>
<tr>
<td>CKF_USER_PIN_INITIALIZED</td>
<td>0x00000008</td>
<td>True if the normal user’s PIN has been initialized</td>
</tr>
<tr>
<td>CKF_RESTORE_KEY_NOT_NEEDED</td>
<td>0x00000020</td>
<td>True if a successful save of a session’s cryptographic operations state always contains all keys needed to restore the state of the session</td>
</tr>
<tr>
<td>CKF_CLOCK_ON_TOKEN</td>
<td>0x00000040</td>
<td>True if token has its own hardware clock</td>
</tr>
<tr>
<td>CKF_PROTECTED_AUTHENTICATION_PATH</td>
<td>0x00000100</td>
<td>True if token has a “protected authentication path”, whereby a user can log into the token without passing a PIN through the Cryptoki library</td>
</tr>
<tr>
<td>CKF_DUAL_CRYPTO_OPERATIONS</td>
<td>0x00000200</td>
<td>True if a single session with the token can perform dual cryptographic operations (see Section 5.14)</td>
</tr>
<tr>
<td>CKF_TOKEN_INITIALIZED</td>
<td>0x00000400</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_SECONDARY_AUTHENTICATION</td>
<td>0x00000800</td>
<td>True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag)</td>
</tr>
<tr>
<td>CKF_USER_PIN_COUNT_LOW</td>
<td>0x00010000</td>
<td>True if an incorrect user login PIN has been entered at least once since the last successful authentication.</td>
</tr>
<tr>
<td>Bit Flag</td>
<td>Mask</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CKF_USER_PIN_FINAL_TRY</td>
<td>0x00020000</td>
<td>True if supplying an incorrect user PIN will cause it to become locked.</td>
</tr>
<tr>
<td>CKF_USER_PIN_LOCKED</td>
<td>0x00040000</td>
<td>True if the user PIN has been locked. User login to the token is not possible.</td>
</tr>
<tr>
<td>CKF_USER_PIN_TO_BE_CHANGED</td>
<td>0x00080000</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_SO_PIN_COUNT_LOW</td>
<td>0x00100000</td>
<td>True if an incorrect SO login PIN has been entered at least once since the last successful authentication.</td>
</tr>
<tr>
<td>CKF_SO_PIN_FINAL_TRY</td>
<td>0x00200000</td>
<td>True if supplying an incorrect SO PIN will cause it to become locked.</td>
</tr>
<tr>
<td>CKF_SO_PIN_LOCKED</td>
<td>0x00400000</td>
<td>True if the SO PIN has been locked. SO login to the token is not possible.</td>
</tr>
<tr>
<td>CKF_SO_PIN_TO_BE_CHANGED</td>
<td>0x00800000</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_ERROR_STATE</td>
<td>0x01000000</td>
<td>True if the token failed a FIPS 140-2 self-test and entered an error state.</td>
</tr>
</tbody>
</table>

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

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

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

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

The **CKF_USER_PIN_TO_BE_CHANGED** and **CKF_SO_PIN_TO_BE_CHANGED** flags may always be set to false if the token does not support the functionality. If a PIN is set to the default value, or has expired, the appropriate **CKF_USER_PIN_TO_BE_CHANGED** or **CKF_SO_PIN_TO_BE_CHANGED** flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed, but only the C_SetPIN function can be called. Calling any other function that required the user to be logged in will cause CKR_PIN_EXPIRED to be returned until C_SetPIN is called successfully.
CK_TOKEN_INFO Note: The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount, ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and ulFreePrivateMemory can have the special value CK_UNAVAILABLE_INFORMATION, which means that the token and/or library is unable or unwilling to provide that information. In addition, the fields ulMaxSessionCount and ulMaxRwSessionCount can have the special value CK_EFFECTIVELY_INFINITE, which means that there is no practical limit on the number of sessions (resp. R/W sessions) an application can have open with the token.

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

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

```c
CK_TOKEN_INFO info;

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

CK_TOKEN_INFO_PTR is a pointer to a CK_TOKEN_INFO.

3.3 Session types

Cryptoki represents session information with the following types:

- **CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR**

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

  ```c
typedef CK_ULONG CK_SESSION_HANDLE;
```

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

  ```c
  CK_INVALID_HANDLE
  ```

  CK_SESSION_HANDLE_PTR is a pointer to a CK_SESSION_HANDLE.

- **CK_USER_TYPE**

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

  ```c
typedef CK_ULONG CK_USER_TYPE;
```
For this version of Cryptoki, the following types of users are defined:

- CKU_SO
- CKU_USER
- CKU_CONTEXT_SPECIFIC

**CK_STATE**

CK_STATE holds the session state, as described in [PKCS11-UG]. It is defined as follows:

```c
typedef CK_ULONG CK_STATE;
```

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

- CKS_RO_PUBLIC_SESSION
- CKS_RO_USER_FUNCTIONS
- CKS_RW_PUBLIC_SESSION
- CKS_RW_USER_FUNCTIONS
- CKS_RW_SO_FUNCTIONS

**CK_SESSION_INFO; CK_SESSION_INFO_PTR**

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

```c
typedef struct CK_SESSION_INFO {
    CK_SLOT_ID slotID;
    CK_STATE state;
    CK_FLAGS flags;
    CK_ULONG ulDeviceError;
} CK_SESSION_INFO;
```

The fields of the structure have the following meanings:

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

The following table defines the `flags` field:

<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_RW_SESSION</td>
<td>0x00000002</td>
<td>True if the session is read/write; false if the session is read-only</td>
</tr>
<tr>
<td>CKF_SERIAL_SESSION</td>
<td>0x00000004</td>
<td>This flag is provided for backward compatibility, and should always be set to true</td>
</tr>
</tbody>
</table>

CK_SESSION_INFO_PTR is a pointer to a CK_SESSION_INFO.

### 3.4 Object types

Cryptoki represents object information with the following types:
**CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR**

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

```c
typedef CK_ULONG CK_OBJECT_HANDLE;
```

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

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

```c
CK_INVALID_HANDLE
```

CK_OBJECT_HANDLE_PTR is a pointer to a CK_OBJECT_HANDLE.

**CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR**

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

```c
typedef CK_ULONG CK_OBJECT_CLASS;
```

Object classes are defined with the objects that use them. The type is specified on an object through the CKA_CLASS attribute of the object. Vendor defined values for this type may also be specified.

```c
CKO_VENDOR_DEFINED
```

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

CK_OBJECT_CLASS_PTR is a pointer to a CK_OBJECT_CLASS.

**CK_HW_FEATURE_TYPE**

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

```c
typedef CK_ULONG CK_HW_FEATURE_TYPE;
```

Hardware feature types are defined with the objects that use them. The type is specified on an object through the CKA_HW_FEATURE_TYPE attribute of the object. Vendor defined values for this type may also be specified.

```c
CKH_VENDOR_DEFINED
```

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

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

```c
typedef CK_ULONG CK_KEY_TYPE;
```

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

Vendor defined values for this type may also be specified.

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

**CK_CERTIFICATE_TYPE**

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

```c
typedef CK_ULONG CK_CERTIFICATE_TYPE;
```

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

Vendor defined values for this type may also be specified.

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

**CK_CERTIFICATE_CATEGORY**

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

```c
typedef CK_ULONG CK_CERTIFICATECATEGORY;
```

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_CERTIFICATECATEGORY_UNSPECIFIED</td>
<td>0x00000000UL</td>
<td>No category specified</td>
</tr>
<tr>
<td>CK_CERTIFICATECATEGORY_TOKEN_USER</td>
<td>0x00000001UL</td>
<td>Certificate belongs to owner of the token</td>
</tr>
<tr>
<td>CK_CERTIFICATECATEGORY_AUTHORITY</td>
<td>0x00000002UL</td>
<td>Certificate belongs to a certificate authority</td>
</tr>
<tr>
<td>CK_CERTIFICATECATEGORY_OTHER_ENTITY</td>
<td>0x00000003UL</td>
<td>Certificate belongs to an end entity (i.e.: not a CA)</td>
</tr>
</tbody>
</table>

**CK_ATTRIBUTE_TYPE**

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

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

Vendor defined values for this type may also be specified.

<table>
<thead>
<tr>
<th>Attribute types</th>
<th>CKA_VENDOR_DEFINED</th>
</tr>
</thead>
</table>

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

**CK_ATTRIBUTE; CK_ATTRIBUTE_PTR**

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

```c
typedef struct CK_ATTRIBUTE {
    CK_ATTRIBUTE_TYPE type;
    CK_VOID_PTR pValue;
    CK_ULONG ulValueLen;
} CK_ATTRIBUTE;
```

The fields of the structure have the following meanings:

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

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

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

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

**CK_DATE**

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

```c
typedef struct CK_DATE {
    CK_CHAR year[4];
    CK_CHAR month[2];
    CK_CHAR day[2];
} CK_DATE;
```

The fields of the structure have the following meanings:

- `year` the year ("1900" - "9999")
- `month` the month ("01" - "12")
- `day` the day ("01" - "31")

The fields hold numeric characters from the character set in Table 3, not the literal byte values.
When a Cryptoki object carries an attribute of this type, and the default value of the attribute is specified to be "empty," then Cryptoki libraries SHALL set the attribute's ulValueLen to 0.

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

- **CK_PROFILE_ID; CK_PROFILE_ID_PTR**

  **CK_PROFILE_ID** is an unsigend ulong value representing a specific token profile. It is defined as follows:

  ```c
  typedef CK_ULONG CK_PROFILE_ID;
  ```

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

  Vendor defined values for this type may also be specified.

  ```c
  CK_PROFILE_ID_PTR is a pointer to a CK_PROFILE_ID.
  ```

- **CK_JAVA_MIDP_SECURITY_DOMAIN**

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

  ```c
  typedef CK_ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
  ```

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SECURITY_DOMAIN_UNSPECIFIED</td>
<td>0x00000000UL</td>
<td>No domain specified</td>
</tr>
<tr>
<td>CK_SECURITY_DOMAIN_MANUFACTURER</td>
<td>0x00000001UL</td>
<td>Manufacturer protection domain</td>
</tr>
<tr>
<td>CK_SECURITY_DOMAIN_OPERATOR</td>
<td>0x00000002UL</td>
<td>Operator protection domain</td>
</tr>
<tr>
<td>CK_SECURITY_DOMAIN_THIRD_PARTY</td>
<td>0x00000003UL</td>
<td>Third party protection domain</td>
</tr>
</tbody>
</table>

### 3.5 Data types for mechanisms

Cryptoki supports the following types for describing mechanisms and parameters to them:
MK MECHANISM_TYPE; CK MECHANISM_TYPE_PTR

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

typedef CK_ULONG CK_MECHANISM_TYPE;

Mechanism types are defined with the objects and mechanism descriptions that use them. Vendor defined values for this type may also be specified.

MK VENDOR_DEFINED

Mechanism types CKM_VENDOR_DEFINED and above are permanently reserved for token vendors. For interoperability, vendors should register their mechanism types through the PKCS process.

MK MECHANISM_TYPE_PTR is a pointer to a CK MECHANISM_TYPE.

MK MECHANISM; CK MECHANISM_PTR

MK MECHANISM is a structure that specifies a particular mechanism and any parameters it requires. It is defined as follows:

typedef struct CK_MECHANISM {
   CK_MECHANISM_TYPE mechanism;
   CK_VOID_PTR pParameter;
   CK_ULONG ulParameterLen;
} CK_MECHANISM;

The fields of the structure have the following meanings:

mechanism the type of mechanism
pParameter pointer to the parameter if required by the mechanism
ulParameterLen length in bytes of the parameter

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

MK MECHANISM_PTR is a pointer to a CK_MECHANISM.

MK MECHANISM_INFO; CK MECHANISM_INFO_PTR

MK MECHANISM_INFO is a structure that provides information about a particular mechanism. It is defined as follows:

typedef struct CK_MECHANISM_INFO {
   CK_ULONG ulMinKeySize;
   CK_ULONG ulMaxKeySize;
   CK_FLAGS flags;
} CK_MECHANISM_INFO;

The fields of the structure have the following meanings:

ulMinKeySize the minimum size of the key for the mechanism (whether this is measured in bits or in bytes is mechanism-dependent)
ulMaxKeySize the maximum size of the key for the mechanism (whether this is measured in bits or in bytes is mechanism-dependent)
flags bit flags specifying mechanism capabilities
For some mechanisms, the `ulMinKeySize` and `ulMaxKeySize` fields have meaningless values.

The following table defines the `flags` field:

**Table 8, Mechanism Information Flags**

<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_HW</td>
<td>0x00000001</td>
<td>True if the mechanism is performed by the device; false if the mechanism is performed in software</td>
</tr>
<tr>
<td>CKF_MESSAGE_ENCRYPT</td>
<td>0x00000002</td>
<td>True if the mechanism can be used with <code>C_MessageEncryptInit</code></td>
</tr>
<tr>
<td>CKF_MESSAGE_DECRYPT</td>
<td>0x00000004</td>
<td>True if the mechanism can be used with <code>C_MessageDecryptInit</code></td>
</tr>
<tr>
<td>CKF_MESSAGE_SIGN</td>
<td>0x00000008</td>
<td>True if the mechanism can be used with <code>C_MessageSignInit</code></td>
</tr>
<tr>
<td>CKF_MESSAGE_VERIFY</td>
<td>0x00000010</td>
<td>True if the mechanism can be used with <code>C_MessageVerifyInit</code></td>
</tr>
<tr>
<td>CKF_MULTI_MESSAGE</td>
<td>0x00000020</td>
<td>True if the mechanism can be used with <code>C_*MessageBegin</code>. One of CKF_MESSAGE_* flag must also be set.</td>
</tr>
<tr>
<td>CKF_FIND_OBJECTS</td>
<td>0x00000040</td>
<td>This flag can be passed in as a parameter to <code>C_SessionCancel</code> to cancel an active object search operation. Any other use of this flag is outside the scope of this standard.</td>
</tr>
<tr>
<td>CKF_ENCRYPT</td>
<td>0x00000100</td>
<td>True if the mechanism can be used with <code>C_EncryptInit</code></td>
</tr>
<tr>
<td>CKF_DECRYPT</td>
<td>0x00000200</td>
<td>True if the mechanism can be used with <code>C_DecryptInit</code></td>
</tr>
<tr>
<td>CKF_DIGEST</td>
<td>0x00000400</td>
<td>True if the mechanism can be used with <code>C_DigestInit</code></td>
</tr>
<tr>
<td>CKF_SIGN</td>
<td>0x00000800</td>
<td>True if the mechanism can be used with <code>C_SignInit</code></td>
</tr>
<tr>
<td>CKF_SIGN_RECOVER</td>
<td>0x00001000</td>
<td>True if the mechanism can be used with <code>C_SignRecoverInit</code></td>
</tr>
<tr>
<td>CKF_VERIFY</td>
<td>0x00002000</td>
<td>True if the mechanism can be used with <code>C_VerifyInit</code></td>
</tr>
<tr>
<td>CKF_VERIFY_RECOVER</td>
<td>0x00004000</td>
<td>True if the mechanism can be used with <code>C_VerifyRecoverInit</code></td>
</tr>
<tr>
<td>CKF_GENERATE</td>
<td>0x00008000</td>
<td>True if the mechanism can be used with <code>C_GenerateKey</code></td>
</tr>
<tr>
<td>CKF_GENERATE_KEY_PAIR</td>
<td>0x00010000</td>
<td>True if the mechanism can be used with <code>C_GenerateKeyPair</code></td>
</tr>
<tr>
<td>CKF_WRAP</td>
<td>0x00020000</td>
<td>True if the mechanism can be used with <code>C_WrapKey</code></td>
</tr>
<tr>
<td>CKF_UNWRAP</td>
<td>0x00040000</td>
<td>True if the mechanism can be used with <code>C_UnwrapKey</code></td>
</tr>
<tr>
<td>CKF_DERIVE</td>
<td>0x00080000</td>
<td>True if the mechanism can be used with <code>C_DeriveKey</code></td>
</tr>
<tr>
<td>Bit Flag</td>
<td>Mask</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CKF_EXTENSION</td>
<td>0x80000000</td>
<td>True if there is an extension to the flags; false if no extensions. MUST be false for this version.</td>
</tr>
</tbody>
</table>

CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

### 3.6 Function types

Cryptoki represents information about functions with the following data types:

**CK_RV**

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

```c
typedef CK_ULONG CK_RV;
```

Vendor defined values for this type may also be specified.

```c
CKR_VENDOR_DEFINED
```

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

**CK_NOTIFY**

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

```c
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY)(
    CK_SESSION_HANDLE hSession,
    CK_NOTIFICATION event,
    CK_VOID_PTR pApplication
);
```

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

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

**CK_C_XXX**

Cryptoki also defines an entire family of other function pointer types. For each function C_XXX in the Cryptoki API (see Section 4.12 for detailed information about each of them), Cryptoki defines a type CK_C_XXX, which is a pointer to a function with the same arguments and return value as C_XXX has.

An appropriately-set variable of type CK_C_XXX may be used by an application to call the Cryptoki function C_XXX.
CK_FUNCTION_LIST is a structure which contains a Cryptoki version and a function pointer to each function in the Cryptoki API. It is defined as follows:

```c
typedef struct CK_FUNCTION_LIST {
    CK_VERSION version;
    CK_C_Initialize C_Initialize;
    CK_C_Finalize C_Finalize;
    CK_GetInfo C_GetInfo;
    CK_GetFunctionList C_GetFunctionList;
    CK_GetSlotList C_GetSlotList;
    CK_GetSlotInfo C_GetSlotInfo;
    CK_GetTokenInfo C_GetTokenInfo;
    CK_GetMechanismList C_GetMechanismList;
    CK_GetMechanismInfo C_GetMechanismInfo;
    CK_InitToken C_InitToken;
    CK_InitPIN C_InitPIN;
    CK_SetPIN C_SetPIN;
    CK_OpenSession C_OpenSession;
    CK_CloseSession C_CloseSession;
    CK_CloseAllSessions C_CloseAllSessions;
    CK_GetSessionInfo C_GetSessionInfo;
    CK_GetOperationState C_GetOperationState;
    CK_SetOperationState C_SetOperationState;
    CK_Login C_Login;
    CK_Logout C_Logout;
    CK_CreateObject C_CreateObject;
    CK_CopyObject C_CopyObject;
    CK_DestroyObject C_DestroyObject;
    CK_GetObjectSize C_GetObjectSize;
    CK_GetAttributeValue C_GetAttributeValue;
    CK_SetAttributeValue C_SetAttributeValue;
    CK_FindObjectsInit C_FindObjectsInit;
    CK_FindObjects C_FindObjects;
    CK_FindObjectsFinal C_FindObjectsFinal;
    CK_EncryptInit C_EncryptInit;
    CK_Encrypt C_Encrypt;
    CK_EncryptUpdate C_EncryptUpdate;
    CK_EncryptFinal C_EncryptFinal;
    CK_DecryptInit C_DecryptInit;
    CK_Decrypt C_Decrypt;
    CK_DecryptUpdate C_DecryptUpdate;
    CK_DecryptFinal C_DecryptFinal;
    CK_DigestInit C_DigestInit;
    CK_Digest C_Digest;
    CK_DigestUpdate C_DigestUpdate;
    CK_DigestKey C_DigestKey;
    CK_DigestFinal C_DigestFinal;
    CK_SignInit C_SignInit;
    CK_Sign C_Sign;
    CK_SignUpdate C_SignUpdate;
    CK_SignFinal C_SignFinal;
    CK_SignRecoverInit C_SignRecoverInit;
    CK_SignRecover C_SignRecover;
    CK_VerifyInit C_VerifyInit;
    CK_Verify C_Verify;
    CK_VerifyUpdate C_VerifyUpdate;
    CK_VerifyFinal C_VerifyFinal;
    CK_VerifyRecoverInit C_VerifyRecoverInit;
    CK_VerifyRecover C_VerifyRecover;
};
```
Each Cryptoki library has a static CK_FUNCTION_LIST structure, and a pointer to it (or a copy of it which is also owned by the library) may be obtained by the C_GetFunctionList function (see Section 5.2). The value that this pointer points to can be used by an application to quickly find out where the executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API MUST have an entry point defined in the Cryptoki library's CK_FUNCTION_LIST structure. If a particular function in the Cryptoki API is not supported by a library, then the function pointer for that function in the library's CK_FUNCTION_LIST structure should point to a function stub which simply returns CKR_FUNCTION_NOT_SUPPORTED.

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

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

PKCS #11 modules must not add new functions at the end of the CK_FUNCTION_LIST that are not contained within the defined structure. If a PKCS#11 module needs to define additional functions, they should be placed within a vendor defined interface returned via C_GetInterfaceList or C_GetInterface.

CK_FUNCTION_LIST_PTR is a pointer to a CK_FUNCTION_LIST.

CK_FUNCTION_LIST_PTR_PTR is a pointer to a CK_FUNCTION_LIST_PTR.

♦ CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR;
CK_FUNCTION_LIST_3_0_PTR_PTR

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

```c
typedef struct CK_FUNCTION_LIST_3_0 {
    CK_VERSION version;
    CK_C_Initialize C_Initialize;
    CK_C_Finalize C_Finalize;
    CK_C_GetInfo C_GetInfo;
    CK_C_GetFunctionList C_GetFunctionList;
    CK_C_GetSlotList C_GetSlotList;
    CK_C_GetSlotInfo C_GetSlotInfo;
    CK_C_GetTokenInfo C_GetTokenInfo;
    CK_C_GetMechanismList C_GetMechanismList;
    CK_C_GetMechanismInfo C_GetMechanismInfo;
    CK_C_InitToken C_InitToken;
    CK_C_InitPIN C_InitPIN;
} CK_FUNCTION_LIST_3_0;
```
CK_C_SetPIN C_SetPIN;
CK_C_OpenSession C_OpenSession;
CK_C_CloseSession C_CloseSession;
CK_C_CloseAllSessions C_CloseAllSessions;
CK_C_GetSessionInfo C_GetSessionInfo;
CK_C_GetOperationState C_GetOperationState;
CK_C_SetOperationState C_SetOperationState;
CK_C_Login C_Login;
CK_C_Logout C_Logout;
CK_C_CreateObject C_CreateObject;
CK_C_CopyObject C_CopyObject;
CK_C_DestroyObject C_DestroyObject;
CK_C_GetObjectSize C_GetObjectSize;
CK_C_GetAttributeValue C_GetAttributeValue;
CK_C_SetAttributeValue C_SetAttributeValue;
CK_C_FindObjectsInit C_FindObjectsInit;
CK_C_FindObjects C_FindObjects;
CK_C_FindObjectsFinal C_FindObjectsFinal;
CK_C_EncryptInit C_EncryptInit;
CK_C_Encrypt C_Encrypt;
CK_C_EncryptUpdate C_EncryptUpdate;
CK_C_EncryptFinal C_EncryptFinal;
CK_C_DecryptInit C_DecryptInit;
CK_C_Decrypt C_Decrypt;
CK_C_DecryptUpdate C_DecryptUpdate;
CK_C_DecryptFinal C_DecryptFinal;
CK_C_DigestInit C_DigestInit;
CK_C_Digest C_Digest;
CK_C_DigestUpdate C_DigestUpdate;
CK_C_DigestKey C_DigestKey;
CK_C_DigestFinal C_DigestFinal;
CK_C_SignInit C_SignInit;
CK_C_Sign C_Sign;
CK_C_SignUpdate C_SignUpdate;
CK_C_SignFinal C_SignFinal;
CK_C_SignRecoverInit C_SignRecoverInit;
CK_C_SignRecover C_SignRecover;
CK_C_SignRecoverInit C_SignRecoverInit;
CK_C_SignRecover C_SignRecover;
CK_C_WrapKey C_WrapKey;
CK_C_UnwrapKey C_UnwrapKey;
CK_C_DeriveKey C_DeriveKey;
CK_C_GenerateRandom C_GenerateRandom;
CK_C_GetFunctionStatus C_GetFunctionStatus;
CK_C_CancelFunction C_CancelFunction;
CK_C_WaitForSlotEvent C_WaitForSlotEvent;
CK_C_GetInterfaceList C_GetInterfaceList;
CK_C_GetInterface C_GetInterface;
CK_C_LoginUser C_LoginUser;
CK_C_SessionCancel C_SessionCancel;
CK_C_MessageEncryptInit C_MessageEncryptInit;
CK_C_EncryptMessage C_EncryptMessage;
CK_C_EncryptMessageBegin C_EncryptMessageBegin;
CK_FUNCTION_LIST_3_0;

For a general description of CK_FUNCTION_LIST_3_0 see CK_FUNCTION_LIST.

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

This function list may be returned via C_GetInterfaceList or C_GetInterface

CK_FUNCTION_LIST_3_0_PTR

CK_INTERFACE_PTR_PTR

CK_INTERFACE

CK_INTERFACE_PTR

CK_INTERFACE;

CK_INTERFACE_PTR_PTR

CK_INTERFACE

CK_INTERFACE_PTR

CK_INTERFACE;

The fields of the structure have the following meanings:

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

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

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

The following table defines the flags field:

Table 9, CK_INTERFACE Flags
<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_INTERFACE_FORK_SAFE</td>
<td>0x00000001</td>
<td>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.</td>
</tr>
</tbody>
</table>

1198

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

1199

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

1200

### 3.7 Locking-related types

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

1204

#### CK_CREATEMUTEX

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

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

1207

1208

1209

1210

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

- CKR_OK, CKR_GENERAL_ERROR
- CKR_HOST_MEMORY

1211

1212

1213

1214

#### CK_DestroyMutex

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

```c
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DestroyMutex)(
    CK VOID_PTR pMutex
);
```

1218

1219

1220

1221

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

- CKR_OK, CKR_GENERAL_ERROR
- CKR_HOST_MEMORY
- CKR_MUTEX_BAD

1222

1223

1224

1225

1226
CK_LOCKMUTEX and CK_UNLOCKMUTEX

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

- If a CK_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a lock on that mutex and returns.
- If a CK_LOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- If a CK_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the behavior of the function call is undefined.
- If a CK_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:
  - If exactly one thread was blocking on that particular mutex, then that thread stops blocking, obtains a lock on that mutex, and its CK_LOCKMUTEX call returns.
  - If more than one thread was blocking on that particular mutex, then exactly one of the blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on the mutex, and its CK_LOCKMUTEX call returns. All other threads blocking on that particular mutex continue to block.
- If a CK_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call returns the error code CKR_MUTEX_NOT_LOCKED.
- If a CK_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the behavior of the function call is undefined.

CK_LOCKMUTEX is defined as follows:

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

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

- CKR_OK
- CKR_GENERAL_ERROR
- CKR_HOST_MEMORY
- CKR_MUTEX_BAD

CK_UNLOCKMUTEX is defined as follows:

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

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

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

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

The fields of the structure have the following meanings:

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

The following table defines the flags field:

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

CK_C_INITIALIZE_ARGS_PTR is a pointer to a CK_C_INITIALIZE_ARGS.
4 Objects

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

![Diagram of object attribute hierarchy]

Figure 1, Object Attribute Hierarchy

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

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

Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (e.g., `CK_OBJECT_CLASS`). Attribute values may also take the following types:

- **Byte array** an arbitrary string (array) of `CK_BYTE`es
- **Big integer** a string of `CK_BYTE`es representing an unsigned integer of arbitrary size, most-significant byte first (e.g., the integer 32768 is represented as the 2-byte string 0x80 0x00)
- **Local string** an unpadded string of `CK_CHAR`es (see Table 3) with no null-termination
- **RFC2279 string** an unpadded string of `CK_UTF8CHAR`es with no null-termination

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

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

### 4.1 Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 5.18) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see [PKCS11-Curr] section 6 Mechanisms and [PKCS11-Hist] for specification of mechanisms for PKCS #11). In any case, all the required attributes supported by an object class that do not have default values MUST be specified when an object is created, either in the template or by the function itself.

#### 4.1.1 Creating objects

Objects may be created with the Cryptoki functions **C.CreateObject** (see Section 5.7), **C.GenerateKey**, **C.GenerateKeyPair**, **C.UnwrapKey**, and **C DeriveKey** (see Section 5.18). In addition, copying an existing object (with the function **C.CopyObject**) also creates a new object, but we consider this type of object creation separately in Section 4.1.3.

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

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

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

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

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

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

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

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

4.1.2 Modifying objects

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

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

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

4.1.3 Copying objects

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

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

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

4.2 Common attributes

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

Table 12, Common Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CKA_CLASS</code></td>
<td><code>CK_OBJECT_CLASS</code></td>
<td>Object class (type)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The above table defines the attributes common to all objects.

4.3 Hardware Feature Objects

4.3.1 Definitions

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

4.3.2 Overview

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

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

Table 13, Hardware Feature Common Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CKA_HW_FEATURE_TYPE</code></td>
<td><code>CK_HW_FEATURE_TYPE</code></td>
<td>Hardware feature (type)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes
4.3.3 Clock

4.3.3.1 Definition
The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type CK_HW_FEATURE_TYPE.

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

Table 14, Clock Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>CK_CHAR[16]</td>
<td>Current time as a character-string of length 16, represented in the format YYYYMMDDhhmssxx (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).</td>
</tr>
</tbody>
</table>

The CKA_VALUE attribute may be set using the C_SetAttributeValue function if permitted by the device. The session used to set the time MUST be logged in. The device may require the SO to be the user logged in to modify the time value. C_SetAttributeValue will return the error CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.

4.3.4 Monotonic Counter Objects

4.3.4.1 Definition
The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type CK_HW_FEATURE_TYPE.

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

Table 15, Monotonic Counter Attributes

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

1Read Only

The CKA_VALUE attribute may not be set by the client.

4.3.5 User Interface Objects

4.3.5.1 Definition
The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type CK_HW_FEATURE_TYPE.
### 4.3.5.2 Description

User interface objects represent the presentation capabilities of the device.

#### Table 16, User Interface Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PIXEL_X</td>
<td>CK_ULONG</td>
<td>Screen resolution (in pixels) in X-axis (e.g. 1280)</td>
</tr>
<tr>
<td>CKA_PIXEL_Y</td>
<td>CK_ULONG</td>
<td>Screen resolution (in pixels) in Y-axis (e.g. 1024)</td>
</tr>
<tr>
<td>CKA_RESOLUTION</td>
<td>CK_ULONG</td>
<td>DPI, pixels per inch</td>
</tr>
<tr>
<td>CKA_CHAR_ROWS</td>
<td>CK_ULONG</td>
<td>For character-oriented displays; number of character rows (e.g. 24)</td>
</tr>
<tr>
<td>CKA_CHAR_COLUMNS</td>
<td>CK_ULONG</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_COLOR</td>
<td>CK_BBOOL</td>
<td>Color support</td>
</tr>
<tr>
<td>CKA_BITS_PER_PIXEL</td>
<td>CK_ULONG</td>
<td>The number of bits of color or grayscale information per pixel.</td>
</tr>
<tr>
<td>CKA_CHAR_SETS</td>
<td>RFC 2279  string</td>
<td>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”.</td>
</tr>
<tr>
<td>CKA_ENCODING_METHODS</td>
<td>RFC 2279  string</td>
<td>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”.</td>
</tr>
<tr>
<td>CKA_MIME_TYPES</td>
<td>RFC 2279  string</td>
<td>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 &quot;a/b&quot;, &quot;a/c&quot; and &quot;a/d&quot; would set the attribute value to &quot;a/b;a/c;a/d&quot;.</td>
</tr>
</tbody>
</table>

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

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

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

### 4.4 Storage Objects

This is not an object class; hence no CKO_ definition is required. It is a category of object classes with common attributes for the object classes that follow.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_TOKEN</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.</td>
</tr>
<tr>
<td>CKA_PRIVATE</td>
<td>CK_BBOOL</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_MODIFIABLE</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if object can be modified Default is CK_TRUE.</td>
</tr>
<tr>
<td>CKA_LABEL</td>
<td>RFC2279 string</td>
<td>Description of the object (default empty).</td>
</tr>
<tr>
<td>CKA_COPYABLE</td>
<td>CK_BBOOL</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_DESTROYABLE</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.</td>
</tr>
<tr>
<td>CKA_UNIQUE_ID246</td>
<td>RFC2279 string</td>
<td>The unique identifier assigned to the object.</td>
</tr>
</tbody>
</table>

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

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

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

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

The CKA_LABEL attribute is intended to assist users in browsing.

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

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

4.4.1 The CKA_UNIQUE_ID attribute

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

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

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

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_APPLICATION</td>
<td>RFC2279 string</td>
<td>Description of the application that manages the object (default empty)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID</td>
<td>Byte Array</td>
<td>DER-encoding of the object identifier indicating the data object type (default empty)</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Value of the object (default empty)</td>
</tr>
</tbody>
</table>

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

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

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

```c
CK_OBJECT_CLASS class = CKO_DATA;
CK_UTF8CHAR label[] = "A data object";
CK_UTF8CHAR application[] = "An application";
CK_BYTE data[] = "Sample data";
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_APPLICATION, application, sizeof(application)-1},
    {CKA_VALUE, data, sizeof(data)}
};
```

4.6 Certificate objects

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

4.6.2 Overview
Certificate objects (object class CKO_CERTIFICATE) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes defined for this object class:
Table 19, Common Certificate Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_CERTIFICATE_TYPE</td>
<td>CK_CERTIFICATE_TYPE</td>
<td>Type of certificate</td>
</tr>
<tr>
<td>CKA_TRUSTED</td>
<td>CK_BBOOL</td>
<td>The certificate can be trusted for the application that it was created.</td>
</tr>
<tr>
<td>CKA_CERTIFICATE_CATEGORY</td>
<td>CKA_CERTIFICATE_CATEGORY</td>
<td>(default CK_CERTIFICATE_CATEGORY_UNSPECIFIED)</td>
</tr>
<tr>
<td>CKA_CHECK_VALUE</td>
<td>Byte array</td>
<td>Checksum</td>
</tr>
<tr>
<td>CKA_START_DATE</td>
<td>CK_DATE</td>
<td>Start date for the certificate (default empty)</td>
</tr>
<tr>
<td>CKA_END_DATE</td>
<td>CK_DATE</td>
<td>End date for the certificate (default empty)</td>
</tr>
<tr>
<td>CKA_PUBLIC_KEY_INFO</td>
<td>Byte Array</td>
<td>DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

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

- X.509 public key certificate
- WTLS public key certificate
- X.509 attribute certificate

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

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

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

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

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

4.6.3 X.509 public key certificate objects

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

Table 20, X.509 Certificate Object Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT1</td>
<td>Byte array</td>
<td>DER-encoding of the certificate subject name</td>
</tr>
<tr>
<td>CKA_ID</td>
<td>Byte array</td>
<td>Key identifier for public/private key pair (default empty)</td>
</tr>
<tr>
<td>CKA_ISSUER</td>
<td>Byte array</td>
<td>DER-encoding of the certificate issuer name (default empty)</td>
</tr>
<tr>
<td>CKA_SERIAL_NUMBER</td>
<td>Byte array</td>
<td>DER-encoding of the certificate serial number (default empty)</td>
</tr>
<tr>
<td>CKA_VALUE2</td>
<td>Byte array</td>
<td>BER-encoding of the certificate</td>
</tr>
<tr>
<td>CKA_URL3</td>
<td>RFC2279 string</td>
<td>If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)</td>
</tr>
<tr>
<td>CKA_HASH_OF_SUBJECT_PUBLIC_KEY4</td>
<td>Byte array</td>
<td>Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM</td>
</tr>
<tr>
<td>CKA_HASH_OF_ISSUER_PUBLIC_KEY4</td>
<td>Byte array</td>
<td>Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM</td>
</tr>
<tr>
<td>CKA_JAVA_MIDP_SECURITY_DOMAIN</td>
<td>CK_JAVA_MIDP_SECURITY_DOMAIN</td>
<td>Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)</td>
</tr>
<tr>
<td>CKA_NAME_HASH_ALGORITHM</td>
<td>CK_MECHANISM_TY</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

Only the `CKA_ID`, `CKA_ISSUER`, and `CKA_SERIAL_NUMBER` attributes may be modified after the object is created.

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

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

The `CKA_ISSUER` and `CKA_SERIAL_NUMBER` attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the `CKA_ID` value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.
The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

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

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

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

```c
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

### 4.6.4 WTLS public key certificate objects

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

**Table 21: WTLS Certificate Object Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Byte array</td>
<td>WTLS-encoding (Identifier type) of the certificate subject</td>
</tr>
<tr>
<td>CKA_ISSUER</td>
<td>Byte array</td>
<td>WTLS-encoding (Identifier type) of the certificate issuer (default empty)</td>
</tr>
<tr>
<td>CKA_VALUE&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Byte array</td>
<td>WTLS-encoding of the certificate</td>
</tr>
<tr>
<td>CKA_URL&lt;sup&gt;3&lt;/sup&gt;</td>
<td>RFC2279 string</td>
<td>If not empty this attribute gives the URL where the complete certificate can be obtained</td>
</tr>
<tr>
<td>CKA_HASH_OF_SUBJECT_PUBLIC_KEY&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Byte array</td>
<td>SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by <strong>CKA_NAME_HASH_ALGORITHM</strong></td>
</tr>
<tr>
<td>CKA_HASH_OF_ISSUER_PUBLIC_KEY&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Byte array</td>
<td>SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by <strong>CKA_NAME_HASH_ALGORITHM</strong></td>
</tr>
<tr>
<td>CKA_NAME_HASH_ALGORITHM</td>
<td></td>
<td>Defines the mechanism used to calculate <strong>CKA_HASH_OF_SUBJECT_PUBLIC</strong></td>
</tr>
<tr>
<td>Attribute</td>
<td>Data type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

The encoding for the CKA_SUBJECT, CKA_ISSUER, and CKA_VALUE attributes can be found in [WTLS].

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

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

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

```c
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_WTLS;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] =
{
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

### 4.6.5 X.509 attribute certificate objects

X.509 attribute certificate objects (certificate type CKC X.509_ATTRCERT) hold X.509 attribute certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the common attributes defined for this object class:
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_OWNER(^1)</td>
<td>Byte Array</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_AC_ISSUER</td>
<td>Byte Array</td>
<td>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)</td>
</tr>
<tr>
<td>CKA_SERIAL_NUMBER</td>
<td>Byte Array</td>
<td>DER-encoding of the certificate serial number. (default empty)</td>
</tr>
<tr>
<td>CKA_ATTR_TYPES</td>
<td>Byte Array</td>
<td>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)</td>
</tr>
<tr>
<td>CKA_VALUE(^1)</td>
<td>Byte Array</td>
<td>BER-encoding of the certificate.</td>
</tr>
</tbody>
</table>

\(^1\)MUST be specified when the object is created

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

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

```c
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
CK_UTF8CHAR label[] = "An attribute certificate object";
CK_BYTE owner[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OWNER, owner, sizeof(owner)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

### 4.7 Key objects

#### 4.7.1 Definitions

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

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

#### 4.7.2 Overview

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

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes defined for this object class:
Table 23, Common Key Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_KEY_TYPE</td>
<td>CK_KEY_TYPE</td>
<td>Type of key</td>
</tr>
<tr>
<td>CKA_ID</td>
<td>Byte array</td>
<td>Key identifier for key (default empty)</td>
</tr>
<tr>
<td>CKA_START_DATE</td>
<td>CK_DATE</td>
<td>Start date for the key (default empty)</td>
</tr>
<tr>
<td>CKA_END_DATE</td>
<td>CK_DATE</td>
<td>End date for the key (default empty)</td>
</tr>
<tr>
<td>CKA_DERIVE</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports key derivation (i.e., if other keys can be derived from this one (default CK_FALSE)</td>
</tr>
<tr>
<td>CKA_LOCAL</td>
<td>CK_BBOOL</td>
<td>CK_TRUE only if key was either</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• generated locally (i.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_KEY_GEN_MECHANISM</td>
<td>CK_MECHANISM_TYPE</td>
<td>Identifier of the mechanism used to generate the key material.</td>
</tr>
<tr>
<td>CKA_ALLOWED_MECHANISMS</td>
<td>CK_MECHANISM_TYPE_PTR,</td>
<td>A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the ulValueLen component of the attribute divided by the size of CK_MECHANISM_TYPE.</td>
</tr>
<tr>
<td></td>
<td>pointer to a CK_MECHANISM_TYPE array</td>
<td></td>
</tr>
</tbody>
</table>

- Refer to Table 11 for footnotes

The CKA_ID field is intended to distinguish among multiple keys. In the case of public and private keys, this field assists in handling multiple keys held by the same subject; the key identifier for a public key and its corresponding private key should be the same. The key identifier should also be the same as for the corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See Section 4.6 for further commentary.) In the case of secret keys, the meaning of the CKA_ID attribute is up to the application. Note that the CKA_START_DATE and CKA_END_DATE attributes are for reference only; Cryptoki does not attach any special meaning to them. In particular, it does not restrict usage of a key according to the dates; doing this is up to the application. The CKA_DERIVE attribute has the value CK_TRUE if and only if it is possible to derive other keys from the key. The CKA_LOCAL attribute has the value CK_TRUE if and only if the value of the key was originally generated on the token by a C_GenerateKey or C_GenerateKeyPair call. The CKA_KEY_GEN_MECHANISM attribute identifies the key generation mechanism used to generate the key material. It contains a valid value only if the CKA_LOCAL attribute has the value CK_TRUE. If CKA_LOCAL has the value CK_FALSE, the value of the attribute is CK_UNAVAILABLE_INFORMATION.

4.8 Public key objects

Public key objects (object class CKO_PUBLIC_KEY) hold public keys. The following table defines the attributes common to all public keys, in addition to the common attributes defined for this object class:
Table 24, Common Public Key Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT³</td>
<td>Byte array</td>
<td>DER-encoding of the key subject name (default empty)</td>
</tr>
<tr>
<td>CKA_ENCRYPT³</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports encryption</td>
</tr>
<tr>
<td>CKA_VERIFY³</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports verification where the signature is an appendix to the data³</td>
</tr>
<tr>
<td>CKA_VERIFY_RECOVER³</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports verification where the data is recovered from the signature³</td>
</tr>
<tr>
<td>CKA_WRAP³</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys³)</td>
</tr>
<tr>
<td>CKA_TRUSTED¹⁰</td>
<td>CK_BBOOL</td>
<td>The key can be trusted for the application that it was created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.</td>
</tr>
<tr>
<td>CKA_WRAP_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
<tr>
<td>CKA_PUBLIC_KEY_INFO</td>
<td>Byte array</td>
<td>DER-encoding of the SubjectPublicKeyInfo for this public key. (MAY be empty, DEFAULT derived from the underlying public key data)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

To map between ISO/IEC 9594-8 (X.509) keyUsage flags for public keys and the PKCS #11 attributes for public keys, use the following table.
Table 25, Mapping of X.509 key usage flags to Cryptoki attributes for public keys

<table>
<thead>
<tr>
<th>Key usage flags for public keys in X.509 public key certificates</th>
<th>Corresponding cryptoki attributes for public keys.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataEncipherment</td>
<td>CKA_ENCRYPT</td>
</tr>
<tr>
<td>digitalSignature, keyCertSign, cRLSign</td>
<td>CKA_VERIFY</td>
</tr>
<tr>
<td>digitalSignature, keyCertSign, cRLSign</td>
<td>CKA_VERIFY_RECOVER</td>
</tr>
<tr>
<td>keyAgreement</td>
<td>CKA_DERIVE</td>
</tr>
<tr>
<td>keyEncipherment</td>
<td>CKA_WRAP</td>
</tr>
<tr>
<td>nonRepudiation</td>
<td>CKA_VERIFY</td>
</tr>
<tr>
<td>nonRepudiation</td>
<td>CKA_VERIFY_RECOVER</td>
</tr>
</tbody>
</table>

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

\[
\text{SubjectPublicKeyInfo} ::= \text{SEQUENCE} \{ \\
\text{algorithm} \text{ AlgorithmIdentifier,} \\
\text{subjectPublicKey} \text{ BIT-STRING } \}
\]

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

4.9 Private key objects

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

Table 26, Common Private Key Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT^8</td>
<td>Byte array</td>
<td>DER-encoding of certificate subject name (default empty)</td>
</tr>
<tr>
<td>CKA_SENSITIVE^8,11</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key is sensitive^9</td>
</tr>
<tr>
<td>CKA_DECRYPT^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports decryption^9</td>
</tr>
<tr>
<td>CKA_SIGN^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports signatures where the signature is an appendix to the data^9</td>
</tr>
<tr>
<td>CKA_SIGN_RECOVER^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports signatures where the data can be recovered from the signature^9</td>
</tr>
<tr>
<td>CKA_UNWRAP^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)^9</td>
</tr>
<tr>
<td>CKA_EXTRACTABLE^8,12</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key is extractable and can be wrapped^9</td>
</tr>
<tr>
<td>CKA_ALWAYS_SENSITIVE^2,4,6</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key has always had the CKA_SENSITIVE attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_NEVER_EXTRACTABLE^2,4,6</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key has never had the CKA_EXTRACTABLE attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_WRAP_WITH_TRUSTED^11</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CKA_UNWRAP_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>Default is CK_FALSE. For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
<tr>
<td>CKA_ALWAYS_AUTHENTICATE</td>
<td>CK_BBOOL</td>
<td>If CK_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK_FALSE.</td>
</tr>
<tr>
<td>CKA_PUBLIC_KEY_INFO</td>
<td>Byte Array</td>
<td>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)</td>
</tr>
<tr>
<td>CKA_DERIVE_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>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 ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

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

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

Re-authentication occurs by calling C_Login with userType set to CKU_CONTEXT_SPECIFIC immediately after a cryptographic operation using the key has been initiated (e.g. after C_SignInit). In this call, the actual user type is implicitly given by the usage requirements of the active key. If C_Login returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g.
by C_Sign, C_SignFinal,...). A return value CKR_PIN_INCORRECT from C_Login means that the user
was denied permission to use the key and continuing the cryptographic operation will result in a behavior
as if C_Login had not been called. In both of these cases the session state will remain the same,
however repeated failed re-authentication attempts may cause the PIN to be locked. C_Login returns in
this case CKR_PIN_LOCKED and this also logs the user out from the token. Failing or omitting to re-
authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_TRUE will result in
CKR_USER_NOT_LOGGED_IN to be returned from calls using the key. C_Login will return
CKR_OPERATION_NOT_INITIALIZED, but the active cryptographic operation will not be affected, if an
attempt is made to re-authenticate when CKA_ALWAYS_AUTHENTICATE is set to CK_FALSE.
The CKA_PUBLIC_KEY_INFO attribute represents the public key associated with this private key. The
data it represents may either be stored as part of the private key data, or regenerated as needed from the
private key.
If this attribute is supplied as part of a template for C_CreateObject, C_CopyObject or
C_SetAttributeValue for a private key, the token MUST verify correspondence between the private key
data and the public key data as supplied in CKA_PUBLIC_KEY_INFO. This can be done either by
deriving a public key from the private key and comparing the values, or by doing a sign and verify
operation. If there is a mismatch, the command SHALL return CKR_ATTRIBUTE_VALUE_INVALID. A
token MAY choose not to support the CKA_PUBLIC_KEY_INFO attribute for commands which create
new private keys. If it does not support the attribute, the command SHALL return
CKR_ATTRIBUTE_TYPE_INVALID.
As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public
key information. In particular, the RSA private key description has been modified in PKCS #11 V2.40 to
add the CKA_PUBLIC_EXPONENT to the list of attributes required for an RSA private key. All other
private key types described in this specification contain sufficient information to recover the associated
public key.

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SENSITIVE^8,11</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if object is sensitive (default CK_FALSE)</td>
</tr>
<tr>
<td>CKA_ENCRYPT^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports encryption^9</td>
</tr>
<tr>
<td>CKA_DECRYPT^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports decryption^9</td>
</tr>
<tr>
<td>CKA_SIGN^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports signatures (i.e., authentication codes) where the signature is an appendix to the data^9</td>
</tr>
<tr>
<td>CKA_VERIFY^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports verification (i.e., of authentication codes) where the signature is an appendix to the data^9</td>
</tr>
<tr>
<td>CKA_WRAP^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys)^9</td>
</tr>
<tr>
<td>CKA_UNWRAP^8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)^9</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data type</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CKA_EXTRACTABLE(^6,12)</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key is extractable and can be wrapped (^9)</td>
</tr>
<tr>
<td>CKA_ALWAYS_SENSITIVE(^2,4,6)</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key has always had the CKA_SENSITIVE attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_NEVER_EXTRACTABLE(^2,4,6)</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key has never had the CKA_EXTRACTABLE attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_CHECK_VALUE</td>
<td>Byte array</td>
<td>Key checksum</td>
</tr>
<tr>
<td>CKA_WRAP_WITH_TRUSTED(^11)</td>
<td>CK_BBOOL</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_TRUSTED(^10)</td>
<td>CK_BBOOL</td>
<td>The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.</td>
</tr>
<tr>
<td>CKA_WRAP_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
<tr>
<td>CKA_UNWRAP_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
<tr>
<td>A_DERIVE_TEMPLATE</td>
<td>CK_ATTRIBUTE_PTR</td>
<td>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 ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.
If the CKA_SENSITIVE attribute is CK_TRUE, or if the CKA_EXTRACTABLE attribute is CK_FALSE,
then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which
attributes these are is specified for each type of secret key in the attribute table in the section describing
that type of key.

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

Properties:

1. For two keys that are cryptographically identical the value of this attribute should be identical.

2. CKA_CHECK_VALUE should not be usable to obtain any part of the key value.

3. Non-uniqueness. Two different keys can have the same CKA_CHECK_VALUE. This is unlikely
   (the probability can easily be calculated) but possible.

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

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

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

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

4.11 Domain parameter objects

4.11.1 Definitions

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

4.11.2 Overview

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

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

Domain parameter objects (object class CKO_DOMAIN_PARAMETERS) hold public domain parameters.
The following table defines the attributes common to domain parameter objects in addition to the common
attributes defined for this object class:

<p>| Table 28, Common Domain Parameter Attributes |</p>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_KEY_TYPE¹</td>
<td>CK_KEY_TYPE</td>
<td>Type of key the domain parameters can be used to generate.</td>
</tr>
<tr>
<td>CKA_LOCAL²,⁴</td>
<td>CK_BBOOL</td>
<td>CK_TRUE only if domain parameters were either</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• generated locally (i.e., on the token) with a C_GenerateKey</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• created with a C_CopyObject call as a copy of domain parameters which had</td>
</tr>
<tr>
<td></td>
<td></td>
<td>its CKA_LOCAL attribute set to CK_TRUE</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

4.12 Mechanism objects

4.12.1 Definitions

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

4.12.2 Overview

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

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_MECHANISM_TYPE</td>
<td>CK_MECHANISM_TYPE</td>
<td>The type of mechanism object</td>
</tr>
</tbody>
</table>

The CKA_MECHANISM_TYPE attribute may not be set.

4.13 Profile objects

4.13.1 Definitions

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

4.13.2 Overview

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

Table 30, Profile Object Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PROFILE_ID</td>
<td>CK_PROFILE_ID</td>
<td>ID of the supported profile.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General purpose functions</td>
<td>C_Initialize</td>
<td>initializes Cryptoki</td>
</tr>
<tr>
<td></td>
<td>C_Finalize</td>
<td>clean up miscellaneous Cryptoki-associated resources</td>
</tr>
<tr>
<td></td>
<td>C_GetInfo</td>
<td>obtains general information about Cryptoki</td>
</tr>
<tr>
<td></td>
<td>C_GetFunctionList</td>
<td>obtains entry points of Cryptoki library functions</td>
</tr>
<tr>
<td></td>
<td>C_GetInterfaceList</td>
<td>obtains list of interfaces supported by Cryptoki library</td>
</tr>
<tr>
<td></td>
<td>C_GetInterface</td>
<td>obtains interface specific entry points to Cryptoki library functions</td>
</tr>
<tr>
<td>Slot and token management functions</td>
<td>C_GetSlotList</td>
<td>obtains a list of slots in the system</td>
</tr>
<tr>
<td></td>
<td>C_GetSlotInfo</td>
<td>obtains information about a particular slot</td>
</tr>
<tr>
<td></td>
<td>C_GetTokenInfo</td>
<td>obtains information about a particular token</td>
</tr>
<tr>
<td></td>
<td>C_WaitForSlotEvent</td>
<td>waits for a slot event (token insertion, removal, etc.) to occur</td>
</tr>
<tr>
<td></td>
<td>C_GetMechanismList</td>
<td>obtains a list of mechanisms supported by a token</td>
</tr>
<tr>
<td></td>
<td>C_GetMechanismInfo</td>
<td>obtains information about a particular mechanism</td>
</tr>
<tr>
<td></td>
<td>C_InitToken</td>
<td>initializes a token</td>
</tr>
<tr>
<td></td>
<td>C_InitPIN</td>
<td>initializes the normal user’s PIN</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>C_SetPIN</td>
<td>modifies the PIN of the current user</td>
</tr>
<tr>
<td>Session management functions</td>
<td>C_OpenSession</td>
<td>opens a connection between an application and a particular token or sets up an application callback for token insertion</td>
</tr>
<tr>
<td></td>
<td>C_CloseSession</td>
<td>closes a session</td>
</tr>
<tr>
<td></td>
<td>C_CloseAllSessions</td>
<td>closes all sessions with a token</td>
</tr>
<tr>
<td></td>
<td>C_GetSessionInfo</td>
<td>obtains information about the session</td>
</tr>
<tr>
<td></td>
<td>C_SessionCancel</td>
<td>terminates active session based operations</td>
</tr>
<tr>
<td></td>
<td>C_GetOperationState</td>
<td>obtains the cryptographic operations state of a session</td>
</tr>
<tr>
<td></td>
<td>C_SetOperationState</td>
<td>sets the cryptographic operations state of a session</td>
</tr>
<tr>
<td></td>
<td>C_Login</td>
<td>logs into a token</td>
</tr>
<tr>
<td></td>
<td>C_LoginUser</td>
<td>logs into a token with explicit user name</td>
</tr>
<tr>
<td></td>
<td>C_Logout</td>
<td>logs out from a token</td>
</tr>
<tr>
<td>Object management functions</td>
<td>C_CreateObject</td>
<td>creates an object</td>
</tr>
<tr>
<td></td>
<td>C_CopyObject</td>
<td>creates a copy of an object</td>
</tr>
<tr>
<td></td>
<td>C_DestroyObject</td>
<td>destroys an object</td>
</tr>
<tr>
<td></td>
<td>C_GetObjectSize</td>
<td>obtains the size of an object in bytes</td>
</tr>
<tr>
<td></td>
<td>C_GetAttributeValue</td>
<td>obtains an attribute value of an object</td>
</tr>
<tr>
<td></td>
<td>C_SetAttributeValue</td>
<td>modifies an attribute value of an object</td>
</tr>
<tr>
<td></td>
<td>C_FindObjectsInit</td>
<td>initializes an object search operation</td>
</tr>
<tr>
<td></td>
<td>C_FindObjects</td>
<td>continues an object search operation</td>
</tr>
<tr>
<td></td>
<td>C_FindObjectsFinal</td>
<td>finishes an object search operation</td>
</tr>
<tr>
<td>Encryption functions</td>
<td>C_EncryptInit</td>
<td>initializes an encryption operation</td>
</tr>
<tr>
<td></td>
<td>C_Encrypt</td>
<td>encrypts single-part data</td>
</tr>
<tr>
<td></td>
<td>C_EncryptUpdate</td>
<td>continues a multiple-part encryption operation</td>
</tr>
<tr>
<td></td>
<td>C_EncryptFinal</td>
<td>finishes a multiple-part encryption operation</td>
</tr>
<tr>
<td>Message-based Encryption Functions</td>
<td>C_MessageEncryptInit</td>
<td>initializes a message-based encryption process</td>
</tr>
<tr>
<td></td>
<td>C_EncryptMessage</td>
<td>encrypts a single-part message</td>
</tr>
<tr>
<td></td>
<td>C_EncryptMessageBegin</td>
<td>begins a multiple-part message encryption operation</td>
</tr>
<tr>
<td></td>
<td>C_EncryptMessageNext</td>
<td>continues or finishes a multiple-part message encryption operation</td>
</tr>
<tr>
<td></td>
<td>C_MessageEncryptFinal</td>
<td>finishes a message-based encryption process</td>
</tr>
<tr>
<td>Decryption Functions</td>
<td>C_DecryptInit</td>
<td>initializes a decryption operation</td>
</tr>
<tr>
<td></td>
<td>C_Decrypt</td>
<td>decrypts single-part encrypted data</td>
</tr>
<tr>
<td></td>
<td>C_DecryptUpdate</td>
<td>continues a multiple-part decryption operation</td>
</tr>
<tr>
<td></td>
<td>C_DecryptFinal</td>
<td>finishes a multiple-part decryption operation</td>
</tr>
<tr>
<td>Message-based Decryption Functions</td>
<td>C_MessageDecryptInit</td>
<td>initializes a message decryption operation</td>
</tr>
<tr>
<td></td>
<td>C_DecryptMessage</td>
<td>decrypts single-part data</td>
</tr>
<tr>
<td></td>
<td>C_DecryptMessageBegin</td>
<td>starts a multiple-part message decryption operation</td>
</tr>
<tr>
<td></td>
<td>C_DecryptMessageNext</td>
<td>continues and finishes a multiple-part message decryption operation</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>Message Digesting Functions</td>
<td>C_MessageDecryptFinal</td>
<td>finishes a message decryption operation</td>
</tr>
<tr>
<td></td>
<td>C_DigestInit</td>
<td>initializes a message-digesting operation</td>
</tr>
<tr>
<td></td>
<td>C_Digest</td>
<td>digests single-part data</td>
</tr>
<tr>
<td></td>
<td>C_DigestUpdate</td>
<td>continues a multiple-part digesting operation</td>
</tr>
<tr>
<td></td>
<td>C_DigestKey</td>
<td>digests a key</td>
</tr>
<tr>
<td></td>
<td>C_DigestFinal</td>
<td>finishes a multiple-part digesting operation</td>
</tr>
<tr>
<td>Signing and MACing functions</td>
<td>C_SignInit</td>
<td>initializes a signature operation</td>
</tr>
<tr>
<td></td>
<td>C_Sign</td>
<td>signs single-part data</td>
</tr>
<tr>
<td></td>
<td>C_SignUpdate</td>
<td>continues a multiple-part signature operation</td>
</tr>
<tr>
<td></td>
<td>C_SignFinal</td>
<td>finishes a multiple-part signature operation</td>
</tr>
<tr>
<td></td>
<td>C_SignRecoverInit</td>
<td>initializes a signature operation, where the data can be recovered from the signature</td>
</tr>
<tr>
<td></td>
<td>C_SignRecover</td>
<td>signs single-part data, where the data can be recovered from the signature</td>
</tr>
<tr>
<td>Message-based Signature functions</td>
<td>C_MessageSignInit</td>
<td>initializes a message signature operation</td>
</tr>
<tr>
<td></td>
<td>C_SignMessageBegin</td>
<td>starts a multiple-part message signature operation</td>
</tr>
<tr>
<td></td>
<td>C_SignMessageNext</td>
<td>continues and finishes a multiple-part message signature operation</td>
</tr>
<tr>
<td></td>
<td>C_MessageSignFinal</td>
<td>finishes a message signature operation</td>
</tr>
<tr>
<td>Functions for verifying signatures and MACs</td>
<td>C_VerifyInit</td>
<td>initializes a verification operation</td>
</tr>
<tr>
<td></td>
<td>C_Verify</td>
<td>verifies a signature on single-part data</td>
</tr>
<tr>
<td></td>
<td>C_VerifyUpdate</td>
<td>continues a multiple-part verification operation</td>
</tr>
<tr>
<td></td>
<td>C_VerifyFinal</td>
<td>finishes a multiple-part verification operation</td>
</tr>
<tr>
<td></td>
<td>C_VerifyRecoverInit</td>
<td>initializes a verification operation where the data is recovered from the signature</td>
</tr>
<tr>
<td></td>
<td>C_VerifyRecover</td>
<td>verifies a signature on single-part data, where the data is recovered from the signature</td>
</tr>
<tr>
<td>Message-based Functions for verifying signatures and MACs</td>
<td>C_MessageVerifyInit</td>
<td>initializes a message verification operation</td>
</tr>
<tr>
<td></td>
<td>C_VerifyMessageBegin</td>
<td>starts a multiple-part message verification operation</td>
</tr>
<tr>
<td></td>
<td>C_VerifyMessageNext</td>
<td>continues and finishes a multiple-part message verification operation</td>
</tr>
<tr>
<td></td>
<td>C_MessageVerifyFinal</td>
<td>finishes a message verification operation</td>
</tr>
<tr>
<td>Dual-purpose cryptographic functions</td>
<td>C_DigestEncryptUpdate</td>
<td>continues simultaneous multiple-part digesting and encryption operations</td>
</tr>
<tr>
<td></td>
<td>C_DecryptDigestUpdate</td>
<td>continues simultaneous multiple-part decryption and digesting operations</td>
</tr>
<tr>
<td></td>
<td>C_SignEncryptUpdate</td>
<td>continues simultaneous multiple-part signature and encryption operations</td>
</tr>
<tr>
<td></td>
<td>C_DecryptVerifyUpdate</td>
<td>continues simultaneous multiple-part decryption and verification operations</td>
</tr>
<tr>
<td>Key management</td>
<td>C.GenerateKey</td>
<td>generates a secret key</td>
</tr>
<tr>
<td></td>
<td>C.GenerateKeyPair</td>
<td>generates a public-key/private-key pair</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>functions</td>
<td>C_WrapKey</td>
<td>wraps (encrypts) a key</td>
</tr>
<tr>
<td></td>
<td>C_UnwrapKey</td>
<td>unwraps (decrypts) a key</td>
</tr>
<tr>
<td></td>
<td>C_DeriveKey</td>
<td>derives a key from a base key</td>
</tr>
<tr>
<td>Random number generation</td>
<td>C_SeedRandom</td>
<td>mixes in additional seed material to the random number generator</td>
</tr>
<tr>
<td>functions</td>
<td>C_GenerateRandom</td>
<td>generates random data</td>
</tr>
<tr>
<td>Parallel function management</td>
<td>C_GetFunctionStatus</td>
<td>legacy function which always returns CKR_FUNCTION_NOT_PARALLEL</td>
</tr>
<tr>
<td>functions</td>
<td>C_CancelFunction</td>
<td>legacy function which always returns CKR_FUNCTION_NOT_PARALLEL</td>
</tr>
<tr>
<td>Callback function</td>
<td></td>
<td>application-supplied function to process notifications from Cryptoki</td>
</tr>
</tbody>
</table>

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

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

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

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function MUST have a "stub" in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED. The function's entry in the library's CK_FUNCTION_LIST structure (as obtained by C_GetFunctionList) should point to this stub function (see Section 3.6).

### 5.1 Function return values

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

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

See Section 5.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.
5.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

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

- **CKR_HOST_MEMORY**: The computer that the Cryptoki library is running on has insufficient memory to perform the requested function.

- **CKR_FUNCTION_FAILED**: The requested function could not be performed, but detailed information about why not is not available in this error return. If the failed function uses a session, it is possible that the **CK_SESSION_INFO** structure that can be obtained by calling **C_GetSessionInfo** will hold useful information about what happened in its **ulDeviceError** field. In any event, although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when **CKR_GENERAL_ERROR** is returned. Depending on what the root cause of the error actually was, it is possible that an attempt to make the exact same function call again would succeed.

- **CKR_OK**: The function executed successfully. Technically, CKR_OK is not quite a “universal” return value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section 5.20) cannot return CKR_OK.

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

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

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

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

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

- **CKR_SESSION_CLOSED**: The session was closed during the execution of the function. Note that, as stated in [PKCS11-UG], the behavior of Cryptoki is undefined if multiple threads of an application attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no guarantee that a function invocation could ever return the value **CKR_SESSION_CLOSED**. An example of multiple threads accessing a common session simultaneously is where one thread is using a session when another thread closes that same session.

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

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

5.1.3 Cryptoki function return values for functions that use a token

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

- **CKR_DEVICE_MEMORY**: The token does not have sufficient memory to perform the requested function.
• CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can be returned by more than just the functions mentioned above; in particular, it is possible for C_GetSlotInfo to return CKRDEVICE_ERROR.

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

• CKRDEVICE_REMOVED: The token was removed from its slot during the execution of the function.

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

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

5.1.4 Special return value for application-supplied callbacks

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

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

5.1.5 Special return values for mutex-handling functions

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

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

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

5.1.6 All other Cryptoki function return values

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

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

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

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

• CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or un-extractable.
• **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 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_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.
2. The library cannot function properly without being able to spawn new threads in the above fashion.

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

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

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

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

- **CKR_PIN_EXPIRED**: The specified PIN has expired, and the requested operation cannot be carried out unless `C_SetPIN` is called to change the PIN value. Whether or not the normal user's PIN on a token ever expires varies from token to token.

- **CKR_PIN_INCORRECT**: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the token. More generally-- when authentication to the token involves something other than a PIN-- the attempt to authenticate the user has failed.

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

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

- **CKR_PIN_LOCKED**: The specified PIN is “locked”, and cannot be used. That is, because some particular number of failed authentication attempts has been reached, the token is unwilling to permit further attempts at authentication. Depending on the token, the specified PIN may or may not remain locked indefinitely.

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

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

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

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

- **CKR_SAVED_STATE_INVALID**: This value can only be returned by `C_SetOperationState`. It indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be restored to the specified session.
• CKR_SESSION_COUNT: This value can only be returned by C_OpenSession. It indicates that the attempt to open a session failed, either because the token has too many sessions already open, or because the token has too many read/write sessions already open.

• CKR_SESSION_EXISTS: This value can only be returned by C_InitToken. It indicates that a session with the token is already open, and so the token cannot be initialized.

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

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

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

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

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

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

• CKR_SLOT_ID_INVALID: The specified slot ID is not valid.

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

• CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by C_UnwrapKey. It indicates that the key handle specified to be used to unwrap another key is not valid.

• CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by C_UnwrapKey. It indicates that although the requested unwrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key’s size is outside the range of key sizes that it can handle.

• CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by C_UnwrapKey. It indicates that the type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.

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

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

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

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

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

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

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

- **CKR_WRAPPED_KEY_LEN_RANGE**: This value can only be returned by **C_UnwrapKey**. It
  indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
  This return value has higher priority than **CKR_WRAPPED_KEY_INVALID**.

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

- **CKR_WRAPPING_KEY_SIZE_RANGE**: This value can only be returned by **C_WrapKey**. It indicates
  that although the requested wrapping operation could in principle be carried out, this Cryptoki library
  (or the token) is unable to actually do it because the supplied wrapping key’s size is outside the range
  of key sizes that it can handle.

- **CKR_WRAPPING_KEY_TYPE_INCONSISTENT**: This value can only be returned by **C_WrapKey**. It
  indicates that the type of the key specified to wrap another key is not consistent with the mechanism
  specified for wrapping.

- **CKR_OPERATION_CANCEL_FAILED**: This value can only be returned by **C_SessionCancel**. It
  means that one or more of the requested operations could not be cancelled for implementation or
  vendor-specific reasons.

5.1.7 More on relative priorities of Cryptoki errors

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

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

1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction between a token being removed before a function invocation and a token being removed during a function invocation.

2. As mentioned in Section 5.1.2, an application should never count on getting a CKR_SESSION_CLOSED error.

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

4. Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and CKR_ENCRYPTED_DATA_LEN_RANGE, and between CKR_WRAPPED_KEY_INVALID and CKR_WRAPPED_KEY_LEN_RANGE, can be subtle, and it may be best to treat these return values equivalently.

5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which of CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, or CKR_TEMPLATE_INCONSISTENT to return. When possible, it is recommended that application developers be generous in their interpretations of these error codes.

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

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

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

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

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

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

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

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

5.3 Disclaimer concerning sample code

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

5.4 General-purpose functions

Cryptoki provides the following general-purpose functions:

5.4.1 C_Initialize

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

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

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

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

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

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

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

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

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

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_CANT_LOCK, CKR_CRYPTOKI_ALREADY_INITIALIZED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKR_HOST_MEMORY, CKR_NEED_TO_CREATE_THREADS, CKR_OK.

Example: see C_GetInfo.

### 5.4.2 C_Finalize

```c
CK_DECLARE_FUNCTION(CK_RV, C_Finalize)(
    CK_VOID_PTR pReserved
);
```

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

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

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

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

Example: see C_GetInfo.

### 5.4.3 C_GetInfo

```c
CK_DECLARE_FUNCTION(CK_RV, C_GetInfo)(
    CK_INFO_PTR pInfo
);
```

C_GetInfo returns general information about Cryptoki. pInfo points to the location that receives the information.
Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

Example:

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

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

rv = C_Initialize((CK_VOID_PTR)&InitArgs);
assert(rv == CKR_OK);
rv = C_GetInfo(&info);
assert(rv == CKR_OK);
if(info.cryptokiVersion.major == 2) {
    /* Do lots of interesting cryptographic things with the token */
    .
    .
    }
rv = C_Finalize(NULL_PTR);
assert(rv == CKR_OK);
```

5.4.4 C_GetFunctionList

```c
CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList)(
    CK_FUNCTION_LIST_PTR_PTR ppFunctionList
);
```

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

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

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

Example:

```c
CK_FUNCTION_LIST_PTR pFunctionList;
```
CK_C_Initialize pC_Initialize;
CK_RV rv;
/* It’s OK to call C_GetFunctionList before calling C_INITIALIZE */
rv = C_GetFunctionList(&pFunctionList);
assert(rv == CKR_OK);
pC_Initialize = pFunctionList -> C_Initialize;
/* Call the C_INITIALIZE function in the library */
rv = (*pC_Initialize)(NULL_PTR);

5.4.5 C_GetInterfaceList

CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList)(
    CK_INTERFACE_PTR      pInterfaceList,
    CK_ULONG_PTR          pulCount
);

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

There are two ways for an application to call C_GetInterfaceList:
1. If pInterfaceList is NULL_PTR, then all that C_GetInterfaceList does is return (in *pulCount) the
   number of interfaces, without actually returning a list of interfaces. The contents of *pulCount on
   entry to C_GetInterfaceList has no meaning in this case, and the call returns the value CKR_OK.
2. If pInterfaceList is not NULL_PTR, then *pulCount MUST contain the size (in terms of
   CK_INTERFACE elements) of the buffer pointed to by pInterfaceList. If that buffer is large enough to
   hold the list of interfaces, then the list is returned in it, and CKR_OK is returned. If not, then the call
   to C_GetInterfaceList returns the value CKR_BUFFER_TOO_SMALL. In either case, the value
   *pulCount is set to hold the number of interfaces.

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

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

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

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

Example:

CK_ULONG ulCount=0;
CK_INTERFACE_PTR interfaceList=NULL;
CK_RV rv;
int I;
/* get number of interfaces */
rv = C_GetInterfaceList(NULL,&ulCount);
if (rv == CKR_OK) {
    /* get copy of interfaces */
    interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
    rv = C_GetInterfaceList(interfaceList,&ulCount);
    for(i=0;i<ulCount;i++) {
        printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
               interfaceList[i].pInterfaceName,
               ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
               ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
               interfaceList[i].pFunctionList,
               interfaceList[i].flags);
    }
}

5.4.6 C_GetInterface

CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) {
    CK_UTF8CHAR_PTR       pInterfaceName,
    CK_VERSION_PTR        pVersion,
    CK_INTERFACE_PTR_PTR  ppInterface,
    CK_FLAGS              flags
};

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

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

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.
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.
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.

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

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

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

Example:

CK_INTERFACE_PTR interface;
CK_RV rv;
CK_VERSION version;
CK_FLAGS flags=CKF_INTERFACE_FORK_SAFE;

/* get default interface */
rv = C_GetInterface(NULL,NULL,&interface,flags);
if (rv == CKR_OK) {
    printf("interface %s version %d.%d funcs %p flags 0x%lu
",
            interface->pInterfaceName,
            ((CK_VERSION *)interface->pFunctionList)->major,
            ((CK_VERSION *)interface->pFunctionList)->minor,
            interface->pFunctionList,
            interface->flags);
}

/* get default standard interface */
rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
if (rv == CKR_OK) {
    printf("interface %s version %d.%d funcs %p flags 0x%lu
",
            interface->pInterfaceName,
            ((CK_VERSION *)interface->pFunctionList)->major,
            ((CK_VERSION *)interface->pFunctionList)->minor,
            interface->pFunctionList,
            interface->flags);
}

/* get specific standard version interface */
version.major=3;
version.minor=0;
rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
if (rv == CKR_OK) {
    CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
    /* ... use the new functions */
    pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
                       pUsername,ulUsernameLen);
}

/* get specific vendor version interface */
version.major=1;
version.minor=0;
rv = C_GetInterface((CK_UTF8CHAR_PTR)"Vendor VendorName",&version,&interface,flags);
if (rv == CKR_OK) {

5.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management:

5.5.1 C_GetSlotList

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

There are two ways for an application to call C_GetSlotList:

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

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

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

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

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

Example:

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

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

/* Get list of all slots with a token present */
pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
ulSlotWithTokenCount = 0;
while (1) {
    rv = C_GetSlotList(
        CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
    if (rv != CKR_BUFFER_TOO_SMALL)
        break;
    pSlotWithTokenList = realloc(
        pSlotWithTokenList,
        ulSlotWithTokenList*sizeof(CK_SLOT_ID));
    }
    if (rv == CK_OK) {
        /* Now use that list of all slots with a token present */
    .
    .
    .
    }
    free(pSlotWithTokenList);

5.5.2 C_GetSlotInfo

CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo)(
    CK_SLOT_ID slotID,
    CK_SLOT_INFO_PTR pInfo
);
C_GetSlotInfo obtains information about a particular slot in the system. slotID is the ID of the slot; pInfo points to the location that receives the slot information.

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

Example: see C_GetTokenInfo.

5.5.3 C_GetTokenInfo

```c
CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo)(
    CK_SLOT_ID slotID,
    CK_TOKEN_INFO_PTR pInfo
);
```

C_GetTokenInfo obtains information about a particular token in the system. slotID is the ID of the token’s slot; pInfo points to the location that receives the token information.

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

Example:

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

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

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

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

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

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

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

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

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

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

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

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

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

Example:

```c
CK_FLAGS flags = 0;
CK_SLOT_ID slotID;
CK_SLOT_INFO slotInfo;
CK_RV rv;

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

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

5.5.5 C_GetMechanismList

CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList)(
    CK_SLOT_ID slotID,
    CK_MECHANISM_TYPE_PTR pMechanismList,
    CK_ULONG_PTR pulCount
);

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

There are two ways for an application to call C_GetMechanismList:

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

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

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

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

Example:

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

rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);
if ((rv == CKR_OK) && (ulCount > 0)) {
    pMechanismList = (CK_MECHANISM_TYPE_PTR)malloc(ulCount*sizeof(CK_MECHANISM_TYPE));
    rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);
    if (rv == CKR_OK) {
        
        
    }
5.5.6 C_GetMechanismInfo

```c
CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo)(
    CK_SLOT_ID slotID,
    CK_MECHANISM_TYPE type,
    CK_MECHANISM_INFO_PTR pInfo
);
```

C_GetMechanismInfo obtains information about a particular mechanism possibly supported by a token. `slotID` is the ID of the token’s slot; `type` is the type of mechanism; `pInfo` points to the location that receives the mechanism information.

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

Example:

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

/* Get information about the CKM_MD2 mechanism for this token */
rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
```

5.5.7 C_InitToken

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

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

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

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

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

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

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

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

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

Example:

```c
CK_Slot_ID slotID;
CK_UTF8CHAR pin[] = {“MyPIN”};
CK_UTF8CHAR label[32];
CK_RV rv;

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

```
5.5.8 C_InitPIN

```c
CK_DECLARE_FUNCTION(CK_RV, C_InitPIN)(
    CK_SESSION_HANDLE hSession,
    CK_UTF8CHAR_PTR pPin,
    CKULONG ulPinLen
);
```

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

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

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR/general ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY, CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN, CKR_ARGUMENTS_BAD.

Example:

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

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

5.5.9 C_SetPIN

```c
CK_DECLARE_FUNCTION(CK_RV, C_SetPIN)(
    CK_SESSION_HANDLE hSession,
    CK_UTF8CHAR_PTR pOldPin,
    CK_ULONGLONG ulOldLen,
    CK_UTF8CHAR_PTR pNewPin,
    CK_ULONGLONG ulNewLen
);
```
**C_SetPIN** modifies the PIN of the user that is currently logged in, or the CKU_USER PIN if the session is not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

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

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

**Example:**

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

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

### 5.6 Session management functions

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

1. Select a token.
2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a shared login state, **C_Login** only needs to be called for one of the sessions.
4. Perform cryptographic operations using the sessions with the token.
5. Call **C_CloseSession** once for each session that the application has with the token, or call **C_CloseAllSessions** to close all the application’s sessions simultaneously.
As has been observed, an application may have concurrent sessions with more than one token. It is also possible for a token to have concurrent sessions with more than one application.

Cryptoki provides the following functions for session management:

### 5.6.1 C_OpenSession

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

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

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

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

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

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

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

Return values: *CKR_CRYPTOKI_NOT_INITIALIZED*, *CKR_DEVICE_ERROR*, *CKR_DEVICE_MEMORY*, *CKR_DEVICE_REMOVED*, *CKR_FUNCTION_FAILED*, *CKR/general_error*, *CKR_HOST_MEMORY*, *CKR_OK*, *CKR_SESSION_COUNT*, *CKR_SESSION_PARALLEL_NOT_SUPPORTED*, *CKR_SESSION_READ_WRITE_SO_EXISTS*, *CKR_SLOT_ID_INVALID*, *CKR_TOKEN_NOT_PRESENT*, *CKR_TOKEN_NOT_RECOGNIZED*, *CKR_TOKEN_READ_WRITE_PROTECTED*, *CKR_ARGUMENTS_BAD*.

Example: see *C_CloseSession*.

### 5.6.2 C_CloseSession

```c
CK_DECLARE_FUNCTION(CK_RV, C_CloseSession)(
    CK_SESSION_HANDLE hSession
);
```

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

When a session is closed, all session objects created by the session are destroyed automatically, even if the application has other sessions “using” the objects (see [*PKCS11-UG*] for further details).
If this function is successful and it closes the last session between the application and the token, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

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

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

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

Example:

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

application = 17;
MyNotify = &EncryptionSessionCallback;
rv = C_OpenSession(
  slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,
  (CK_VOID_PTR) &application, MyNotify,
  &hSession);
if (rv == CKR_OK) {
  
  C_CloseSession(hSession);
}
```

5.6.3 C_CloseAllSessions

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

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

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

Depending on the token, when the last open session any application has with the token is closed, the token may be “ejected” from its reader (if this capability exists).
Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

Example:

```c
CK_SLOT_ID slotID;
CK_RV rv;
rv = C_CloseAllSessions(slotID);
```

### 5.6.4 C_GetSessionInfo

```c
def CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo)(
    CK_SESSION_HANDLE hSession,
    CK_SESSION_INFO_PTR pInfo
) { }
```

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

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

Example:

```c
CK_SESSION_HANDLE hSession;
CK_SESSION_INFO info;
CK_RV rv;
rv = C_GetSessionInfo(hSession, &info);
if (rv == CKR_OK) {
    if (info.state == CKS_RW_USER_FUNCTIONS) {
        //
        //
    }
    //
    //
}
```

### 5.6.5 C_SessionCancel

```c
def CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel)(
    CK_SESSION_HANDLE hSession
) { }
```

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

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

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

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

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

If C_SessionCancel is called from an application callback (see Section 5.21), no action will be taken by the library and CKR_FUNCTION_FAILED must be returned.

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_CANCEL_FAILED, CKR_TOKEN_NOT_PRESENT.

Example:

```c
CK_SESSION_HANDLE hSession;
CK_RV rv;

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

rv = C_SessionCancel (hSession, CKF_ENCRYPT);
if (rv != CKR_OK)
{
    ...
}

rv = C_EncryptInit(hSession, &mechanism, hKey);
if (rv != CKR_OK)
{
    ...
}
```
Below are modifications to existing API descriptions to allow an alternate method of cancelling individual operations. The additional text is highlighted.

### 5.6.6 C_GetOperationState

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

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

Although the saved state output by C_GetOperationState is not really produced by a “cryptographic mechanism”, C_GetOperationState nonetheless uses the convention described in Section 5.2 on producing output.

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

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

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

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

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

An attempt to save the cryptographic operations state of a session which is performing an appropriate cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain necessary state information and/or key information can’t leave the token, for example) should fail with the error CKR_STATE_UNSAVEABLE.
Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
CKR_STATE_UNSAVEABLE, CKR_ARGUMENTS_BAD.

Example: see C_SetOperationState.

5.6.7 C_SetOperationState

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

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

The state need not have been obtained from the same session (the “source session”) as it is being
restored to (the “destination session”). However, the source session and destination session should have
a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token.

There is also no guarantee that cryptographic operations state may be carried across logins, or across
different Cryptoki implementations.

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

Saved state obtained from calls to C_GetOperationState may or may not contain information about keys
in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing
encryption or decryption operation, and the key in use for the operation is not saved in the state, then it
 MUST be supplied to C_SetOperationState in the hEncryptionKey argument. If it is not, then
C_SetOperationState will fail and return the error CKR_KEY_NEEDED. If the key in use for the
operation is saved in the state, then it can be supplied in the hEncryptionKey argument, but this is not
required.

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

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

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

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

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

Return values: \texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY}, \texttt{CKR\_DEVICE\_REMOVED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR}, \texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_KEY\_CHANGED}, \texttt{CKR\_KEY\_NEEDED}, \texttt{CKR\_KEY\_NOT\_NEEDED}, \texttt{CKR\_OK}, \texttt{CKR\_SAVED\_STATE\_INVALID}, \texttt{CKR\_SESSION\_CLOSED}, \texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_ARGUMENTS\_BAD}.

Example:

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

```c
/* Initialize hash operation */
rv = C_DigestInit(hSession, &digestMechanism);
assert(rv == CKR\_OK);
```

```c
/* Start hashing */
rv = C_DigestUpdate(hSession, data1, sizeof(data1));
assert(rv == CKR\_OK);
```

```c
/* Find out how big the state might be */
rv = C_GetOperationState(hSession, NULL\_PTR, &ulStateLen);
assert(rv == CKR\_OK);
```

```c
/* Allocate some memory and then get the state */
pState = (CK\_BYTE\_PTR) malloc(ulStateLen);
rv = C_GetOperationState(hSession, pState, &ulStateLen);
```
/* Continue hashing */
rv = C_DigestUpdate(hSession, data2, sizeof(data2));
assert(rv == CKR_OK);

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

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

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

5.6.8 C_Login

CK_DECLARE_FUNCTION(CK_RV, C_Login)(
    CK_SESSION_HANDLE hSession,
    CK_USER_TYPE userType,
    CK_UTF8CHAR_PTR pPin,
    CK_ULONG ulPinLen
);

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

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

If the token has a “protected authentication path”, as indicated by the
CKF_PROTECTED_AUTHENTICATION_PATH flag in its CK_TOKEN_INFO being set, then that means
that there is some way for a user to be authenticated to the token without having to send a PIN through
the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
fingerprint-reading device, for example. To log into a token with a protected authentication path, the pPin
parameter to C_Login should be NULL_PTR. When C_Login returns, whatever authentication method
supported by the token will have been performed; a return value of CKR_OK means that the user was
successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
denied access.
If there are any active cryptographic or object finding operations in an application’s session, and then
C_Login is successfully executed by that application, it may or may not be the case that those operations
are still active. Therefore, before logging in, any active operations should be finished.

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

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

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

Example: see C_Logout.

5.6.9 C_LoginUser

CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) {
    CK_SESSION_HANDLE hSession,
    CK_USER_TYPE userType,
    CK_UTF8CHAR_PTR pPin,
    CK_ULONG ulPinLen,
    CK_UTF8CHAR_PTR pUsername,
    CK_ULONG ulUsernameLen
} ;

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

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

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

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

If the application calling C_LoginUser has a R/O session open with the token, then it will be unable to log
the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
code CKR_SESSION_READ_ONLY_EXISTS.
C_LoginUser may be called repeatedly, without intervening C_Logout calls, if (and only if) a key with the CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do cryptographic operation on this key. See further Section 4.9.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_ERROR, CKRDEVICE_MEMORY, CKRDEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKRHOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION CLOSED, CKR_SESSION_HANDLE INVALID, CKR_SESSION_READ_ONLY EXISTS, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

Example:

```c
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR userPin[] = {"MyPIN"};
CK_UTF8CHAR userName[] = {"MyUserName"};
CK_RV rv;

rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName, sizeof(userName)-1);
if (rv == CKR_OK) {
    rv = C_Logout(hSession);
    if (rv == CKR_OK) {
        
    }
}
```

5.6.10 C_Logout

C_Logout logs a user out from a token. hSession is the session’s handle.

Depending on the current user type, if the call succeeds, each of the application’s sessions will enter either the “R/W Public Session” state or the “R/O Public Session” state.

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_ERROR, CKRDEVICE_MEMORY, CKRDEVICE_REMOVED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKRHOST_MEMORY, CKR_OK, CKRSESSION_CLOSED, CKRSESSION_HANDLE INVALID, CKR_USER_NOT_LOGGED_IN.

Example:
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR userPin[] = {"MyPIN"};
CK_RV rv;

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

5.7 Object management functions

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

5.7.1 C_CreateObject

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

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

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

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

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:
CK_SESSION_HANDLE hSession;

CK_OBJECT_HANDLE
  hData,
  hCertificate,
  hKey;

CK_OBJECT_CLASS
  dataClass = CKO_DATA,
  certificateClass = CKO_CERTIFICATE,
  keyClass = CKO_PUBLIC_KEY;

CK_KEY_TYPE keyType = CKK_RSA;

CK_UTF8CHAR application[] = {"My Application");
CK_BYTE dataValue[] = {...};
CK_BYTE subject[] = {...};
CK_BYTE id[] = {...};
CK_BYTE certificateValue[] = {...};
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BBOOL true = CK_TRUE;

CK_ATTRIBUTE dataTemplate[] = {
  {CKA_CLASS, &dataClass, sizeof(dataClass)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_APPLICATION, application, sizeof(application)-1},
  {CKA_VALUE, dataValue, sizeof(dataValue)}
};

CK_ATTRIBUTE certificateTemplate[] = {
  {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_VALUE, certificateValue, sizeof(certificateValue)}
};

CK_ATTRIBUTE keyTemplate[] = {
  {CKA_CLASS, &keyClass, sizeof(keyClass)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_WRAP, &true, sizeof(true)},
  {CKA_MODULUS, modulus, sizeof(modulus)},
  {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};

CK_RV rv;

/* Create a data object */
rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
if (rv == CKR_OK) {
  .
  .
  }
/* Create a certificate object */
rv = C_CreateObject(
  hSession, certificateTemplate, 5, &hCertificate);
if (rv == CKR_OK) {
  .
  .
  }
/* Create an RSA public key object */
rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
if (rv == CKR_OK) {
  .
  .
  }

5.7.2 C_CopyObject

CK_DECLARE_FUNCTION(CK_RV, C_CopyObject)(
  CK_SESSION_HANDLE hSession,
  CK_OBJECT_HANDLE hObject,
  CK_ATTRIBUTE_PTR pTemplate,
  CK_ULONG ulCount,
  CK_OBJECT_HANDLE_PTR phNewObject
);

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

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

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

Whenever an object is copied, a new value for CKA_UNIQUE_ID is generated and assigned to the new object (See Section 4.4.1).
Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_CHARACTERISTICS_MISMATCH, CKR_SESSION_READ_ONLY, CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey, hNewKey;
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_Type keyType = CKK_DES;
CK_BYTE id[] = {...};
CK_BYTE keyValue[] = {...};
CK_BBOOL false = CK_FALSE;
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE keyTemplate[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &false, sizeof(false)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, keyValue, sizeof(keyValue)}
};
CK_ATTRIBUTE copyTemplate[] = {
    {CKA_TOKEN, &true, sizeof(true)}
};
CK_RV rv;
```

/* Create a DES secret key session object */
rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
if (rv == CKR_OK) {
    /* Create a copy which is a token object */
    rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);
}

5.7.3 C_DestroyObject

```
CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject)(
    CK_SESSION_HANDLE hSession,
    ```
C_DestroyObject destroys an object. hSession is the session's handle; and hObject is the object's handle.

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

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

Return values: CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

Example: see C_GetObjectSize.

5.7.4 C_GetObjectSize

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_OBJECT_CLASS dataClass = CKO_DATA;
CK_UTF8CHAR application[] = {"My Application"};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &dataClass, sizeof(dataClass)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_APPLICATION, application, sizeof(application)-1},
    {CKA_VALUE, value, sizeof(value)}
};
CK_ULONG ulSize;
CK_RV rv;

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

5.7.5 C_GetAttributeValue

CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);

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

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

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

If case 1 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes, then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the requested attributes will CKR_OK be returned.
In the special case of an attribute whose value is an array of attributes, for example
CKA_WRAP_TEMPLATE, where it is passed in with pValue not NULL, the length specified in ulValueLen
MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is
NULL_PTR then the ulValueLen of elements within the array will be set to the required length. If the
pValue of elements within the array is not NULL_PTR, then the ulValueLen element of attributes within
the array MUST reflect the space that the corresponding pValue points to, and pValue is filled in if there is
sufficient room. Therefore it is important to initialize the contents of a buffer before calling
C_GetAttributeValue to get such an array value. Note that the type element of attributes within the array
MUST be ignored on input and MUST be set on output. If any ulValueLen within the array isn't large
enough, it will be set to CK_UNAVAILABLE_INFORMATION and the function will return
CKR_BUFFER_TOO_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too
small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
type having the CKF_ARRAY_ATTRIBUTE bit set.

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE,
CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL,
CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED,
CKR_SESSION_HANDLE_INVALID.

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_BYTE_PTR pModulus, pExponent;
CK_ATTRIBUTE template[] = {
    {CKA_MODULUS, NULL_PTR, 0},
    {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
};

CK_RV rv;
```

```c
rv = C_GetAttributeValue(hSession, hObject, template, 2);
if (rv == CKR_OK) {
    pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
    template[0].pValue = pModulus;
    /* template[0].ulValueLen was set by C_GetAttributeValue */

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

    rv = C_GetAttributeValue(hSession, hObject, template, 2);
```
5.7.6 C_SetAttributeValue

```c
CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);  

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

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

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

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

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

Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:
```
5.7.7 C_FindObjectsInit

```c
CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit)(
    CK_SESSION_HANDLE hSession,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);
```

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

After calling C_FindObjectsInit, the application may call C_FindObjects one or more times to obtain handles for objects matching the template, and then eventually call C_FindObjectsFinal to finish the active search operation. At most one search operation may be active at a given time in a given session. The object search operation will only find objects that the session can view. For example, an object search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the search template specifies that the search is for private objects).

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

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

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

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

Example: see C_FindObjectsFinal.

5.7.8 C_FindObjects

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

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

If there are no more objects matching the template, then the location that pulObjectCount points to receives the value 0.
The search MUST have been initialized with C_FindObjectsInit.
Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
Example: see C_FindObjectsFinal.

5.7.9 C_FindObjectsFinal

C_FindObjectsFinal terminates a search for token and session objects. hSession is the session's
handle.
Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
CKR_SESSION_HANDLE_INVALID.
Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_ULONG ulObjectCount;
CK_RV rv;
rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
assert(rv == CKR_OK);
while (1) {
    rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
    if (rv != CKR_OK || ulObjectCount == 0)
        break;
}
rv = C_FindObjectsFinal(hSession);
assert(rv == CKR_OK);
```

5.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

5.8.1 C_EncryptInit

```c
CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
);`
C_EncryptInit initializes an encryption operation. hSession is the session's handle; pMechanism points to the encryption mechanism; hKey is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

Example: see C_EncryptFinal.

5.8.2 C_Encrypt

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

C_Encrypt uses the convention described in Section 5.2 on producing output. The encryption operation MUST have been initialized with C_EncryptInit. A call to C_Encrypt always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

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

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

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

For most mechanisms, C_Encrypt is equivalent to a sequence of C_EncryptUpdate operations followed by C_EncryptFinal.
Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see C_EncryptFinal for an example of similar functions.

5.8.3 C_EncryptUpdate

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

C_EncryptUpdate continues a multiple-part encryption operation, processing another data part.

hSession is the session’s handle; pPart points to the data part; ulPartLen is the length of the data part; pEncryptedPart points to the location that receives the encrypted data part; pulEncryptedPartLen points to the location that holds the length in bytes of the encrypted data part.

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

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

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

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

Example: see C_EncryptFinal.

5.8.4 C_EncryptFinal

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

C_EncryptFinal finishes a multiple-part encryption operation. hSession is the session’s handle; pLastEncryptedPart points to the location that receives the last encrypted data part, if any; pulLastEncryptedPartLen points to the location that holds the length of the last encrypted data part.

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

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

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism’s input data MUST consist of an integral number of blocks. If these constraints are not satisfied, then C_EncryptFinal will fail with return code CKR_DATA_LEN_RANGE.
Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```c
#define PLAINTEXT_BUF_SZ 200
#define CIPHERTEXT_BUF_SZ 256

CK_ULONG firstPieceLen, secondPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
    CKM_DES_CBC_PAD, iv, sizeof(iv)
};
CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulEncryptedData1Len;
CK_ULONG ulEncryptedData2Len;
CK_ULONG ulEncryptedData3Len;
CK_RV rv;

firstPieceLen = 90;
secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
rv = C_EncryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    /* Encrypt first piece */
    ulEncryptedData1Len = sizeof(encryptedData);
    rv = C_EncryptUpdate(
        hSession,
        &data[0], firstPieceLen,
        &encryptedData[0], &ulEncryptedData1Len);
    if (rv != CKR_OK) {
        
    }
    /* Encrypt second piece */
    ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
    rv = C_EncryptUpdate(
        hSession,
        &data[0], secondPieceLen,
        &encryptedData[0], &ulEncryptedData2Len);
```
&data[firstPieceLen], secondPieceLen,
&encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
if (rv != CKR_OK) {
  .
  .
  }
/* Get last little encrypted bit */
ulEncryptedData3Len =
  sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
rv = C_EncryptFinal(
  hSession,
  &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
  &ulEncryptedData3Len);
if (rv != CKR_OK) {
  .
  .
  }
}

5.9 Message-based encryption functions

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

5.9.1 C_MessageEncryptInit

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

C_MessageEncryptInit prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. hSession is the session's handle; pMechanism points to the encryption mechanism; hKey is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling C_MessageEncryptInit, the application can either call C_EncryptMessage to encrypt a message in a single part, or call C_EncryptMessageBegin, followed by C_EncryptMessageNext one or more times, to encrypt a message in multiple parts. This may be repeated several times. The message-based encryption process is active until the application calls C_MessageEncryptFinal to finish the message-based encryption process.

C_MessageEncryptInit can be called with pMechanism set to NULL_PTR to terminate a message-based encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an active operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.
Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_ERROR, CKR DEVICE_MEMORY,
CKR_DEVICE_REMOVED, CKR FUNCTION CANCELED, CKR FUNCTION FAILED,
CKR_GENERAL_ERROR, CKR HOST_MEMORY, CKR KEY_FUNCTION_NOT_PERMITTED,
CKR KEY_HANDLE_INVALID, CKR KEY_SIZE_RANGE, CKR KEY_TYPE_INCONSISTENT,
CKR_MECHANISM_INVALID, CKR MECHANISM_PARAM_INVALID, CKR_OK,
CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
CKR_OPERATION_CANCEL_FAILED.

5.9.2 C_EncryptMessage

```c
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage)(
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR p AssociatedData,
    CK_ULONG ul AssociatedDataLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG ulPlaintextLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG_PTR pulCiphertextLen
);
```

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

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

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

C_EncryptMessage uses the convention described in Section 5.2 on producing output.
The message-based encryption process MUST have been initialized with C_MessageEncryptInit. A call
to C_EncryptMessage begins and terminates a message encryption operation.
C_EncryptMessage cannot be called in the middle of a multi-part message encryption operation.
For some encryption mechanisms, the input plaintext data has certain length constraints (either because
the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
C_EncryptMessage will fail with return code CKR_DATA_LEN_RANGE. The plaintext and ciphertext can
be in the same place, i.e., it is OK if pPlaintext and pCiphertext point to the same location.
For most mechanisms, C_EncryptMessage is equivalent to C_EncryptMessageBegin followed by a
sequence of C_EncryptMessageNext operations.

5.9.3 C_EncryptMessageBegin

```c
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin)(
```
C_EncryptMessageBegin begins a multiple-part message encryption operation. hSession is the session's handle; pParameter and ulParameterLen specify any mechanism-specific parameters for the message encryption operation; pAssociatedData and ulAssociatedDataLen specify the associated data for an AEAD mechanism.

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

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

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

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

5.9.4 C_EncryptMessageNext

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

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

C_EncryptMessageNext uses the convention described in Section 5.2 on producing output.
encryption operation. A call to `C_EncryptMessageNext` with flags=CKF_END_OF_MESSAGE always terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext.

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

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

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

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

### 5.9.5 `C_MessageEncryptFinal`

```c
CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptFinal)(
    CK_SESSION_HANDLE hSession
);
```

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

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

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

Example:

```c
#define PLAINTEXT_BUF_SZ 200
#define AUTH_BUF_SZ 100
#define CIPHERTEXT_BUF_SZ 256

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
CK_BYTE tag[16];
CK_GCM_MESSAGE_PARAMS gcmParams = {
    .iv = iv,
    .sizeof(iv) * 8,
    .0,
    .CKG_NO_GENERATE,
    .tag,
    .sizeof(tag) * 8
```
CK_MECHANISM mechanism = {
    CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
};
CK_BYTE data[2][PLAINTEXT_BUF_SZ];
CK_BYTE auth[2][AUTH_BUF_SZ];
CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
CKULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
CKULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;

/* error handling is omitted for better readability */

C_MessageEncryptInit(hSession, &mechanism, hKey);
/* encrypt message en bloc with given IV */
ulEncryptedDataLen = sizeof(encryptedData[0]);
C_EncryptMessage(hSession,
    &gcmParams, sizeof(gcmParams),
    &auth[0][0], sizeof(auth[0]),
    &data[0][0], sizeof(data[0]),
    &encryptedData[0][0], &ulEncryptedDataLen);
/* iv and tag are set now for message */
/* encrypt message in two steps with generated IV */
gcmParams.ivGenerator = CKG_GENERATE;
C_EncryptMessageBegin(hSession,
    &gcmParams, sizeof(gcmParams),
    &auth[1][0], sizeof(auth[1])
);
/* encrypt first piece */
ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
C_EncryptMessageNext(hSession,
    &gcmParams, sizeof(gcmParams),
    &data[1][0], firstPieceLen,
    &encryptedData[1][0], &ulFirstEncryptedDataLen,
    0);
/* encrypt second piece */
ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
C_EncryptMessageNext(hSession,
    &gcmParams, sizeof(gcmParams),
    &data[1][firstPieceLen], sizeof(data[1]) - firstPieceLen,
    &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,
    0);
5.10 Decryption functions

Cryptoki provides the following functions for decrypting data:

5.10.1 C_DecryptInit

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

C_DecryptInit initializes a decryption operation. `hSession` is the session's handle; `pMechanism` points to the decryption mechanism; `hKey` is the handle of the decryption key.

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

After calling C_DecryptInit, the application can either call C_Decrypt to decrypt data in a single part; or call C_DecryptUpdate zero or more times, followed by C_DecryptFinal, to decrypt data in multiple parts.

The decryption operation is active until the application uses a call to C_Decrypt or C_DecryptFinal to actually obtain the final piece of plaintext. To process additional data (in single or multiple parts), the application MUST call C_DecryptInit again.

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

Example: see C_DecryptFinal.

5.10.2 C_Decrypt

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

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

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

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

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

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

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

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

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

Example: see C_DecryptFinal for an example of similar functions.

5.10.3 C_DecryptUpdate

```c
CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) {
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG ulEncryptedPartLen,
    CK_BYTE_PTR pPart,
    CK_ULONG_PTR pulPartLen
};
```

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

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

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

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

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

Example: See C_DecryptFinal.
5.10.4 C_DecryptFinal

```c
CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pLastPart,
    CK_UULONG_PTR pulLastPartLen
);
```

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

C_DecryptFinal uses the convention described in Section 5.2 on producing output. The decryption operation MUST have been initialized with C_DecryptInit. A call to C_DecryptFinal always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

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

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

Example:

```c
#define CIPHERTEXT_BUF_SZ 256
#define PLAINTEXT_BUF_SZ 256

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

firstEncryptedPieceLen = 90;
secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ - firstEncryptedPieceLen;
rv = C_DecryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    /* Decrypt first piece */
    ulData1Len = sizeof(data);
```
rv = C_DecryptUpdate{
    hSession,
    &encryptedData[0], firstEncryptedPieceLen,
    &data[0], &ulData1Len);
if (rv != CKR_OK) {
    .
    .
    }
} /* Decrypt second piece */
ulData2Len = sizeof(data)-ulData1Len;
rv = C_DecryptUpdate{
    hSession,
    &encryptedData[firstEncryptedPieceLen],
    secondEncryptedPieceLen,
    &data[ulData1Len], &ulData2Len);
if (rv != CKR_OK) {
    .
    .
    }
} /* Get last little decrypted bit */
ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
rv = C_DecryptFinal{
    hSession,
    &data[ulData1Len+ulData2Len], &ulData3Len);
if (rv != CKR_OK) {
    .
    .
    }
} 5.11 Message-based decryption functions
Message-based decryption refers to the process of decrypting multiple encrypted messages using the same decryption mechanism and decryption key. The decryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm. Cryptoki provides the following functions for message-based decryption.

5.11.1 C_MessageDecryptInit

CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
C_MessageDecryptInit initializes a message-based decryption process, preparing a session for one or more decryption operations that use the same decryption mechanism and decryption key. hSession is the session's handle; pMechanism points to the decryption mechanism; hKey is the handle of the decryption key.

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

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

5.11.2 C_DecryptMessage

C_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) {
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG ulCiphertextLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG_PTR pulPlaintextLen
};

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

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

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

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

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

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

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

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

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.
For most mechanisms, \texttt{C\_DecryptMessage} is equivalent to \texttt{C\_DecryptMessageBegin} followed by a sequence of \texttt{C\_DecryptMessageNext} operations.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_BUFFER\_TOO\_SMALL},
\texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY},
\texttt{CKR\_DEVICE\_REMOVED}, \texttt{CKR\_ENCCRYPTED\_DATA\_INVALID},
\texttt{CKR\_ENCRYPTED\_DATA\_LEN\_RANGE}, \texttt{CKR\_AEAD\_DECRYPT\_FAILED},
\texttt{CKR\_FUNCTION\_CANCELED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR},
\texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_OK}, \texttt{CKR\_OPERATION\_NOT\_INITIALIZED}, \texttt{CKR\_SESSION\_CLOSED},
\texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_USER\_NOT\_LOGGED\_IN},
\texttt{CKR\_OPERATION\_CANCEL\_FAILED}.

\subsection*{5.11.3 \texttt{C\_DecryptMessageBegin}}

\begin{verbatim}
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) {
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
};
\end{verbatim}

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

Typically, \texttt{pParameter} is an initialization vector (IV) or nonce. Unlike the \texttt{pParameter} parameter of \texttt{C\_EncryptMessageBegin}, \texttt{pParameter} is always an input parameter.

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

After calling \texttt{C\_DecryptMessageBegin}, the application should call \texttt{C\_DecryptMessageNext} one or more times to decrypt the encrypted message in multiple parts. The message decryption operation is active until the application uses a call to \texttt{C\_DecryptMessageNext} with flags=\texttt{CKF\_END\_OF\_MESSAGE} to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or multiple parts), the application MUST call \texttt{C\_DecryptMessage} or \texttt{C\_DecryptMessageBegin} again.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED},
\texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY}, \texttt{CKR\_DEVICE\_REMOVED},
\texttt{CKR\_FUNCTION\_CANCELED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR},
\texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_OK}, \texttt{CKR\_OPERATION\_ACTIVE}, \texttt{CKR\_PIN\_EXPIRED},
\texttt{CKR\_SESSION\_CLOSED}, \texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_USER\_NOT\_LOGGED\_IN}.

\subsection*{5.11.4 \texttt{C\_DecryptMessageNext}}

\begin{verbatim}
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) {
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pCiphertextPart,
    CK_ULONG ulCiphertextPartLen,
    CK_BYTE_PTR pPlaintextPart,
    CK_ULONG_PTR pulPlaintextPartLen,
    CK_FLAGS flags
};
\end{verbatim}

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

Typically, \texttt{pParameter} is an initialization vector (IV) or nonce. Unlike the \texttt{pParameter} parameter of
\texttt{C\_EncryptMessageNext}, \texttt{pParameter} is always an input parameter.

\texttt{C\_DecryptMessageNext} uses the convention described in Section 5.2 on producing output.

The message decryption operation MUST have been started with \texttt{C\_DecryptMessageBegin}. This
function may be called any number of times in succession. A call to \texttt{C\_DecryptMessageNext} with
flags=0 which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message
decryption operation. A call to \texttt{C\_DecryptMessageNext} with flags=CKF\_END\_OF\_MESSAGE always
terminates the active message decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a
successful call (i.e., one which returns CKR\_OK) to determine the length of the buffer needed to hold the
plaintext.

The ciphertext and plaintext can be in the same place, i.e., it is OK if \texttt{pCiphertextPart} and \texttt{pPlaintextPart}
point to the same location.

Although the last \texttt{C\_DecryptMessageNext} call ends the decryption of a message, it does not finish the
message-based decryption process. Additional \texttt{C\_DecryptMessage} or \texttt{C\_DecryptMessageBegin} and
\texttt{C\_DecryptMessageNext} calls may be made on the session.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
\texttt{CKR\_ENCRYPTED\_DATA\_INVALID} or \texttt{CKR\_ENCRYPTED\_DATA\_LEN\_RANGE} may be returned by
the last \texttt{C\_DecryptMessageNext} call.

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
ciphertext cannot be verified, then \texttt{CKR\_AEAD\_DECRYPT\_FAILED} is returned by the last
\texttt{C\_DecryptMessageNext} call.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_BUFFER\_TOO\_SMALL},
\texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY},
\texttt{CKR\_DEVICE\_REMOVED}, \texttt{CKR\_ENCRYPTED\_DATA\_INVALID},
\texttt{CKR\_ENCRYPTED\_DATA\_LEN\_RANGE}, \texttt{CKR\_AEAD\_DECRYPT\_FAILED},
\texttt{CKR\_FUNCTION\_CANCELED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR},
\texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_OK}, \texttt{CKR\_OPERATION\_NOT\_INITIALIZED}, \texttt{CKR\_SESSION\_CLOSED},
\texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_USER\_NOT\_LOGGED\_IN}.

\subsection{5.11.5 C\_MessageDecryptFinal}

\begin{verbatim}
CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal)(
    CK_SESSION_HANDLE hSession
);}
\end{verbatim}

\texttt{C\_MessageDecryptFinal} finishes a message-based decryption process. \texttt{hSession} is the session’s
handle.

The message-based decryption process MUST have been initialized with \texttt{C\_MessageDecryptInit}.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED},
\texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY}, \texttt{CKR\_DEVICE\_REMOVED},
\texttt{CKR\_FUNCTION\_CANCELED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR},
\texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_OK}, \texttt{CKR\_OPERATION\_NOT\_INITIALIZED}, \texttt{CKR\_SESSION\_CLOSED},
\texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_USER\_NOT\_LOGGED\_IN}.

\subsection{5.12 Message digesting functions}

Cryptoki provides the following functions for digesting data:
5.12.1 C_DigestInit

```c
CK_DECLARE_FUNCTION(CK_RV, C_DigestInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism
);
```

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

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

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

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

Example: see C_DigestFinal.

5.12.2 C_Digest

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

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

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

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

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

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.
Example: see **C_DigestFinal** for an example of similar functions.

### 5.12.3 C_DigestUpdate

```c
CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CKULONG ulPartLen
);
```

*C_DigestUpdate* continues a multiple-part message-digesting operation, processing another data part. 

*hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the data part.

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

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

Example: see **C_DigestFinal**.

### 5.12.4 C_DigestKey

```c
CK_DECLARE_FUNCTION(CK_RV, C_DigestKey)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hKey
);
```

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

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

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

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

Example: see **C_DigestFinal**.

### 5.12.5 C_DigestFinal

```c
CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pDigest,
    CKULONG_PTR pulDigestLen
);
```

*C_DigestFinal* finishes a multiple-part message-digesting operation, returning the message digest. 

*hSession* is the session’s handle; *pDigest* points to the location that receives the message digest; 

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

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

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

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

Example:

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

CK_RV rv;

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

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

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

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

5.13 Signing and MACing functions

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

5.13.1 C_SignInit

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

The `C_SignInit` initializes a signature operation, where the signature is an appendix to the data. `hSession` is the session's handle; `pMechanism` points to the signature mechanism; `hKey` is the handle of the signature key.

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

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

`C_SignInit` can be called with `pMechanism` set to NULL_PTR to terminate an active signature operation.

If an operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

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

Example: see `C_SignFinal`.

5.13.2 C_Sign

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

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

The signing operation MUST have been initialized with `C_SignInit`. A call to `C_Sign` always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.
4771 C_Sign cannot be used to terminate a multi-part operation, and MUST be called after C_SignInit without
4772 intervening C_SignUpdate calls.
4773 For most mechanisms, C_Sign is equivalent to a sequence of C_SignUpdate operations followed by
4774 C_SignFinal.
4775 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED, CKR_TOKEN_RESOURCE_EXCEEDED.
4778 Example: see C_SignFinal for an example of similar functions.

5.13.3 C_SignUpdate

4784 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate)(
4785     CK_SESSION_HANDLE hSession,
4786     CK_BYTE_PTR pPart,
4787     CK_ULONG ulPartLen
4788 )

4789 C_SignUpdate continues a multi-part signature operation, processing another data part. hSession is
4790 the session's handle, pPart points to the data part; ulPartLen is the length of the data part.
4791 The signature operation MUST have been initialized with C_SignInit. This function may be called any
4792 number of times in succession. A call to C_SignUpdate which results in an error terminates the current
4793 signature operation.
4794 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_RESOURCE_EXCEEDED.
4797 Example: see C_SignFinal.

5.13.4 C_SignFinal

4802 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal)(
4803     CK_SESSION_HANDLE hSession,
4804     CK_BYTE_PTR pSignature,
4805     CK_ULONG_PTR pulSignatureLen
4806 )

4807 C_SignFinal finishes a multi-part signature operation, returning the signature. hSession is the
4808 session's handle; pSignature points to the location that receives the signature; pulSignatureLen points to
4809 the location that holds the length of the signature.
4810 C_SignFinal uses the convention described in Section 5.2 on producing output.
4811 The signing operation MUST have been initialized with C_SignInit. A call to C_SignFinal always
4812 terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4813 call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.
4814 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5.13.5 C_SignRecoverInit

C_SignRecoverInit initializes a signature operation, where the data can be recovered from the signature. 

$hSession$ is the session's handle; $pMechanism$ points to the structure that specifies the signature mechanism; $hKey$ is the handle of the signature key.

The $CKA_SIGN_RECOVER$ attribute of the signature key, which indicates whether the key supports signatures where the data can be recovered from the signature, MUST be $CK_TRUE$.

After calling $C_SignRecoverInit$, the application may call $C_SignRecover$ to sign in a single part. The signature operation is active until the application uses a call to $C_SignRecover$ to actually obtain the signature. To process additional data in a single part, the application MUST call $C_SignRecoverInit$ again.

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

Return values: $CKR_ARGUMENTS_BAD$, $CKR_CRYPTOKI_NOT_INITIALIZED$, $CKR_DEVICE_ERROR$, $CKR_DEVICE_MEMORY$, $CKR_DEVICE_REMOVED$, $CKR_USER_NOT_LOGGED_IN$, $CKR_FUNCTION_REJECTED$, $CKR_TOKEN_RESOURCE_EXCEEDED$. 

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE mac[4];
CK_ULONG ulMacLen;
CK_RV rv;

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

5.13.6 C_SignRecover

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

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

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

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

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

Example:

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

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

5.14 Message-based signing and MACing functions

Message-based signature refers to the process of signing multiple messages using the same signature mechanism and signature key. Cryptoki provides the following functions for signing messages (for the purposes of Cryptoki, these operations also encompass message authentication codes).

5.14.1 C_MessageSignInit

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

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

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

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

5.14.2 C_SignMessage

CK_DECLARE_FUNCTION(CK_RV, C_SignMessage)(
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
CK_BYTE_PTR pSignature,
CK ULONG_PTR pulSignatureLen

C_SignMessage signs a message in a single part, where the signature is an appendix to the message. C_MessageSignInit must previously been called on the session. hSession is the session’s handle; pParameter and ulParameterLen specify any mechanism-specific parameters for the message signature operation; pData points to the data; ulDataLen is the length of the data; pSignature points to the location that receives the signature; pulSignatureLen points to the location that holds the length of the signature. Depending on the mechanism parameter passed to C_MessageSignInit, pParameter may be either an input or an output parameter.

C_SignMessage uses the convention described in Section 5.2 on producing output. The message-based signing process MUST have been initialized with C_MessageSignInit. A call to C_SignMessage begins and terminates a message signing operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a successful call (i.e., one which returns CKR_OK).

C_SignMessage cannot be called in the middle of a multi-part message signing operation. C_SignMessage does not finish the message-based signing process. Additional C_SignMessage or C_MessageBegin and C_MessageNext calls may be made on the session.

For most mechanisms, C_SignMessage is equivalent to C_SignMessageBegin followed by a sequence of C_SignMessageNext operations. Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED, CKR_TOKEN_RESOURCE_EXCEEDED.

5.14.3 C_SignMessageBegin

C_SignMessageBegin begins a multiple-part message signature operation, where the signature is an appendix to the message. C_MessageSignInit must previously been called on the session. hSession is the session’s handle; pParameter and ulParameterLen specify any mechanism-specific parameters for the message signature operation. Depending on the mechanism parameter passed to C_MessageSignInit, pParameter may be either an input or an output parameter.

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5.14.4 C_SignMessageNext

```c
CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) {
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CKULONG ulParameterLen,
    CK_BYTE_PTR pDataPart,
    CKULONG ulDataPartLen,
    CK_BYTE_PTR pSignature,
    CKULONG_PTR pulSignatureLen
};
```

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

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

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

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

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

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

5.14.5 C_MessageSignFinal

```c
CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) {
    CK_SESSION_HANDLE hSession
};
```

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

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

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,
5.15 Functions for verifying signatures and MACs

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

5.15.1 C_VerifyInit

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

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

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

After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.

To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

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


Example: see **C_VerifyFinal**.

5.15.2 C_Verify

```c
CK_DECLARE_FUNCTION(CK_RV, C_Verify)(
    CK_SESSION HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG ulSignatureLen
);
```

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

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

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

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

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

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

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

### 5.15.3 C_VerifyUpdate

```c
CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen
);
```

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

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

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

Example: see **C_VerifyFinal**.

### 5.15.4 C_VerifyFinal

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

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

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_BYTE mac[4];
CK_RV rv;

rv = C_VerifyInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    rv = C_VerifyUpdate(hSession, data, sizeof(data));
    rv = C_VerifyFinal(hSession, mac, sizeof(mac));
}
```

5.15.5 C_VerifyRecoverInit

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

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

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

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

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

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

Example: see C_VerifyRecover.

5.15.6 C_VerifyRecover

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

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

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

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

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

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

Example:

```
CK_RV rv;
```

5191 5192 5193 5194 5195 5196 5197 5198 5199 5200 5201 5202 5203 5204 5205 5206 5207 5208 5209 5210 5211 5212 5213 5214 5215 5216 5217 5218 5219 5220 5221 5222 5223 5224 5225 5226 5227 5228 5229 5230 5231 5232 5233 5234 5235 5236 5237 5238
rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    ulDataLen = sizeof(data);
    rv = C_VerifyRecover(
           hSession, signature, sizeof(signature), data, &ulDataLen);
    .
}

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
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
C_VerifyMessage
CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin)(
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen
);

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

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

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

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.

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.

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

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

5.16.3 C_VerifyMessageBegin

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.

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

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.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKRDEVICE_REMOVED,
CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR,
CKRHOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
CKR_SESSION_CLOSED, CKR_SESSIONHANDLE_INVALID, CKR_TOKENRESOURCE_EXCEEDED.
5.16.4 C_VerifyMessageNext

```c
CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext)(
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CKULONG ulParameterLen,
    CK_BYTE_PTR pDataPart,
    CKULONG ulDataPartLen,
    CK_BYTE_PTR pSignature,
    CKULONG ulSignatureLen)
```

C_VerifyMessageNext continues a multiple-part message verification operation, processing another data part, or finishes a multiple-part message verification operation, checking the signature. `hSession` is the session's handle, `pParameter` and `ulParameterLen` specify any mechanism-specific parameters for the message verification operation, `pDataPart` points to the data part; `ulDataPartLen` is the length of the data part; `pSignature` points to the signature; `ulSignatureLen` is the length of the signature.

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

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

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

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

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SIGNATURE_LEN_RANGE`, `CKR_TOKEN_RESOURCE_EXCEEDED`.

5.16.5 C_MessageVerifyFinal

```c
CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal)(
    CK_SESSION_HANDLE hSession
)
```

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

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

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_SIGNATURE_LEN_RANGE`, `CKR_TOKEN_RESOURCE_EXCEEDED`. 
5.17 Dual-function cryptographic functions

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

5.17.1 C_DigestEncryptUpdate

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

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

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

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

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

Example:

```c
#define BUF_SZ 512

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

memset(iv, 0, sizeof(iv));
memset(data, 'A', ((2*BUF_SZ)+5));
rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
if (rv != CKR_OK) {
    
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK) {
    
}

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

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

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

/* First, complete the digest on the buffer */
rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
.
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
.
.
/* Then, pad last part with 3 0x00 bytes, and complete encryption */
for(i=0; i<3; i++)
    data[((BUF_SZ*2)+5)+i] = 0x00;
.
/* Now, get second-to-last piece of ciphertext */
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_EncryptUpdate(hSession,
    &data[BUF_SZ*2], 8,
    encryptedData, &ulEncryptedDataLen);
.
/* Get last piece of ciphertext (should have length 0, here) */
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
.

5.17.2 C_DecryptDigestUpdate

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

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

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

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

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

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

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

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

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

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

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

Example:

```c
#define BUF_SZ 512

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

memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
if (rv != CKR_OK) {
   .
   .
   .
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK) {
   .
   .
   .
}
ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
   hSession,
   &encryptedData[0], BUF_SZ,
   data, &ulDataLen);
   .
   .
ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
   hSession,
   &encryptedData[BUF_SZ], BUF_SZ,
   data, &ulDataLen);
   .
   .
/*
 * The last portion of the buffer needs to be handled with
 * separate calls to deal with padding issues in ECB mode
 */
ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate(
   hSession,
   &encryptedData[BUF_SZ*2], 8,
  /* Get last piece of plaintext (should have length 0, here) */
  ulDataLen = sizeof(data) - ulLastUpdateSize;
  rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
  if (rv != CKR_OK) {
      .
      .
  }

  /* Digest last bit of plaintext */
  rv = C_DigestUpdate(hSession, data, 5);
  if (rv != CKR_OK) {
      .
      .
  }

  ulDigestLen = sizeof(digest);
  rv = C_DigestFinal(hSession, digest, &ulDigestLen);
  if (rv != CKR_OK) {
      .
      .
  }

5.17.3 C_SignEncryptUpdate

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

C_SignEncryptUpdate continues a multiple-part combined signature and encryption operation, processing another data part. hSession is the session's handle; pPart points to the data part; ulPartLen is the length of the data part; pEncryptedPart points to the location that receives the digested and encrypted data part; and pulEncryptedPartLen points to the location that holds the length of the encrypted data part. C_SignEncryptUpdate uses the convention described in Section 5.2 on producing output. If a C_SignEncryptUpdate call does not produce encrypted output (because an error occurs, or because pEncryptedPart has the value NULL_PTR, or because pulEncryptedPartLen is too small to hold the entire encrypted part output), then no plaintext is passed to the active signing operation. Signature and encryption operations MUST both be active (they MUST have been initialized with C_SignInit and C_EncryptInit, respectively). This function may be called any number of times in succession, and may be interspersed with C_SignUpdate and C_EncryptUpdate calls. Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKRDEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
CKR_USER_NOT_LOGGED_IN.

Example:

```c
#define BUF_SZ 512

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

memset(iv, 0, sizeof(iv));
memset(data, 'A', ((2*BUF_SZ)+5));
rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
if (rv != CKR_OK) {
    .
    .
    .
    rv = C_SignInit(hSession, &signMechanism, hMacKey);
    if (rv != CKR_OK) {
        .
        .
        .
    }
}
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_SignEncryptUpdate(
    hSession,
    &data[0], BUF_SZ,
    encryptedData, &ulEncryptedDataLen);
```
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_SignEncryptUpdate(hSession,
    &data[BUF_SZ], BUF_SZ,
    encryptedData, &ulEncryptedDataLen);

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

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

ulMacLen = sizeof(MAC);
rv = C_SignFinal(hSession, MAC, &ulMacLen);

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

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

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

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

C_DecryptVerifyUpdate continues a multi-part combined decryption and verification operation, processing another data part. hSession is the session’s handle; pEncryptedPart points to the encrypted data; ulEncryptedPartLen is the length of the encrypted data; pPart points to the location that receives the recovered data; and pulPartLen points to the location that holds the length of the recovered data.

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

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

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

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

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

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

A call to C_VerifyFinal, therefore, would verify whether or not the signature supplied is a valid signature on the first 16 bytes of the plaintext, not on the entire plaintext. It is crucial that, before C_VerifyFinal is called, the last 2 bytes of plaintext get passed into the active verification operation via a C_VerifyUpdate call.

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

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKRDEVICE_MEMORY, CKRDEVICE_REMOVED, CKRCRYPTOKI_DATA_INVALID, CKRENCRYPTED_DATA_LEN_RANGE, CKRFUNCTION_CANCELED, CKRFUNCTION_FAILED, CKRGENERAL_ERROR, CKRHOST_MEMORY, CKROK, CKROPERATION_NOT_INITIALIZED, CKRSESSIONCLOSED, CKRSESSIONHANDLE_INVALID.

Example:

```c
#define BUF_SZ 512
```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM decryptionMechanism = {
    CKM_DES_ECB, iv, sizeof(iv)
};
CK_MECHANISM verifyMechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE encryptedData[(2*BUF_SZ)+8];
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;

memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
if (rv != CKR_OK) {
    .
    .
    .
    
rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
if (rv != CKR_OK){
    .
    .
    .
    }
ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
    hSession,
    &encryptedData[0], BUF_SZ,
    data, &ulDataLen);
    .
    .
ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
    hSession,
    &encryptedData[BUF_SZ], BUF_SZ,
data, &ulDataLen);

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

/* First, complete the decryption of the buffer */
ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate(
    hSession,
    encryptedData[BUF_SZ*2], 8,
    data, &ulLastUpdateSize);

/* Get last little piece of plaintext. Should have length 0 */
ulDataLen = sizeof(data)-ulLastUpdateSize;
rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
if (rv != CKR_OK) {
    .
    .
    .
}

/* Send last bit of plaintext to verification operation */
rv = C_VerifyUpdate(hSession, data, 5);
if (rv != CKR_OK) {
    .
    .
    .
}
rv = C_VerifyFinal(hSession, MAC, ulMacLen);
if (rv == CKR_SIGNATURE_INVALID) {
    .
    .
    .
}

5.18 Key management functions
Cryptoki provides the following functions for key management:

5.18.1 C_GenerateKey

CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey)(
    CK_SESSION_HANDLE hSession
)
CK_MECHANISM_PTR pMechanism,
CK_ATTRIBUTE_PTR pTemplate,
CK_ULONG ulCount,
CK_OBJECT_HANDLE_PTR phKey
);

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

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

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

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

The object created by a successful call to C_GenerateKey will have its CKA_LOCAL attribute set to CK_TRUE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CRYPTOKI_FUNCTION_FAILED, CKR_CRYPTOKI_HOST_MEMORY, CKR_CRYPTOKI_PIN_RETRY_LIMIT, CKR_CRYPTOKI_PIN_RETRY_LIMIT_EXCEEDED, CKR_MECHANISM_FUNCTION_FAILED, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_SESSION_READ_WRITE, CKR_SESSION_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
    CKM_DES_KEY_GEN, NULL_PTR, 0
};

CK_RV rv;

rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
if (rv == CKR_OK) {
    ...
}
5.18.2 C_GenerateKeyPair

```c
CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_ATTRIBUTE_PTR pPublicKeyTemplate,
    CK_ULONG ulPublicKeyAttributeCount,
    CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
    CK_ULONG ulPrivateKeyAttributeCount,
    CK_OBJECT_HANDLE_PTR phPublicKey,
    CK_OBJECT_HANDLE_PTR phPrivateKey

);```

C_GenerateKeyPair generates a public/private key pair, creating new key objects. `hSession` is the session's handle; `pMechanism` points to the key generation mechanism; `pPublicKeyTemplate` points to the template for the public key; `ulPublicKeyAttributeCount` is the number of attributes in the public-key template; `pPrivateKeyTemplate` points to the template for the private key; `ulPrivateKeyAttributeCount` is the number of attributes in the private-key template; `phPublicKey` points to the location that receives the handle of the new public key; `phPrivateKey` points to the location that receives the handle of the new private key.

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, C_GenerateKeyPair fails and returns the error code CKR_ATTRIBUTE_TYPE_INVALID. The CKA_CLASS attribute is treated similarly.

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

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

The key objects created by a successful call to C_GenerateKeyPair will have their CKA_LOCAL attributes set to CK_TRUE. In addition, the key objects created will both have values for CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
    CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_ULONG modulusBits = 3072;
CK_BYTE publicExponent[] = { 3 };```
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_UNWRAP, &true, sizeof(true)}
};
CK_RV rv;

rv = C_GenerateKeyPair(
    hSession, &mechanism,
    publicKeyTemplate, 5,
    privateKeyTemplate, 8,
    &hPublicKey, &hPrivateKey);
if (rv == CKR_OK) {
    .
    .
}

5.18.3 C_WrapKey

C_DECLARE_FUNCTION(CK_RV, C_WrapKey)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hWrappingKey,
    CK_OBJECT_HANDLE hKey,
    CK_BYTE_PTR pWrappedKey,
    CK_ULONG_PTR pulWrappedKeyLen
);

C_WrapKey wraps (i.e., encrypts) a private or secret key. hSession is the session's handle; pMechanism points to the wrapping mechanism; hWrappingKey is the handle of the wrapping key; hKey is the handle of the key to be wrapped; pWrappedKey points to the location that receives the wrapped key; and pulWrappedKeyLen points to the location that receives the length of the wrapped key.
C_WrapKey uses the convention described in Section 5.2 on producing output.

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

If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its CKA_EXTRACTABLE attribute set to CK_TRUE, then C_WrapKey fails with error code CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism solely because of its length, then C_WrapKey fails with error code CKR_KEY_SIZE_RANGE.

C_WrapKey can be used in the following situations:

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

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

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

Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE, CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hWrappingKey, hKey;
CK_MECHANISM mechanism = {
    .CKM_DES3_ECB, NULL_PTR, 0
};
CK_BYTE wrappedKey[8];
CK_ULONG ulWrappedKeyLen;
CK_RV rv;
```

ulWrappedKeyLen = sizeof(wrappedKey); 
rv = C_WrapKey( 
    hSession, &mechanism, 
    hWrappingKey, hKey, 
);
5.18.4 C_UnwrapKey

C_UnwrapKey unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object. 

hSession is the session's handle; pMechanism points to the unwrapping mechanism; hUnwrappingKey is the handle of the unwrapping key; pWrappedKey points to the wrapped key; ulWrappedKeyLen is the length of the wrapped key; pTemplate points to the template for the new key; ulAttributeCount is the number of attributes in the template; phKey points to the location that receives the handle of the recovered key.

The CKA_UNWRAP attribute of the unwrapping key, which indicates whether the key supports unwrapping, MUST be CK_TRUE.

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

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

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

The key object created by a successful call to C_UnwrapKey will have its CKA_LOCAL attribute set to CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and assigned (See Section 4.4.1).

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT,
Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hUnwrappingKey, hKey;
CK_MECHANISM mechanism = {
    CKM_DES3_ECB, NULL_PTR, 0
};
CK_BYTE wrappedKey[8] = {...};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)}
};
CK_RV rv;
```

```c
rv = C_UnwrapKey(
    hSession, &mechanism, hUnwrappingKey,
    wrappedKey, sizeof(wrappedKey), template, 4, &hKey);
if (rv == CKR_OK) {
    ...
    ...
}
```

---

### 5.18.5 C_DeriveKey

```c
CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hBaseKey,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulAttributeCount,
    CK_OBJECT_HANDLE_PTR phKey
);
```

C_DeriveKey derives a key from a base key, creating a new key object. *hSession* is the session’s handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.
The values of the `CKA_SENSITIVE`, `CKA_ALWAYS_SENSITIVE`, `CKA_EXTRACTABLE`, and `CKA_NEVER_EXTRACTABLE` attributes for the base key affect the values that these attributes can hold for the newly-derived key. See the description of each particular key-derivation mechanism in Section 5.21.2 for any constraints of this type.

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

The key object created by a successful call to `C_DeriveKey` will have its `CKA_LOCAL` attribute set to `CK_FALSE`. In addition, the object created will have a value for `CKA_UNIQUE_ID` generated and assigned (See Section 4.4.1).

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


Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
CK_MECHANISM keyPairMechanism = {
    CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE publicKeyValue[128];
CK_BYTE otherPublicKeyValue[128];
CK_MECHANISM mechanism = {
    CKM_DH_PKCS_DERIVE, otherPublicKeyValue, sizeof(otherPublicKeyValue)
};
CK_ATTRIBUTE template[] = {
    {CKA_VALUE, &publicValue, sizeof(publicValue)}
};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
```
{CKA_PRIME, prime, sizeof(prime)},
{CKA_BASE, base, sizeof(base)}
};

CK_ATTRIBUTE privateKeyTemplate[] = {
  {CKA_DERIVE, &true, sizeof(true)}
};

CK_ATTRIBUTE derivedKeyTemplate[] = {
  {CKA_CLASS, &keyClass, sizeof(keyClass)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_ENCRYPT, &true, sizeof(true)},
  {CKA_DECRYPT, &true, sizeof(true)}
};

CK_RV rv;

rv = C_GenerateKeyPair(
    hSession, &keyPairMechanism,
    publicKeyTemplate, 2,
    privateKeyTemplate, 1,
    &hPublicKey, &hPrivateKey);

if (rv == CKR_OK) {
  rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
  if (rv == CKR_OK) {
    /* Put other guy’s public value in otherPublicValue */
    .
    .
    rv = C_DeriveKey(
        hSession, &mechanism,
        hPrivateKey, derivedKeyTemplate, 4, &hKey);
    if (rv == CKR_OK) {
      .
      .
      }    }

5.19 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

5.19.1 C_SeedRandom

```c
CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom)(
    CK_SESSION_HANDLE hSession,
```
C_SeedRandom mixes additional seed material into the token's random number generator. hSession is the session's handle; pSeed points to the seed material; and.ulSeedLen is the length in bytes of the seed material.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED,
CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see C_GenerateRandom.

5.19.2 C_GenerateRandom

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

C_GenerateRandom generates random or pseudo-random data. hSession is the session's handle;
pRandomData points to the location that receives the random data; and ulRandomLen is the length in bytes of the random or pseudo-random data to be generated.

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

Example:

CK_SESSION_HANDLE hSession;
CK_BYTE seed[] = {...};
CK_BYTE randomData[] = {...};
CK_RV rv;

rv = C_SeedRandom(hSession, seed, sizeof(seed));
if (rv != CKR_OK) {
    .
    .
    .
} else {
    rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
    if (rv == CKR_OK) {
        .
        .
        .
    } else {
        .
        .
        .
    }
}
5.20 Parallel function management functions

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

5.20.1 C_GetFunctionStatus

```c
CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus)(
    CK_SESSION_HANDLE hSession
);
```

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

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

5.20.2 C_CancelFunction

```c
CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction)(
    CK_SESSION_HANDLE hSession
);
```

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

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

5.21 Callback functions

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

5.21.1 Surrender callbacks

Cryptographic functions (i.e., any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session's callback with the arguments (hSession, CKN_SURRENDER, pApplication), where hSession is the session's handle and pApplication was supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the value CKR_OK (to indicate that Cryptoki should continue executing the function) or the value CKR_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation, if desired.

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

A Cryptoki library is not required to make any surrender callbacks.
5.21.2 Vendor-defined callbacks

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

### 6.1 RSA

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_RSA_PKCS_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_X9_31_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_PKCS_OAEP</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_9796</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_X_509</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_X9_31</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA224_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA224_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_RSA_X9_31</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_PKCS_TPM_1_1</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RSA_PKCS_TPM_1_1</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_256_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_384_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_RSA_PKCS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_256_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_384_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_RSA_PKCS_PSS</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 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
6.1.2 RSA public key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_MODULUS</td>
<td>Big integer</td>
<td>Modulus ( n )</td>
</tr>
<tr>
<td>CKA_MODULUS_BITS</td>
<td>CK_ULONG</td>
<td>Length in bits of modulus ( n )</td>
</tr>
<tr>
<td>CKA_PUBLIC_EXPONENT</td>
<td>Big integer</td>
<td>Public exponent ( e )</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.
Depending on the token, there may be limits on the length of key components. See PKCS #1 for more information on RSA keys.

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

```plaintext
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = “An RSA public key object”;
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
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_WRAP, &true, sizeof(true)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};
```

### 6.1.3 RSA private key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CKA_MODULUS</code>(^1,4,6)</td>
<td>Big integer</td>
<td>Modulus (n)</td>
</tr>
<tr>
<td><code>CKA_PUBLIC_EXPONENT</code>(^1,4,6)</td>
<td>Big integer</td>
<td>Public exponent (e)</td>
</tr>
<tr>
<td><code>CKA_PRIVATE_EXPONENT</code>(^1,4,6,7)</td>
<td>Big integer</td>
<td>Private exponent (d)</td>
</tr>
<tr>
<td><code>CKA_PRIME_1</code>(^1,4,6,7)</td>
<td>Big integer</td>
<td>Prime (p)</td>
</tr>
<tr>
<td><code>CKA_PRIME_2</code>(^4,6,7)</td>
<td>Big integer</td>
<td>Prime (q)</td>
</tr>
<tr>
<td><code>CKA_EXPONENT_1</code>(^1,4,6,7)</td>
<td>Big integer</td>
<td>Private exponent (d) modulo (p - 1)</td>
</tr>
<tr>
<td><code>CKA_EXPONENT_2</code>(^2,4,6,7)</td>
<td>Big integer</td>
<td>Private exponent (d) modulo (q - 1)</td>
</tr>
<tr>
<td><code>CKA_COEFFICIENT</code>(^4,6,7)</td>
<td>Big integer</td>
<td>CRT coefficient (q^{-1} \mod p)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

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

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 34 it keeps track of. Later, if an application asks for the values of the key’s various attributes, Cryptoki supplies values only for attributes whose values it can obtain (i.e., if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. E.g., if a particular token stores...
values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is certainly able to report values for all the attributes above (since they can all be computed efficiently from these three values). However, a Cryptoki implementation may or may not actually do this extra computation. The only attributes from Table 34 for which a Cryptoki implementation is required to be able to return values are **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT** and **CKA_PRIVATE_EXPONENT**. A token SHOULD also be able to return **CKA_PUBLIC_KEY_INFO** for an RSA private key.

If an RSA private key object is created on a token, and more attributes from Table 34 are supplied to the object creation call than are supported by the token, the extra attributes are likely to be thrown away. If an attempt is made to create an RSA private key object on a token with insufficient attributes for that particular token, then the object creation call fails and returns CKR_TEMPLATE_INCOMPLETE.

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

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

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = “An RSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE modulus[] = {...};
CK_BYTE publicExponent[] = {...};
CK_BYTE privateExponent[] = {...};
CK_BYTE prime1[] = {...};
CK_BYTE prime2[] = {...};
CK_BYTE exponent1[] = {...};
CK_BYTE exponent2[] = {...};
CK_BYTE coefficient[] = {...};
CK_BBOOL true = 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_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
    {CKA_PRIVATE_EXPONENT, privateExponent, sizeof(privateExponent)},
    {CKA_PRIME_1, prime1, sizeof(prime1)},
    {CKA_PRIME_2, prime2, sizeof(prime2)},
    {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
    {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
    {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
};
```
6.1.4 PKCS #1 RSA key pair generation

The PKCS #1 RSA key pair generation mechanism, denoted \texttt{CKM_RSA_PKCS_KEY_PAIR_GEN}, is a key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

It does not have a parameter.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the \texttt{CKA_MODULUS_BITS} and \texttt{CKA_PUBLIC_EXPONENT} attributes of the template for the public key. The \texttt{CKA_PUBLIC_EXPONENT} may be omitted in which case the mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).

Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by the token.

Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error if this attribute is omitted from the template. Experience has shown that many implementations of 2.11 and prior did allow the \texttt{CKA_PUBLIC_EXPONENT} attribute to be omitted from the template, and behaved as described above. The mechanism contributes the \texttt{CKA_CLASS}, \texttt{CKA_KEY_TYPE}, \texttt{CKA_MODULUS}, and \texttt{CKA_PUBLIC_EXPONENT} attributes to the new public key.

\texttt{CKA_PUBLIC_EXPONENT} will be copied from the template if supplied.

\texttt{CKR_TEMPLATE_INCONSISTENT} shall be returned if the implementation cannot use the supplied exponent value. It contributes the \texttt{CKA_CLASS} and \texttt{CKA_KEY_TYPE} attributes to the new private key; it may also contribute some of the following attributes to the new private key: \texttt{CKA_MODULUS}, \texttt{CKA_PUBLIC_EXPONENT}, \texttt{CKA_PRIVATE_EXPONENT}, \texttt{CKA_PRIME_1}, \texttt{CKA_PRIME_2}, \texttt{CKA_COEFFICIENT}. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

6.1.5 X9.31 RSA key pair generation

The X9.31 RSA key pair generation mechanism, denoted \texttt{CKM_RSA_X9_31_KEY_PAIR_GEN}, is a key pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

It does not have a parameter.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the \texttt{CKA_MODULUS_BITS} and \texttt{CKA_PUBLIC_EXPONENT} attributes of the template for the public key.

The mechanism contributes the \texttt{CKA_CLASS}, \texttt{CKA_KEY_TYPE}, \texttt{CKA_MODULUS}, and \texttt{CKA_PUBLIC_EXPONENT} attributes to the new public key. It contributes the \texttt{CKA_CLASS} and \texttt{CKA_KEY_TYPE} attributes to the new private key; it may also contribute some of the following attributes to the new private key: \texttt{CKA_MODULUS}, \texttt{CKA_PUBLIC_EXPONENT}, \texttt{CKA_PRIVATE_EXPONENT}, \texttt{CKA_PRIME_1}, \texttt{CKA_PRIME_2}, \texttt{CKA_COEFFICIENT}. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values. Unlike the \texttt{CKM_RSA_PKCS_KEY_PAIR_GEN} mechanism, this mechanism is guaranteed to generate \( p \) and \( q \) values, \texttt{CKA_PRIME_1} and \texttt{CKA_PRIME_2} respectively, that meet the strong primes requirement of X9.31.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

6.1.6 PKCS #1 v1.5 RSA

The PKCS #1 v1.5 RSA mechanism, denoted \texttt{CKM_RSA_PKCS}, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports...
single-part encryption and decryption; single-part signatures and verification with and without message
recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1
v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for
the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5.

This mechanism does not have a parameter.

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.

Constraints on key types and the length of the data are summarized in the following table. For
encryption, decryption, signatures and signature verification, the input and output data may begin at the
same location in memory. In the table, \( k \) is the length in bytes of the RSA modulus.

Table 35, PKCS #1 v1.5 RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>( \leq k-11 )</td>
<td>( k )</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-11 )</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>( \leq k-11 )</td>
<td>( k )</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>( \leq k-11 )</td>
<td>( k )</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>( \leq k-11 )</td>
<td>( k )</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>( k )</td>
<td>( \leq k-11 )</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k-11 )</td>
<td>( k )</td>
<td>block type 02</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-11 )</td>
<td>block type 02</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.
\(^2\) Data length, signature length.

For this mechanism, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the \texttt{CK_MECHANISM_INFO} structure
specify the supported range of RSA modulus sizes, in bits.

6.1.7 PKCS #1 RSA OAEP mechanism parameters

\* \texttt{CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR}

\texttt{CK_RSA_PKCS_MGF_TYPE} is used to indicate the Mask Generation Function (MGF) applied to a
message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the
PKCS #1 PSS signature scheme. It is defined as follows:

```c
typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;
```

The following MGFs are defined in PKCS #1. The following table lists the defined functions.

Table 36, PKCS #1 Mask Generation Functions
<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKG_MGF1_SHA1</td>
<td>0x00000001UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA224</td>
<td>0x00000005UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA256</td>
<td>0x00000002UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA384</td>
<td>0x00000003UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA512</td>
<td>0x00000004UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_224</td>
<td>0x00000006UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_256</td>
<td>0x00000007UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_384</td>
<td>0x00000008UL</td>
</tr>
<tr>
<td>CKG_MGF1_SHA3_512</td>
<td>0x00000009UL</td>
</tr>
</tbody>
</table>

**Source Identifier**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000001UL</td>
</tr>
</tbody>
</table>

**CK_RSA_PKCS_MGF_TYPE_PTR** is a pointer to a **CK_RSA_PKCS_MGF_TYPE**.

**CK_RSA_PKCS_OAEP_SOURCE_TYPE**;

**CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR**

**CK_RSA_PKCS_OAEP_SOURCE_TYPE** is used to indicate the source of the encoding parameter when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

```
typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```

The following encoding parameter sources are defined in PKCS #1. The following table lists the defined sources along with the corresponding data type for the `pSourceData` field in the **CK_RSA_PKCS_OAEP_PARAMS** structure defined below.

**Table 37, PKCS #1 RSA OAEP: Encoding parameter sources**

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKZ_DATA_SPECIFIED</td>
<td>0x00000001UL</td>
<td>Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, <code>pSourceData</code> must be NULL and <code>ulSourceDataLen</code> must be zero.</td>
</tr>
</tbody>
</table>

**CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR** is a pointer to a **CK_RSA_PKCS_OAEP_SOURCE_TYPE**.

**CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR**

**CK_RSA_PKCS_OAEP_PARAMS** is a structure that provides the parameters to the **CKM_RSA_PKCS_OAEP** mechanism. The structure is defined as follows:

```
typedef struct CK_RSA_PKCS_OAEP_PARAMS {
    CK_MECHANISM_TYPE hashAlg;
    CK_RSA_PKCS_MGF_TYPE mgf;
    CK_RSA_PKCS_OAEP_SOURCE_TYPE source;
    CK_VOID_PTR pSourceData;
    CK_ULONG ulSourceDataLen;
} CK_RSA_PKCS_OAEP_PARAMS;
```

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
source source of the encoding parameter
pSourceData data used as the input for the encoding parameter source
ulSourceDataLen length of the encoding parameter source input

CK_RSA_PKCS_OAEP_PARAMS_PTR is a pointer to a CK_RSA_PKCS_OAEP_PARAMS.

6.1.8 PKCS #1 RSA OAEP

The PKCS #1 RSA OAEP mechanism, denoted CKM_RSA_PKCS_OAEP, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1. It supports single-part encryption and decryption; key wrapping; and key unwrapping.

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.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, k is the length in bytes of the RSA modulus, and hLen is the output length of the message digest algorithm specified by the hashAlg field of the CK_RSA_PKCS_OAEP_PARAMS structure.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>RSA public key</td>
<td>( \leq k-2-2hLen )</td>
<td>k</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RSA private key</td>
<td>k</td>
<td>( \leq k-2-2hLen )</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k-2-2hLen )</td>
<td>k</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>k</td>
<td>( \leq k-2-2hLen )</td>
</tr>
</tbody>
</table>

1 Single-part operations only.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

6.1.9 PKCS #1 RSA PSS mechanism parameters

 بكشف عن معلمات الجهاز RSA PSS

CK_RSA_PKCS_PSS_PARAMS is a structure that provides the parameters to the CKM_RSA_PKCS_PSS mechanism. The structure is defined as follows:

```c
typedef struct CK_RSA_PKCS_PSS_PARAMS {
    CK_MECHANISM_TYPE hashAlg;
    CK_RSA_PKCS_MGF_TYPE mgf;
    CK_ULONG sLen;
} CK_RSA_PKCS_PSS_PARAMS;
```

The fields of the structure have the following meanings:
hashAlg hash algorithm used in the PSS encoding; if the signature mechanism does not include message hashing, then this value must be the mechanism used by the application to generate the message hash; if the signature mechanism includes hashing, then this value must match the hash algorithm indicated by the signature mechanism

mgf mask generation function to use on the encoded block

sLen length, in bytes, of the salt value used in the PSS encoding; typical values are the length of the message hash and zero

CK_RSA_PKCS_PSS_PARAMS_PTR is a pointer to a CK_RSA_PKCS_PSS_PARAMS.

6.1.10 PKCS #1 RSA PSS

The PKCS #1 RSA PSS mechanism, denoted CKM_RSA_PKCS_PSS, is a mechanism based on the RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part signature generation and verification without message recovery. This mechanism corresponds only to the part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash value on the message to be signed.

It has a parameter, a CK_RSA_PKCS_PSS_PARAMS structure. The sLen field must be less than or equal to \( k^* - 2 - hLen \) and \( hLen \) is the length of the input to the C_Sign or C_Verify function. \( k^* \) is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case \( k^* \) is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, \( k \) is the length in bytes of the RSA.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>( hLen )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>( hLen, k )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

6.1.11 ISO/IEC 9796 RSA

The ISO/IEC 9796 RSA mechanism, denoted CKM_RSA_9796, is a mechanism for single-part signatures and verification with and without message recovery based on the RSA public-key cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly, the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).

- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.
Constraints on key types and the length of input and output data are summarized in the following table.

In the table, \( k \) is the length in bytes of the RSA modulus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>( \leq \lfloor k/2 \rfloor )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>( \leq \lfloor k/2 \rfloor )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>( \leq \lfloor k/2 \rfloor )</td>
<td>N/A</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>( k )</td>
<td>( \leq \lfloor k/2 \rfloor )</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

For this mechanism, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the \text{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

### 6.1.12 X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted \text{CKM_RSA_X_509}, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. All these operations are based on so-called “raw” RSA, as assumed in X.509.

“Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first, applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant-byte first.

The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

This mechanism does not have a parameter.

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 \text{CKA_VALUE} attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other information about the key; the application must convey these separately, and supply them when unwrapping the key.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the sequence of plaintext bytes \( b_1 b_2 \ldots b_n (n \leq k) \), Cryptoki forms \( P = 2^{n-1} b_1 + 2^{n-2} b_2 + \ldots + b_n \). This number must be less than the RSA modulus. The \( k \)-byte ciphertext (\( k \) is the length in bytes of the RSA modulus) is produced by raising \( P \) to the RSA public exponent modulo the RSA modulus. Decryption of a \( k \)-byte ciphertext \( C \) is accomplished by raising \( C \) to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly \( k \) bytes. If the resulting plaintext is to be used to produce an unwrapped key, then however many bytes are specified in the template for the length of the key are taken from the end of this sequence of bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns \text{CKR_DATA_INVALID} (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and \text{CKR_ENCRYPTED_DATA_INVALID} (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

Constraints on key types and the length of input and output data are summarized in the following table.

In the table, \( k \) is the length in bytes of the RSA modulus.
Table 41, X.509 (Raw) RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>(\leq k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>(\leq k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>(\leq k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>(\leq k, k^2)</td>
<td>N/A</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>(k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>(\leq k)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>(k)</td>
<td>(\leq k) (specified in template)</td>
</tr>
</tbody>
</table>

1 Single-part operations only.
2 Data length, signature length.

For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

6.1.13 ANSI X9.31 RSA

The ANSI X9.31 RSA mechanism, denoted \texttt{CKM_RSA_X9_31}, is a mechanism for single-part signatures and verification without message recovery based on the RSA public-key cryptosystem and the block formats defined in ANSI X9.31.

This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must be applied by the application.

This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly, the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table.

In the table, \(k\) is the length in bytes of the RSA modulus. For all operations, the \(k\) value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 42, ANSI X9.31 RSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>RSA private key</td>
<td>(\leq k-2)</td>
<td>(k)</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>(\leq k-2, k^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

1 Single-part operations only.
2 Data length, signature length.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

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.

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 recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted CKM_SHA256_RSA_PKCS_PSS, CKM_SHA384_RSA_PKCS_PSS, and CKM_SHA512_RSA_PKCS_PSS respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a CK_RSA_PKCS_PSS_PARAMS structure. The sLen field must be less than or equal to \( k^* - 2 \cdot hLen \) where \( hLen \) is the length in bytes of the hash value. \( k^* \) is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case \( k^* \) is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, \( k \) is the length in bytes of the RSA modulus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>RSA private key</td>
<td>any</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RSA public key</td>
<td>any, ( k^2 )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

6.1.18 PKCS #1 v1.5 RSA signature with SHA3

The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted CKM_SHA3_224_RSA_PKCS, CKM_SHA3_256_RSA_PKCS, CKM_SHA3_384_RSA_PKCS, CKM_SHA3_512_RSA_PKCS respectively, performs similarly as the other CKM_SHA3_RSA_PKCS mechanisms but uses the corresponding SHA3 hash functions.

6.1.19 PKCS #1 RSA PSS signature with SHA3

The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted CKM_SHA3_224_RSA_PKCS_PSS, CKM_SHA3_256_RSA_PKCS_PSS, CKM_SHA3_384_RSA_PKCS_PSS, and CKM_SHA3_512_RSA_PKCS_PSS respectively, performs similarly as the other CKM_SHA3_RSA_PKCS_PSS mechanisms but uses the corresponding SHA-3 hash functions.

6.1.20 ANSI X9.31 RSA signature with SHA-1

The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted CKM_SHA1_RSA_X9_31, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in ANSI X9.31.

This mechanism does not have 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 all operations, the \( k \) value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

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 | \( k \) |
C_Verify | RSA public key | any, \( k^2 \) | N/A |

For these mechanisms, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

### 6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA

The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted \texttt{CKM_RSA_PKS\_TPM\_1\_1}, is a multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b.

Additional formatting rules remained the same in TCG TPM Specification 1.2 The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

This mechanism does not have a parameter. It differs from the standard PKCS#1 v1.5 RSA encryption mechanism in that the plaintext is wrapped in a TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure before being submitted to the PKCS#1 v1.5 encryption process. On encryption, the version field of the TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

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 \texttt{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 \texttt{CKA_CLASS} and \texttt{CKA_VALUE} (and \texttt{CKA_VALUE\_LEN}, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, \( k \) is the length in bytes of the RSA modulus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt\textsuperscript{1}</td>
<td>RSA public key</td>
<td>( \leq k-11-5 )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Decrypt\textsuperscript{1}</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-11-5 )</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k-11-5 )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-11-5 )</td>
</tr>
</tbody>
</table>

\textsuperscript{1} Single-part operations only.

For this mechanism, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

### 6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP

The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted \texttt{CKM_RSA_PKS\_OAEP\_TPM\_1\_1}, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.
This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption mechanism in that the plaintext is wrapped in a TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure before being submitted to the encryption process and that all of the values of the parameters that are passed to a standard CKM_RSA_PKCS_OAEP operation are fixed. On encryption, the version field of the TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

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 un Wrath of the mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the CKA_CLASS and CKA_VALUE (and CKA_VALUE_LEN, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, \( k \) is the length in bytes of the RSA modulus.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt(^1)</td>
<td>RSA public key</td>
<td>( \leq k-2-40-5 )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Decrypt(^1)</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-2-40-5 )</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RSA public key</td>
<td>( \leq k-2-40-5 )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RSA private key</td>
<td>( k )</td>
<td>( \leq k-2-40-5 )</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

For this mechanism, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the CK_MECHANISM_INFO structure specify the supported range of RSA modulus sizes, in bits.

### 6.1.23 RSA AES KEY WRAP

The RSA AES key wrap mechanism, denoted CKM_RSA_AES_KEY_WRAP, is a mechanism based on the RSA public-key cryptosystem and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

It has a parameter, a CK_RSA_AES_KEY_WRAP_PARAMS structure.

The mechanism can wrap and unwrap a target asymmetric key of any length and type using an RSA key.

- A temporary AES key is used for wrapping the target key using CKM_AES_KEY_WRAP_KWP mechanism.
- The temporary AES key is wrapped with the wrapping RSA key using CKM_RSA_PKCS_OAEP mechanism.

For wrapping, the mechanism -

- Generates a temporary random AES key of \( ulAESKeyBits \) length. This key is not accessible to the user - no handle is returned.
- Wraps the AES key with the wrapping RSA key using CKM_RSA_PKCS_OAEP with parameters of OAEPParams.
- Wraps the target key with the temporary AES key using CKM_AES_KEY_WRAP_KWP.
- Zeroizes the temporary AES key...
• Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES key, and the second is the wrapped target key.

The private target key will be encoded as defined in section 6.7.

The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

For unwrapping, the mechanism:

• 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.

• Un-wraps the temporary AES key from the first part with the private RSA key using CKM_RSA_PKCS_OAEP with parameters of OAEPParams.

• Un-wraps the target key from the second part with the temporary AES key using CKM_AES_KEY_WRAP_KWP.

• Zeroizes the temporary AES key.

• Returns the handle to the newly unwrapped target key.

Table 48, CKM_RSA_AES_KEY_WRAP Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt &amp; Decrypt</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Sign &amp; Verify</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Digest</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Gen. Key/Key Pair</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Wrap &amp; Unwrap</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Derive</td>
<td>CKM_RSA_AES_KEY_WRAP</td>
</tr>
</tbody>
</table>

1SR = SignRecover, VR = VerifyRecover

6.1.24 RSA AES KEY WRAP mechanism parameters

• CK_RSA_AES_KEY_WRAP_PARAMS; CK_RSA_AES_KEY_WRAP_PARAMS_PTR

CK_RSA_AES_KEY_WRAP_PARAMS is a structure that provides the parameters to the CKM_RSA_AES_KEY_WRAP mechanism. It is defined as follows:

```c
typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
    CK_ULONG ulAESKeyBits;
    CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPParams;
} CK_RSA_AES_KEY_WRAP_PARAMS;
```

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.
- pOAEPParams pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters.

CK_RSA_AES_KEY_WRAP_PARAMS_PTR is a pointer to a CK_RSA_AES_KEY_WRAP_PARAMS.
6.1.25 FIPS 186-4

When CKM_RSA_PKCS is operated in FIPS mode, the length of the modulus SHALL only be 1024, 2048, or 3072 bits.

6.2 DSA

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_DSA_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_PROBABILISTIC_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_FIPS_G_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA1</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA224</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA256</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA384</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA512</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA3_224</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA3_256</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA3_384</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DSA_SHA3_512</td>
<td>✓</td>
</tr>
</tbody>
</table>

6.2.1 Definitions

This section defines the key type “CKK_DSA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of DSA key objects.

Mechanisms:

- CKM_DSA_KEY_PAIR_GEN
- CKM_DSA
- 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
CKM_DSA_PARAMETER_GEN
CKM_DSA_PROBABILISTIC_PARAMETER_GEN
CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN
CKM_DSA_FIPS_G_GEN

**CK_DSA_PARAMETER_GEN_PARAM**

CK_DSA_PARAMETER_GEN_PARAM is a structure which provides and returns parameters for the NIST FIPS 186-4 parameter generating algorithms.

CK_DSA_PARAMETER_GEN_PARAM_PTR is a pointer to a CK_DSA_PARAMETER_GEN_PARAM.

```c
typedef struct CK_DSA_PARAMETER_GEN_PARAM {
    CK_MECHANISM_TYPE hash;
    CK_BYTE_PTR pSeed;
    CK_ULONG ulSeedLen;
    CK_ULONG ulIndex;
} CK_DSA_PARAMETER_GEN_PARAM;
```

The fields of the structure have the following meanings:

- **hash**: Mechanism value for the base hash used in PQG generation, Valid values are CKM_SHA_1, CKM_SHA224, CKM_SHA256, CKM_SHA384, CKM_SHA512.
- **pSeed**: Seed value used to generate PQ and G. This value is returned by CKM_DSA_PROBABILISTIC_PARAMETER_GEN, CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN, and passed into CKM_DSA_FIPS_G_GEN.
- **ulSeedLen**: Length of seed value.
- **ulIndex**: Index value for generating G. Input for CKM_DSA_FIPS_G_GEN. Ignored by CKM_DSA_PROBABILISTIC_PARAMETER_GEN and CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN.

### 6.2.2 DSA public key objects

DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DSA**) hold DSA public keys.

The following table defines the DSA public key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME,1,3</td>
<td>Big integer</td>
<td>Prime $p$ (512 to 3072 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME,1,3</td>
<td>Big integer</td>
<td>Subprime $q$ (160, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE,1,3</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_VALUE,1,4</td>
<td>Big integer</td>
<td>Public value $y$</td>
</tr>
</tbody>
</table>

* Refer to Table 11 for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.

The following is a sample template for creating a DSA public key object:
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = “A DSA public key object”;
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};

6.2.3 DSA Key Restrictions

FIPS PUB 186-4 specifies permitted combinations of prime and sub-prime lengths. They are:
- Prime: 1024 bits, Subprime: 160
- Prime: 2048 bits, Subprime: 224
- Prime: 2048 bits, Subprime: 256
- Prime: 3072 bits, Subprime: 256

Earlier versions of FIPS 186 permitted smaller prime lengths, and those are included here for backwards compatibility. An implementation that is compliant to FIPS 186-4 does not permit the use of primes of any length less than 1024 bits.

6.2.4 DSA private key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime q (160 bits, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private value x</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.

Note that when generating a DSA private key, the DSA domain parameters are not specified in the key’s template. This is because DSA private keys are only generated as part of a DSA key pair, and the DSA domain parameters for the pair are specified in the template for the DSA public key.
The following is a sample template for creating a DSA private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = “A DSA private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = 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_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.2.5 DSA domain parameter objects

DSA domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_DSA**) hold DSA domain parameters. The following table defines the DSA domain parameter object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime q (160 bits, 224 bits, or 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_PRIME_BITS</td>
<td>CK_UULONG</td>
<td>Length of the prime value.</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA domain parameters.

To ensure backwards compatibility, if **CKA_SUBPRIME_BITS** is not specified for a call to **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

The following is a sample template for creating a DSA domain parameter object:
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = “A DSA domain parameter object”;  
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
};

6.2.6 DSA key pair generation
The DSA key pair generation mechanism, denoted CKM_DSA_KEY_PAIR_GEN, is a key pair generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.
This mechanism does not have a parameter.
The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as specified in the CKA_PRIME, CKA_SUBPRIME, and CKA_BASE attributes of the template for the public key.
The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_SUBPRIME, CKA_BASE, and CKA_VALUE attributes to the new private key. Other attributes supported by the DSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

6.2.7 DSA domain parameter generation
The DSA domain parameter generation mechanism, denoted CKM_DSA_PARAMETER_GEN, is a domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.
This mechanism does not have a parameter.
The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in the CKA_PRIME_BITS attribute of the template.
The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_SUBPRIME, CKA_BASE and CKA_PRIME_BITS attributes to the new object. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.
For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

6.2.8 DSA probabilistic domain parameter generation
The DSA probabilistic domain parameter generation mechanism, denoted CKM_DSA_PROBABILISTIC_PARAMETER_GEN, is a domain parameter generation mechanism based
This mechanism takes a `CK_DSA_PARAMETER_GEN_PARAM` which supplies the base hash and returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime length in bits, as specified in the `CKA_PRIME_BITS` attribute of the template and the subprime length as specified in the `CKA_SUBPRIME_BITS` attribute of the template.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_PRIME`, `CKA_SUBPRIME`, `CKA_PRIME_BITS`, and `CKA_SUBPRIME_BITS` attributes to the new object. `CKA_BASE` is not set by this call. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of DSA prime sizes, in bits.

### 6.2.9 DSA Shawe-Taylor domain parameter generation

The DSA Shawe-Taylor domain parameter generation mechanism, denoted `CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN`, is a domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2 Construction and Validation of Provable Primes p and q.

This mechanism takes a `CK_DSA_PARAMETER_GEN_PARAM` which supplies the base hash and returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime length in bits, as specified in the `CKA_PRIME_BITS` attribute of the template and the subprime length as specified in the `CKA_SUBPRIME_BITS` attribute of the template.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_PRIME`, `CKA_SUBPRIME`, `CKA_PRIME_BITS`, and `CKA_SUBPRIME_BITS` attributes to the new object. `CKA_BASE` is not set by this call. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of DSA prime sizes, in bits.

### 6.2.10 DSA base domain parameter generation

The DSA base domain parameter generation mechanism, denoted `CKM_DSA_FIPS_G_GEN`, is a base parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.2 Generation of Generator G.

This mechanism takes a `CK_DSA_PARAMETER_GEN_PARAM` which supplies the base hash, the seed (pSeed) and the length (ulSeedLen) and the index value.

The mechanism generates the DSA base with the domain parameter specified in the `CKA_PRIME` and `CKA_SUBPRIME` attributes of the template.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_BASE` attributes to the new object. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of DSA prime sizes, in bits.

### 6.2.11 DSA without hashing

The DSA without hashing mechanism, denoted `CKM_DSA`, is a mechanism for single-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash value.)
For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values \( r \) and \( s \), each represented most-significant byte first.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

### Table 53, DSA: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>DSA private key</td>
<td>20, 28, 32, 48, or 64 bytes</td>
<td>2*length of subprime</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>DSA public key</td>
<td>(20, 28, 32, 48, or 64 bytes), (2*length of subprime)(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^{1}\) Single-part operations only.

\(^{2}\) Data length, signature length.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of DSA prime sizes, in bits.

### 6.2.12 DSA with SHA-1

The DSA with SHA-1 mechanism, denoted \texttt{CKM_DSA_SHA1}, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

This mechanism computes the entire DSA specification, including the hashing with SHA-1.

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values \( r \) and \( s \), each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

### Table 54, DSA with SHA-1: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^{2}\) Data length, signature length.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of DSA prime sizes, in bits.

### 6.2.13 FIPS 186-4

When \texttt{CKM_DSA} is operated in FIPS mode, only the following bit lengths of \( p \) and \( q \), represented by \( L \) and \( N \), SHALL be used:

- \( L = 1024, N = 160 \)
- \( L = 2048, N = 224 \)
- \( L = 2048, N = 256 \)
- \( L = 3072, N = 256 \)
6.2.14 DSA with SHA-224

The DSA with SHA-224 mechanism, denoted CKM_DSA_SHA224, 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 SHA-224.

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 55, DSA with SHA-224: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^{\text{subprime}}$ length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^{\text{subprime}}$ length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.

6.2.15 DSA with SHA-256

The DSA with SHA-256 mechanism, denoted CKM_DSA_SHA256, 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 SHA-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 56, DSA with SHA-256: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>$2^{\text{subprime}}$ length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, $2^{\text{subprime}}$ length$^2$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

$^2$ Data length, signature length.

6.2.16 DSA with SHA-384

The DSA with SHA-384 mechanism, denoted CKM_DSA_SHA384, 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 SHA-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 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  | N/A 

2 Data length, signature length.

### 6.2.17 DSA with SHA-512

The DSA with SHA-512 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. This mechanism computes the entire DSA specification, including the hashing with SHA-512.

For the purposes of this mechanism, a DSA signature is a string of length 2*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 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  | N/A 

2 Data length, signature length.

### 6.2.18 DSA with SHA3-224

The DSA with SHA3-224 mechanism, denoted CKM_DSA_SHA3_224, 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 SHA3-224.

For the purposes of this mechanism, a DSA signature is a string of length 2*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 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  | N/A 

2 Data length, signature length.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of DSA prime sizes, in bits.
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*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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length</td>
<td>N/A</td>
</tr>
</tbody>
</table>

2 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*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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length</td>
<td>N/A</td>
</tr>
</tbody>
</table>

2 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 SHA3-512.

For the purposes of this mechanism, a DSA signature is a string of length 2*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
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>DSA private key</td>
<td>any</td>
<td>2*subprime length</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 2*subprime length²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² Data length, signature length.

### 6.3 Elliptic Curve

The Elliptic Curve (EC) cryptosystem in this document was originally based on the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves were always represented in their Weierstrass form. Since that time, new curves represented in Edwards form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new curves, the EC cryptosystem in this document has been extended from the original. Additional key generation mechanisms have been added as well as an additional signature generation mechanism.

**Table 63, Elliptic Curve Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_EC_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_EDWARDS_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EC_MONTGOMERY_KEYPAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_224</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_256</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_384</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_512</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_EDDSA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_XEDDSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_COFACTOR_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECMQV_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mechanism</td>
<td>Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH_AES_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mechanism Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_EC_F_P</td>
<td>True if the mechanism can be used with EC domain parameters over $F_{p}$</td>
</tr>
<tr>
<td>CKF_EC_F_2M</td>
<td>True if the mechanism can be used with EC domain parameters over $F_{2^m}$</td>
</tr>
<tr>
<td>CKF_EC_ECPARAMETERS</td>
<td>True if the mechanism can be used with EC domain parameters of the choice ecParameters</td>
</tr>
<tr>
<td>CKF_EC_OID</td>
<td>True if the mechanism can be used with EC domain parameters of the choice oid</td>
</tr>
<tr>
<td>CKF_EC_UNCOMPRESS</td>
<td>True if the mechanism can be used with Elliptic Curve point uncompressed</td>
</tr>
<tr>
<td>CKF_EC_COMPRESS</td>
<td>True if the mechanism can be used with Elliptic Curve point compressed</td>
</tr>
<tr>
<td>CKF_EC_CURVENAME</td>
<td>True if the mechanism can be used with EC domain parameters of the choice curveName</td>
</tr>
</tbody>
</table>

Note: CKF_EC_NAMEDCURVE is deprecated with PKCS#11 3.00. It is replaced by CKF_EC_OID.

In these standards, there are two different varieties of EC defined:

1. EC using a field with an odd prime number of elements (i.e. the finite field $F_{p}$).
2. EC using a field of characteristic two (i.e. the finite field $F_{2^m}$).

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.

In these specifications there are also four representation methods to define the domain parameters for an EC key. Only the ecParameters, the oid 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 oid 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.

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 supporting the compressed form.
supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or
both of these flags for each EC mechanism.

Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
of domain parameters or one form may encounter difficulties achieving interoperability with other
implementations.

If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
attempt should fail with the error code CKR_CURVE_NOT_SUPPORTED. If an attempt to create,
generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If
an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

6.3.1 EC Signatures

For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at
most two times nLen octets, where nLen is the length in octets of the base point order n. The signature
octets correspond to the concatenation of the ECDSA values r and s, both represented as an octet string
of equal length of at most nLen with the most significant byte first. If r and s have different octet length,
the shorter of both must be padded with leading zero octets such that both have the same octet length.
Loosely spoken, the first half of the signature is r and the second half is s. For signatures created by a
token, the resulting signature is always of length 2nLen. For signatures passed to a token for verification,
the signature may have a shorter length but must be composed as specified before.

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
length of n will be used. Any truncation is done by the token.

Note: For applications, it is recommended to encode the signature as an octet string of length two times
nLen if possible. This ensures that the application works with PKCS#11 modules which have been
implemented based on an older version of this document. Older versions required all signatures to have
length two times nLen. It may be impossible to encode the signature with the maximum length of two
times nLen if the application just gets the integer values of r and s (i.e. without leading zeros), but does
not know the base point order n, because r and s can have any value between zero and the base point
order n.

An EdDSA signature is an octet string of even length which is two times nLen octets, where nLen is
calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of
the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length
of nLen bytes in little endian order.

6.3.2 Definitions

This section defines the key types "CKK_EC", "CKK_EC_EDWARDS" and "CKK_EC_MONTGOMERY"
for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Note: CKK_ECDSA is deprecated. It is replaced by CKK_EC.

Mechanisms:

<table>
<thead>
<tr>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_EC_KEY_PAIR_GEN</td>
</tr>
<tr>
<td>CKM_EC_EDWARDS_KEY_PAIR_GEN</td>
</tr>
<tr>
<td>CKM_EC_MONTGOMERY_KEY_PAIR_GEN</td>
</tr>
<tr>
<td>CKM_ECDSA</td>
</tr>
<tr>
<td>CKM_ECDSA_SHA1</td>
</tr>
<tr>
<td>CKM_ECDSA_SHA224</td>
</tr>
<tr>
<td>CKM_ECDSA_SHA256</td>
</tr>
<tr>
<td>CKM_ECDSA_SHA384</td>
</tr>
</tbody>
</table>
6.3.3 Short Weierstrass Elliptic Curve public key objects

Short Weierstrass EC public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:

Table 65, Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ECDSA_SHA512</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_224</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_256</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_384</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA_SHA3_512</td>
<td></td>
</tr>
<tr>
<td>CKM_EDDSA</td>
<td></td>
</tr>
<tr>
<td>CKM_XEDDSA</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH1_COFACTOR_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_ECMQV_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDH_AES_KEY_WRAP</td>
<td></td>
</tr>
<tr>
<td>CKD_NULL</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA1_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA224_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA384_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_224_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_384_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA1_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA224_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA256_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA384_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA512_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_224_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_384_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF_SP800</td>
<td></td>
</tr>
<tr>
<td>CKD_BLAKE2B_160_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_BLAKE2B_256_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_BLAKE2B_384_KDF</td>
<td></td>
</tr>
<tr>
<td>CKD_BLAKE2B_512_KDF</td>
<td></td>
</tr>
</tbody>
</table>
### Attribute Data type Meaning

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS(^1,3)</td>
<td>Byte array</td>
<td>DER-encoding of an ANSI X9.62 Parameters value</td>
</tr>
<tr>
<td>CKA_EC_POINT(^1,4)</td>
<td>Byte array</td>
<td>DER-encoding of ANSI X9.62 ECPoint value Q</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

Note: CKA_ECDSA_PARAMS is deprecated. It is replaced by CKA_EC_PARAMS.

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 with the following syntax:

```plaintext
Parameters ::= CHOICE {
  ecParameters  ECPARAMETERS,
  oId           CURVES.&id({CurveNames}),
  implicitlyCA  NULL,
  curveName     PrintableString
}
```

This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

The following is a sample template for creating an short Weierstrass EC public key object:

```plaintext
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = “An EC public key object”;
CK_BYTE ecParams[] = {...};
CK_BYTE ecPoint[] = {...};
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_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

### 6.3.4 Short Weierstrass Elliptic Curve private key objects

Short Weierstrass EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC**) hold EC private keys. See Section 6.3 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:

Table 66, Elliptic Curve Private Key Object Attributes
The `Attribute` data type Meaning

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS&lt;sup&gt;1,4,6&lt;/sup&gt;</td>
<td>Byte array</td>
<td>DER-encoding of an ANSI X9.62 Parameters value</td>
</tr>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>ANSI X9.62 private value d</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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 with the following syntax:

```
Parameters ::= CHOICE {
  ecParameters   ECParameters,
  oId             CURVES.&id({CurveNames}),
  implicitlyCA    NULL,
  curveName       PrintableString
}
```

This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki. Note that when generating an EC private key, the EC domain parameters are not specified in the key's template. This is because EC private keys are only generated as part of an EC key pair, and the EC domain parameters for the pair are specified in the template for the EC public key.

The following is a sample template for creating an short Weierstrass EC private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = "An EC private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
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_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
  {CKA_DERIVE, &true, sizeof(true)},
  {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_VALUE, value, sizeof(value)}
};
```

### 6.3.5 Edwards Elliptic Curve public key objects

Edwards EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC_EDWARDS**) hold Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in addition to the common attributes defined for this object class:
Table 67, Edwards Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_EC_POINT</td>
<td>Byte array</td>
<td>Public key bytes in little endian order as defined in RFC 8032</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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 4th choice is added to support Edwards and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```plaintext
Parameters ::= CHOICE {
  ecParameters  ECPARAMETERS,
  oId            CURVES.&id({CurveNames}),
  implicitlyCA   NULL,
  curveName     PrintableString
}
```

Edwards EC public keys only support the use of the curveName selection to specify a curve name as defined in [RFC 8032] and the use of the oID selection to specify a curve through an EdDSA algorithm as defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

The following is a sample template for creating an Edwards EC public key object with Edwards25519 being specified as curveName:

```plaintext
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
CK_UTF8CHAR label[] = "An Edwards EC public key object";
CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,
                      0x72, 0x64, 0x73, 0x32, 0x35, 0x31, 0x39};
CK_BYTE ecPoint[] = {...};
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_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

6.3.6 Edwards Elliptic Curve private key objects

Edwards EC private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC_EDWARDS) hold Edwards EC private keys. See Section 6.3 for more information about EC. The following table defines the Edwards EC private key object attributes, in addition to the common attributes defined for this object class:

Table 68, Edwards Elliptic Curve Private Key Object Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private key bytes in little endian order as defined in RFC 8032</td>
</tr>
</tbody>
</table>

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 4th choice is added to support Edwards and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```
Parameters ::= CHOICE {
  ecParameters  ECPARAMETERS,
  oId            CURVES.&id({CurveNames}),
  implicitlyCA   NULL,
  curveName      PrintableString
}
```

Edwards EC private keys only support the use of the curveName selection to specify a curve name as defined in [RFC 8032] and the use of the oID selection to specify a curve through an EdDSA algorithm as defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

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

The following is a sample template for creating an Edwards EC private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
CK_UTF8CHAR label[] = “An Edwards EC private key object”;  
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
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_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
  {CKA_DERIVE, &true, sizeof(true)},
  {CKA_VALUE, value, sizeof(value)}
};
```

### 6.3.7 Montgomery Elliptic Curve public key objects

Montgomery EC public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC_MONTGOMERY) hold Montgomery EC public keys. The following table defines the Montgomery EC public key attributes, in addition to the common attributes defined for this object class:
Table 69, Montgomery Elliptic Curve Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS(^{1,3})</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_EC_POINT(^{1,4})</td>
<td>Byte array</td>
<td>Public key bytes in little endian order as defined in RFC 7748</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.

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\(^{th}\) choice is added to support Edwards and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```plaintext
Parameters ::= CHOICE {
  ecParameters  ECPARAMETERS,
  oId            CURVES.&id({CurveNames}),
  implicitlyCA        NULL,
  curveName     PrintableString
}
```

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.

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

```plaintext
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
CK_UTF8CHAR label[] = "A Montgomery EC public key object";
CK_BYTE ecParams[] = {...};
CK_BYTE ecPoint[] = {...};
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_EC_PARAMS, ecParams, sizeof(ecParams)},
  {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

6.3.8 Montgomery Elliptic Curve private key objects

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:

Table 70, Montgomery Elliptic Curve Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_EC_PARAMS(^{1,4,6})</td>
<td>Byte array</td>
<td>DER-encoding of a Parameters value as defined above</td>
</tr>
<tr>
<td>CKA_VALUE(^{1,4,6,7})</td>
<td>Big integer</td>
<td>Private key bytes in little endian order as defined in RFC 7748</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.
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 4th choice is added to support Edwards and Montgomery Elliptic Curves. The CKA_EC_PARAMS attribute has the following syntax:

```
Parameters ::= CHOICE {
    ecParameters        ECPARAMETERS,
    oId                 CURVES.&id({CurveNames}),
    implicitlyCA        NULL,
    curveName           PrintableString
}
```

Montgomery EC private 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.

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
CK_UTF8CHAR label[] = "A Montgomery EC private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
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_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, &true, sizeof(value)}
};
```

### 6.3.9 Elliptic Curve key pair generation

The short Weierstrass ECkey pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is a key pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

The short Weierstrass EC key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS, is a key pair generation mechanism that uses the method defined by FIPS 186-4 Appendix B.4.1.

These mechanisms do not have a parameter.

These mechanisms generate EC public/private key pairs with particular EC domain parameters, as specified in the **CKA_EC_PARAMS** attribute of the template for the public key. Note that this version of Cryptoki does not include a mechanism for generating these EC domain parameters.
These mechanism contribute the `CA_CLASS`, `CA_KEY_TYPE`, and `CA_EC_POINT` attributes to the new public key and the `CA_CLASS`, `CA_KEY_TYPE`, `CA_EC_PARAMS` and `CA_VALUE` attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the 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 $2^{300}$ and $2^{300}$ elements, then `ulMinKeySize = 201` and `ulMaxKeySize = 301` (when written in binary notation, the number $2^{300}$ 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).

### 6.3.10 Edwards Elliptic Curve key pair generation

The Edwards EC key pair generation mechanism, denoted `CKM_EC_EDWARDS_KEY_PAIR_GEN`, is a key pair generation mechanism for EC keys over curves represented in Edwards form.

This mechanism does not have a parameter.

The mechanism can only generate EC public/private key pairs over the curves `edwards25519` and `edwards448` as defined in RFC 8032 or the curves `id-Ed25519` and `id-Ed448` as defined in RFC 8410.

These curves can only be specified in the `CA_EC_PARAMS` attribute of the template for the public key using the `curveName` or the `oID` methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with `CKR_CURVE_NOT_SUPPORTED`.

The mechanism contributes the `CA_CLASS`, `CA_KEY_TYPE`, and `CA_EC_POINT` attributes to the new public key and the `CA_CLASS`, `CA_KEY_TYPE`, `CA_EC_PARAMS` and `CA_VALUE` attributes to the new private key. Other attributes supported by the Edwards EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the `ulMinKeySize` and `ulMaxKeySize` fields accordingly.

### 6.3.11 Montgomery Elliptic Curve key pair generation

The Montgomery EC key pair generation mechanism, denoted `CKM_EC_MONTGOMERY_KEY_PAIR_GEN`, is a key pair generation mechanism for EC keys over curves represented in Montgomery form.

This mechanism does not have a parameter.

The mechanism can only generate Montgomery EC public/private key pairs over the curves `curve25519` and `curve448` as defined in RFC 7748 or the curves `id-X25519` and `id-X448` as defined in RFC 8410.

These curves can only be specified in the `CA_EC_PARAMS` attribute of the template for the public key using the `curveName` or old methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with `CKR_CURVE_NOT_SUPPORTED`.

The mechanism contributes the `CA_CLASS`, `CA_KEY_TYPE`, and `CA_EC_POINT` attributes to the new public key and the `CA_CLASS`, `CA_KEY_TYPE`, `CA_EC_PARAMS` and `CA_VALUE` attributes to the new private key. Other attributes supported by the Edwards EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the `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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign¹</td>
<td>CKK_EC private key</td>
<td>any³</td>
<td>2nLen¹</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>CKK_EC public key</td>
<td>any³, ≤2nLen²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

¹ Single-part operations only.
³ 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_EC private key</td>
<td>any</td>
<td>2nLen</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_EC public key</td>
<td>any, ≤2nLen²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² 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).
6.3.14 EdDSA

The EdDSA mechanism, denoted CKM_EDDSA, is a mechanism for single-part and multipart signatures and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in RFC 8032 and RFC 8410.

For curves according to RFC 8032, this mechanism has an optional parameter, a CK_EDDSA_PARAMS structure. The absence or presence of the parameter as well as its content is used to identify which signature scheme is to be used. The following table enumerates the five signature schemes defined in RFC 8032 and all supported permutations of the mechanism parameter and its content.

<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th>Mechanism Param</th>
<th>phFlag</th>
<th>Context Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ed25519</td>
<td>Not Required</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Ed25519ctx</td>
<td>Required</td>
<td>False</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed25519ph</td>
<td>Required</td>
<td>True</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed448</td>
<td>Required</td>
<td>False</td>
<td>Optional</td>
</tr>
<tr>
<td>Ed448ph</td>
<td>Required</td>
<td>True</td>
<td>Optional</td>
</tr>
</tbody>
</table>

For curves according to RFC 8410, the mechanism is implicitly given by the curve, which is EdDSA in pure mode.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_EC_EDWARDS private key</td>
<td>any</td>
<td>2bLen</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_EC_EDWARDS public key</td>
<td>any, ≤ 2bLen</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Note that for EdDSA in pure mode, Ed25519 and Ed448 the data must be processed twice. Therefore, a token might need to cache all the data, especially when used with C_SignUpdate/C_VerifyUpdate. If tokens are unable to do so they can return CKR_TOKEN_RESOURCE_EXCEEDED.

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 this mechanism, the only allowed values are 255 and 448 as RFC 8032 and RFC 8410 only define curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the ulMinKeySize and ulMaxKeySize fields accordingly.

6.3.15 XEdDSA

The XEdDSA mechanism, denoted CKM_XEDDSA, is a mechanism for single-part signatures and verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in [XEDDSA]. CKM_XEDDSA operates on CKK_EC_MONTGOMERY type EC keys, which allows these keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-term keys and also contributes to the DH key-exchange.

This mechanism has a parameter, a CK_XEDDSA_PARAMS structure.

Table 75, XEdDSA: Key and Data Length
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign²</td>
<td>CKK_EC_MONTGOMERY private key</td>
<td>any³</td>
<td>2b</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>CKK_EC_MONTGOMERY public key</td>
<td>any³, ≤2b²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

² 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 this mechanism, the only allowed values are 255 and 448 as [XEDDSA] only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the ulMinKeySize and ulMaxKeySize fields accordingly.

6.3.16 EC mechanism parameters

♦ CK_EDDSA_PARAMS, CK_EDDSA_PARAMS_PTR

CK_EDDSA_PARAMS is a structure that provides the parameters for the CKM_EDDSA signature mechanism. The structure is defined as follows:

```c
typedef struct CK_EDDSA_PARAMS {
    CK_BBOOL phFlag;
    CK_ULONG ulContextDataLen;
    CK_BYTE_PTR pContextData;
} CK_EDDSA_PARAMS;
```

The fields of the structure have the following meanings:

- phFlag: Boolean value which indicates if Prehashed variant of EdDSA should be used.
- pContextData: Context data shared between the signer and verifier.

CK_EDDSA_PARAMS_PTR is a pointer to a CK_EDDSA_PARAMS.

♦ CK_XEDDSA_PARAMS, CK_XEDDSA_PARAMS_PTR

CK_XEDDSA_PARAMS is a structure that provides the parameters for the CKM_XEDDSA signature mechanism. The structure is defined as follows:

```c
typedef struct CK_XEDDSA_PARAMS {
    CK_XEDDSA_HASH_TYPE hash;
} CK_XEDDSA_PARAMS;
```

The fields of the structure have the following meanings:

- hash: A Hash mechanism to be used by the mechanism.

CK_XEDDSA_PARAMS_PTR is a pointer to a CK_XEDDSA_PARAMS.

♦ CK_XEDDSA_HASH_TYPE, CK_XEDDSA_HASH_TYPE_PTR

CK_XEDDSA_HASH_TYPE is used to indicate the hash function used in XEDDSA. It is defined as follows:

```c
typedef CK_ULONG CK_XEDDSA_HASH_TYPE;
```
The following table lists the defined functions.

Table 76, EC: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_256</td>
</tr>
<tr>
<td>CKM_BLAKE2B_512</td>
</tr>
<tr>
<td>CKM_SHA3_256</td>
</tr>
<tr>
<td>CKM_SHA3_512</td>
</tr>
<tr>
<td>CKM_SHA256</td>
</tr>
<tr>
<td>CKM_SHA512</td>
</tr>
</tbody>
</table>

CK_XEDDSA_HASH_TYPE_PTR is a pointer to a CK_XEDDSA_HASH_TYPE.

CK_EC_KDF_TYPE is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is defined as follows:

typedef CK_ULONG CK_EC_KDF_TYPE;

The following table lists the defined functions.

Table 77, EC: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_SHA1_KDF</td>
</tr>
<tr>
<td>CKD_SHA224_KDF</td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
</tr>
<tr>
<td>CKD_SHA384_KDF</td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_224_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_384_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA224_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA256_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA384_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA512_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA3_224_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA3_384_KDF_SP800</td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF_SP800</td>
</tr>
</tbody>
</table>
The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key derivation function.

The key derivation functions **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**, which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

The key derivation functions **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section 5.8.1.1.

The key derivation functions **CKD_BLAKE2B_[160|256|384|512]_KDF**, which are based on the Blake2b family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section 5.8.1.1. **CK_EC_KDF_TYPE_PTR** is a pointer to a **CXK_EC_KDF_TYPE**.

### **CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

**CK_ECDH1_DERIVE_PARAMS** is a structure that provides the parameters for the **CKM_ECDH1_DERIVE** and **CKM_ECDH1_COFACTOR_DERIVE** key derivation mechanisms, where each party contributes one key pair. The structure is defined as follows:

```c
typedef struct CK_ECDH1_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen;
    CK_BYTE_PTR pSharedData;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK_ECDH1_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf**: key derivation function used on the shared secret value
- **ulSharedDataLen**: the length in bytes of the shared info
- **pSharedData**: some data shared between the two parties
- **ulPublicDataLen**: the length in bytes of the other party’s EC public key
PublicData\(^1\) pointer to other party’s EC public key value. For short Weierstrass EC keys: a token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA_EC_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form. For Montgomery keys: the public key is provided as bytes in little endian order as defined in RFC 7748.

With the key derivation function \(\text{CKD\_NULL}\), \(\text{pSharedData}\) must be NULL and \(\text{ulSharedDataLen}\) must be zero. With the key derivation functions \(\text{CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512\]_KDF}\), \(\text{CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512\]_KDF\_SP800}\), an optional \(\text{pSharedData}\) may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, \(\text{pSharedData}\) must be NULL and \(\text{ulSharedDataLen}\) must be zero.

\(\text{CK\_ECDH1\_DERIVE\_PARAMS\_PTR}\) is a pointer to a \(\text{CK\_ECDH1\_DERIVE\_PARAMS}\).

\(\ast\) \(\text{CK\_ECDH2\_DERIVE\_PARAMS}, \text{CK\_ECDH2\_DERIVE\_PARAMS\_PTR}\)

\(\text{CK\_ECDH2\_DERIVE\_PARAMS}\) is a structure that provides the parameters to the \(\text{CKM\_ECMQV\_DERIVE}\) key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_ECDH2_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen;
    CK_BYTE_PTR pSharedData;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_BYTE_PTR pPublicData2;
} CK_ECDH2_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- \(\text{kdf}\) key derivation function used on the shared secret value
- \(\text{ulSharedDataLen}\) the length in bytes of the shared info
- \(\text{pSharedData}\) some data shared between the two parties
- \(\text{ulPublicDataLen}\) the length in bytes of the other party’s first EC public key
- \(\text{pPublicData}\) pointer to other party’s first EC public key value. Encoding rules are as per \(\text{pPublicData}\) of \(\text{CK\_ECDH1\_DERIVE\_PARAMS}\)
- \(\text{ulPrivateDataLen}\) 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.
hPrivateData     key handle for second EC private key value
ulPublicDataLen2  the length in bytes of the other party’s second EC public key
pPublicData2     pointer to other party’s second EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

With the key derivation function CKD_NULL, pSharedData must be NULL and ulSharedDataLen must be zero. With the key derivation function CKD_SHA1_KDF, an optional pSharedData may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, pSharedData must be NULL and ulSharedDataLen must be zero.

CK_ECDH2_DERIVE_PARAMS_PTR is a pointer to a CK_ECDH2_DERIVE_PARAMS.

**CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR**

CK_ECMQV_DERIVE_PARAMS is a structure that provides the parameters to the CKM_ECMQV_DERIVE key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_ECMQV_DERIVE_PARAMS {
    CK_EC_KDF_TYPE         kdf;
    CK_ULONG               ulSharedDataLen;
    CK_BYTE_PTR            pSharedData;
    CK_ULONG               ulPublicDataLen;
    CK_BYTE_PTR            pPublicData;
    CK_ULONG               ulPrivateDataLen;
    CK_OBJECT_HANDLE       hPrivateData;
    CK_ULONG               ulPublicDataLen2;
    CK_BYTE_PTR            pPublicData2;
    CK_OBJECT_HANDLE       publicKey;
} CK_ECMQV_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf** key derivation function used on the shared secret value
- **ulSharedDataLen** the length in bytes of the shared info
- **pSharedData** some data shared between the two parties
- **ulPublicDataLen** the length in bytes of the other party’s first EC public key
- **pPublicData** pointer to other party’s first EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS
- **ulPrivateDataLen** the length in bytes of the second EC private key
- **hPrivateData** key handle for second EC private key value
- **ulPublicDataLen2** the length in bytes of the other party’s second EC public key
- **pPublicData2** pointer to other party’s second EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS
- **publicKey** Handle to the first party’s ephemeral public key

With the key derivation function CKD_NULL, pSharedData must be NULL and ulSharedDataLen must be zero. With the key derivation functions CKD_[SHA1|SHA224|SHA384|SHA512]|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF, CKD_[SHA1|SHA224|SHA384|SHA512]|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800, an
optional $p_{SharedData}$ may be supplied, which consists of some data shared by the two parties intending
to share the shared secret. Otherwise, $p_{SharedData}$ must be NULL and $ul_{SharedDataLen}$ must be zero.

CK_ECMQV_DERIVE_PARAMS_PTR is a pointer to a CK_ECMQV_DERIVE_PARAMS structure.

6.3.17 Elliptic Curve Diffie-Hellman key derivation

The Elliptic Curve Diffie-Hellman (ECDH) key derivation mechanism, denoted CKM_ECDH1_DERIVE, is
a mechanism for key derivation based on the Diffie-Hellman version of the Elliptic Curve key agreement
scheme, as defined in ANSI X9.63 for short Weierstrass EC keys and RFC 7748 for Montgomery keys,
where each party contributes one key pair all using the same EC domain parameters.

It has a parameter, a CK_ECDH1_DERIVE_PARAMS structure.

This mechanism derives a secret value, and truncates the result according to the CKA_VALUE
attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of
the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key
type must be specified in the template.

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.

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 EC using a field of characteristic 2 which has between $2^{300}$
and $2^{300}$ elements, then ulMinKeySize = 201 and ulMaxKeySize = 301 (when written in binary notation,
the number $2^{300}$ 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).

Constraints on key types are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC or CKK_EC_MONTGOMERY</td>
</tr>
</tbody>
</table>

6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation

The Elliptic Curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted
CKM_ECDH1_COFACTOR_DERIVE, is a mechanism for key derivation based on the cofactor Diffie-Hellman
version of the Elliptic Curve key agreement scheme, as defined in ANSI X9.63, where each party
contributes one key pair all using the same EC domain parameters. Cofactor multiplication is
computationally efficient and helps to prevent security problems like small group attacks.

It has a parameter, a CK_ECDH1_DERIVE_PARAMS structure.

This mechanism derives a secret value, and truncates the result according to the CKA_KEY_TYPE
attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of
the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key
type must be specified in the template.
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.

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 EC using a field of characteristic 2 which has between $2^{300}$ and $2^{300}$ elements, then `ulMinKeySize` = 201 and `ulMaxKeySize` = 301 (when written in binary notation, the number $2^{300}$ 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).

Constraints on key types are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC</td>
</tr>
</tbody>
</table>

### 6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation

The Elliptic Curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted **CKM_ECMQV_DERIVE**, is a mechanism for key derivation based on the MQV version of the Elliptic Curve key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using the same EC domain parameters.

It has a parameter, a **CK_ECMQV_DERIVE_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

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.

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 EC using a field of characteristic 2 which has between $2^{300}$ 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).

Constraints on key types are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Derive</td>
<td>CKK_EC</td>
</tr>
</tbody>
</table>

### Table 80: ECDH MQV: Allowed Key Types

#### 6.3.20 ECDH AES KEY WRAP

The ECDH AES KEY WRAP mechanism, denoted **CKM_ECDH_AES_KEY_WRAP**, is a mechanism based on Elliptic Curve public-key crypto-system and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

It has a parameter, a **CK_ECDH_AES_KEY_WRAP_PARAMS** structure.

The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC key.

For wrapping, the mechanism -

- A temporary AES key is derived from a temporary EC key and the wrapping EC key using the **CKM_ECDH1_DERIVE** mechanism.
- The derived AES key is used for wrapping the target key using the **CKM_AES_KEY_WRAP_KWP** mechanism.

For unwrapping, the mechanism -

- Splits the input into two parts. The first part is the public key material of the transport key and the second part is the wrapped target key. The length of the first part is equal to the length of the public key material of the unwrapping EC key.

Note: since the transport key and the wrapping EC key share the same domain, the length of the public key material of the transport key is the same length of the public key material of the unwrapping EC key.

- Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.
- Un-wraps the target key from the second part with the temporary AES key using `CKM_AES_KEY_WRAP_KWP`.

- Zeroizes the temporary AES key.

### Table 81, CKM_ECDH_AES_KEY_WRAP Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ECDH_AES_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

¹SR = SignRecover, VR = VerifyRecover

Constraints on key types are summarized in the following table:

### Table 82, ECDH AES Key Wrap: Allowed Key Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Wrap / C_Unwrap</td>
<td>CKK_EC or CKK_EC_MONTGOMERY</td>
</tr>
</tbody>
</table>

### 6.3.21 ECDH AES KEY WRAP mechanism parameters

- `CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR`

`CK_ECDH_AES_KEY_WRAP_PARAMS` is a structure that provides the parameters to the `CKM_ECDH_AES_KEY_WRAP` mechanism. It is defined as follows:

```c
typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
    CK_ULONG ulAESKeyBits;
    CK_EC_KDF_TYPE kdf;
    CK_ULONG ulSharedDataLen;
    CK_BYTE_PTR pSharedData;
} CK_ECDH_AES_KEY_WRAP_PARAMS;
```

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.
- `kdf` key derivation function used on the shared secret value to generate AES key.
- `ulSharedDataLen` the length in bytes of the shared info
- `pSharedData` Some data shared between the two parties

`CK_ECDH_AES_KEY_WRAP_PARAMS_PTR` is a pointer to a `CK_ECDH_AES_KEY_WRAP_PARAMS`. 
6.3.22 FIPS 186-4

When CKM_ECDSA is operated in FIPS mode, the curves SHALL either be NIST recommended curves (with a fixed set of domain parameters) or curves with domain parameters generated as specified by ANSI X9.64. The NIST recommended curves are:

- P-192, P-224, P-256, P-384, P-521
- K-163, B-163, K-233, B-233
- K-283, B-283, K-409, B-409
- K-571, B-571

6.4 Diffie-Hellman

Table 83, Diffie-Hellman Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DH_PKCS_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DH_PKCS_PARAMETER_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DH_PKCS_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_PARAMETER_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_HYBRID_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_MQV_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.4.1 Definitions

This section defines the key type “CKK_DH” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of [DH] key objects.

Mechanisms:

- 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
6.4.2 Diffie-Hellman public key objects

Diffie-Hellman public key objects (object class CKO_PUBLIC_KEY, key type CKK_DH) hold Diffie-Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 84, Diffie-Hellman Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Public value y</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The CKA_PRIME and CKA_BASE attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See [PKCS #3] for more information on Diffie-Hellman keys.

The following is a sample template for creating a Diffie-Hellman public key object:

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = “A Diffie-Hellman public key object”;
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.4.3 X9.42 Diffie-Hellman public key objects

X9.42 Diffie-Hellman public key objects (object class CKO_PUBLIC_KEY, key type CKK_X9_42_DH) hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 85, X9.42 Diffie-Hellman Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p (≥ 1024 bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime q (≥ 160 bits)</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Public value y</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The CKA_PRIME, CKA_BASE and CKA_SUBPRIME attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.
The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
};
```

### 6.4.4 Diffie-Hellman private key objects

Diffie-Hellman private key objects (object class `CKO_PRIVATE_KEY`, key type `CKK_DH`) hold Diffie-Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME[^1^4^6]</td>
<td>Big integer</td>
<td>Prime $p$</td>
</tr>
<tr>
<td>CKA_BASE[^1^4^6]</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_VALUE[^1^4^6^7]</td>
<td>Big integer</td>
<td>Private value $x$</td>
</tr>
<tr>
<td>CKA_VALUE_BITS[^2^6]</td>
<td>CK_ULONG</td>
<td>Length in bits of private value $x$</td>
</tr>
</tbody>
</table>

[^1]: Refer to Table 11 for footnotes
[^2]: The `CKA_PRIME` and `CKA_BASE` attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See [PKCS #3] for more information on Diffie-Hellman keys.
[^4]: Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are not specified in the key’s template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-Hellman key pair, and the Diffie-Hellman parameters for the pair are specified in the template for the Diffie-Hellman public key.
[^6]: The following is a sample template for creating a Diffie-Hellman private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
```
CK_BYTE value[] = {...};
CK_BBOOL true = 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_SUBJECT, subject, sizeof(subject)},
{CKA_ID, id, sizeof(id)},
{CKA_SENSITIVE, &true, sizeof(true)},
{CKA_DERIVE, &true, sizeof(true)},
{CKA_PRIME, prime, sizeof(prime)},
{CKA_BASE, base, sizeof(base)},
{CKA_VALUE, value, sizeof(value)}
};

6.4.5 X9.42 Diffie-Hellman private key objects

X9.42 Diffie-Hellman private key objects (object class CKO_PRIVATE_KEY, key type CKK_X9_42_DH) hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME</td>
<td>Big integer</td>
<td>Prime p (≥ 1024 bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_SUBPRIME</td>
<td>Big integer</td>
<td>Subprime q (≥ 160 bits)</td>
</tr>
<tr>
<td>CKA_VALUE</td>
<td>Big integer</td>
<td>Private value x</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The CKA_PRIME, CKA_BASE and CKA_SUBPRIME attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain parameters are not specified in the key's template. This is because X9.42 Diffie-Hellman private keys are only generated as part of a X9.42 Diffie-Hellman key pair, and the X9.42 Diffie-Hellman domain parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = “A X9.42 Diffie-Hellman private key object”;
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
{CKA_CLASS, &class, sizeof(class)},
```
6.4.6 Diffie-Hellman domain parameter objects

Diffie-Hellman domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_DH) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain parameter object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME^{1,4}</td>
<td>Big integer</td>
<td>Prime p</td>
</tr>
<tr>
<td>CKA_BASE^{1,4}</td>
<td>Big integer</td>
<td>Base g</td>
</tr>
<tr>
<td>CKA_PRIME_BITS^{2,3}</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
</tbody>
</table>

The CKA_PRIME and CKA_BASE attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See [PKCS #3] for more information on Diffie-Hellman domain parameters.

The following is a sample template for creating a Diffie-Hellman domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
};
```

6.4.7 X9.42 Diffie-Hellman domain parameters objects

X9.42 Diffie-Hellman domain parameters objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_X9_42_DH) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42
Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime (p \geq 1024) bits, in steps of 256 bits</td>
</tr>
<tr>
<td>CKA_BASE&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Base (g)</td>
</tr>
<tr>
<td>CKA_SUBPRIME&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Subprime (q \geq 160) bits</td>
</tr>
<tr>
<td>CKA_PRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
<tr>
<td>CKA_SUBPRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the subprime value.</td>
</tr>
</tbody>
</table>

Table 89, X9.42 Diffie-Hellman Domain Parameters Object Attributes

Refer to Table 11 for footnotes.

The CKA_PRIME, CKA_BASE and CKA_SUBPRIME attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the domain parameters components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman domain parameters.

The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain parameters object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
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_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
};
```

### 6.4.8 PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted CKM_DH_PKCS_KEY_PAIR_GEN, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in [PKCS #3-1]. This is what PKCS #3 calls “phase I”. It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the CKA_PRIME and CKA_BASE attributes of the template for the public key. If the CKA_VALUE_BITS attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in [PKCS #3-1].

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_PRIME, CKA_BASE, and CKA_VALUE (and the CKA_VALUE_BITS attribute, if it is not already provided in the template) attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.
For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of Diffie-Hellman prime sizes, in bits.

### 6.4.9 PKCS #3 Diffie-Hellman domain parameter generation

The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted `CKM_DH_PKCS_PARAMETER_GEN`, is a domain parameter generation mechanism based on Diffie-Hellman key agreement, as defined in [PKCS #3].

It does not have a parameter.

The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as specified in the `CKA_PRIME_BITS` attribute of the template.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_PRIME`, `CKA_BASE`, and `CKA_PRIME_BITS` attributes to the new object. Other attributes supported by the Diffie-Hellman domain parameter types may also be specified in the template, 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 Diffie-Hellman prime sizes, in bits.

### 6.4.10 PKCS #3 Diffie-Hellman key derivation

The PKCS #3 Diffie-Hellman key derivation mechanism, denoted `CKM_DH_PKCS_DERIVE`, is a mechanism for key derivation based on Diffie-Hellman key agreement, as defined in [PKCS #3]. This is what PKCS #3 calls “phase II”.

It has a parameter, which is the public value of the other party in the key agreement protocol, represented as a Cryptoki “Big integer” (i.e., a sequence of bytes, most-significant byte first).

This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other party. It computes a Diffie-Hellman secret value from the public value and private key according to [PKCS #3], and truncates the result according to the `CKA_KEY_TYPE` attribute of the template and, if it has one and the key type supports it, the `CKA_VALUE_LEN` attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the `CKA_VALUE` attribute of the new key; other attributes required by the key type must be specified in the template.

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.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of Diffie-Hellman prime sizes, in bits.

---

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`. 

---

pkcs11-spec-v3.1-cs01
Standards Track Work Product Copyright © OASIS Open 2022. All Rights Reserved.
6.4.11 X9.42 Diffie-Hellman mechanism parameters

✓ **CK_X9_42_DH_KDF_TYPE, CK_X9_42_DH_KDF_TYPE_PTR**

**CK_X9_42_DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X9.42 Diffie-Hellman key agreement schemes. It is defined as follows:

```c
typedef CK_ULONG CK_X9_42_DH_KDF_TYPE;
```

The following table lists the defined functions.

**Table 90, X9.42 Diffie-Hellman Key Derivation Functions**

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_ASN1</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_CONCATENATE</td>
</tr>
</tbody>
</table>

The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key derivation function whereas the key derivation functions **CKD_SHA1_KDF_ASN1** and **CKD_SHA1_KDF_CONCATENATE**, which are both based on SHA-1, derive keying data from the shared secret value as defined in the ANSI X9.42 standard.

**CK_X9_42_DH_KDF_TYPE_PTR** is a pointer to a **CK_X9_42_DH_KDF_TYPE**.

✓ **CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR**

**CK_X9_42_DH1_DERIVE_PARAMS** is a structure that provides the parameters to the key derivation mechanism, where each party contributes one key pair. The structure is defined as follows:

```c
typedef struct CK_X9_42_DH1_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK_X9_42_DH1_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- kdf: key derivation function used on the shared secret value
- ulOtherInfoLen: the length in bytes of the other info
- pOtherInfo: some data shared between the two parties
- ulPublicDataLen: the length in bytes of the other party’s X9.42 Diffie-Hellman public key
- pPublicData: pointer to other party’s X9.42 Diffie-Hellman public key value

- With the key derivation function **CKD_NULL**, **pOtherInfo** must be NULL and **ulOtherInfoLen** must be zero.
- With the key derivation function **CKD_SHA1_KDF_ASN1**, **pOtherInfo** must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional **pOtherInfo** may be supplied, which consists of some
data shared by the two parties intending to share the shared secret. Otherwise, pOtherInfo must be NULL and ulOtherInfoLen must be zero.

CK_X9_42_DH1_DERIVE_PARAMS_PTR is a pointer to a CK_X9_42_DH1_DERIVE_PARAMS.

- CK_X9_42_DH2ДЕRIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR

CK_X9_42_DH2_DERIVE_PARAMS is a structure that provides the parameters to the CKM_X9_42_DH_HYBRID_DERIVE and CKM_X9_42_MQV_DERIVE key derivation mechanisms, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_X9_42_DH2_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_BYTE_PTR pPublicData2;
} CK_X9_42_DH2_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- `kdf` key derivation function used on the shared secret value
- `ulOtherInfoLen` the length in bytes of the other info
- `pOtherInfo` some data shared between the two parties
- `ulPublicDataLen` the length in bytes of the other party’s first X9.42 Diffie-Hellman public key
- `pPublicData` pointer to other party’s first X9.42 Diffie-Hellman public key value
- `ulPrivateDataLen` the length in bytes of the second X9.42 Diffie-Hellman private key
- `hPrivateData` key handle for second X9.42 Diffie-Hellman private key value
- `ulPublicDataLen2` the length in bytes of the other party’s second X9.42 Diffie-Hellman public key
- `pPublicData2` pointer to other party’s second X9.42 Diffie-Hellman public key value

With the key derivation function CKD_NULL, pOtherInfo must be NULL and ulOtherInfoLen must be zero.

With the key derivation function CKD_SHA1_KDF_ASN1, pOtherInfo must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function CKD_SHA1_KDF_CONCATENATE, an optional pOtherInfo may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, pOtherInfo must be NULL and ulOtherInfoLen must be zero.

CK_X9_42_DH2_DERIVE_PARAMS_PTR is a pointer to a CK_X9_42_DH2_DERIVE_PARAMS.
• **CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR**

**CK_X9_42_MQV_DERIVE_PARAMS** is a structure that provides the parameters to the **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```c
typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE kdf;
    CK_ULONG ulOtherInfoLen;
    CK_BYTE_PTR pOtherInfo;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPrivateDataLen;
    CK_OBJECT_HANDLE hPrivateData;
    CK_ULONG ulPublicDataLen2;
    CK_BYTE_PTR pPublicData2;
    CK_OBJECT_HANDLE publicKey;
} CK_X9_42_MQV_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **kdf** key derivation function used on the shared secret value
- **ulOtherInfoLen** the length in bytes of the other info
- **pOtherInfo** some data shared between the two parties
- **ulPublicDataLen** the length in bytes of the other party’s first X9.42 Diffie-Hellman public key
- **pPublicData** pointer to other party’s first X9.42 Diffie-Hellman public key value
- **ulPrivateDataLen** the length in bytes of the second X9.42 Diffie-Hellman private key
- **hPrivateData** key handle for second X9.42 Diffie-Hellman private key value
- **ulPublicDataLen2** the length in bytes of the other party’s second X9.42 Diffie-Hellman public key
- **pPublicData2** pointer to other party’s second X9.42 Diffie-Hellman public key value
- **publicKey** Handle to the first party’s ephemeral public key

With the key derivation function **CKD_NULL**, **pOtherInfo** must be NULL and **ulOtherInfoLen** must be zero.

With the key derivation function **CKD_SHA1_KDF_ASN1**, **pOtherInfo** must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional **pOtherInfo** may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, **pOtherInfo** must be NULL and **ulOtherInfoLen** must be zero.

**CK_X9_42_MQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_MQV_DERIVE_PARAMS**.

### 6.4.12 X9.42 Diffie-Hellman key pair generation

The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM_X9_42_DH_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

It does not have a parameter.
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.
For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the \texttt{CK\_PRIME} attribute.

### 6.4.15 X9.42 Diffie-Hellman hybrid key derivation

The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted \texttt{CK\_X9\_42\_DH\_HYBRID\_DERIVE}, is a mechanism for key derivation based on the Diffie-Hellman hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a \texttt{CK\_X9\_42\_DH\_DERIVE\_PARAMS} structure.

This mechanism derives a secret value, and truncates the result according to the \texttt{CK\_KEY\_TYPE} attribute of the template and, if it has one and the key type supports it, the \texttt{CK\_VALUE\_LEN} attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the \texttt{CK\_VALUE} attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the \texttt{CK\_VALUE} attribute as the key of a general-length MAC mechanism (e.g. \texttt{CK\_SHA\_1\_HMAC\_GENERAL}) over some test data.

This mechanism has the following rules about key sensitivity and extractability:

- The \texttt{CK\_SENSITIVE} and \texttt{CK\_EXTRACTABLE} attributes in the template for the new key can both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on some default value.
- If the base key has its \texttt{CK\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key will as well. If the base key has its \texttt{CK\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CK\_ALWAYS\_SENSITIVE} attribute set to the same value as its \texttt{CK\_SENSITIVE} attribute.
- Similarly, if the base key has its \texttt{CK\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, then the derived key will, too. If the base key has its \texttt{CK\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CK\_NEVER\_EXTRACTABLE} attribute set to the \texttt{opposite} value from its \texttt{CK\_EXTRACTABLE} attribute.

For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the \texttt{CK\_PRIME} attribute.

### 6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted \texttt{CK\_X9\_42\_MQV\_DERIVE}, is a mechanism for key derivation based the MQV scheme, as defined in the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a \texttt{CK\_X9\_42\_MQV\_DERIVE\_PARAMS} structure.

This mechanism derives a secret value, and truncates the result according to the \texttt{CK\_KEY\_TYPE} attribute of the template and, if it has one and the key type supports it, the \texttt{CK\_VALUE\_LEN} attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the \texttt{CK\_VALUE} attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the \texttt{CK\_VALUE} attribute as the key of a general-length MAC mechanism (e.g. \texttt{CK\_SHA\_1\_HMAC\_GENERAL}) over some test data.

This mechanism has the following rules about key sensitivity and extractability:

- The \texttt{CK\_SENSITIVE} and \texttt{CK\_EXTRACTABLE} attributes in the template for the new key can both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on some default value.
- If the base key has its \texttt{CK\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key will as well. If the base key has its \texttt{CK\_ALWAYS\_SENSITIVE} attribute set to \texttt{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 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the `CKA_PRIME` attribute.

### 6.5 Extended Triple Diffie-Hellman (x3dh)

The Extended Triple Diffie-Hellman mechanism described here is the one described in [SIGNAL].

**Table 91, Extended Triple Diffie-Hellman Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_X3DH_INITIALIZE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X3DH_RESPOND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 6.5.1 Definitions

Mechanisms:
- `CKM_X3DH_INITIALIZE`
- `CKM_X3DH_RESPOND`

#### 6.5.2 Extended Triple Diffie-Hellman key objects

Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation (`CKK_EC_MONTGOMERY`). Three different kinds of keys are used, they differ in their lifespan:

- identity keys are long-term keys, which identify the peer,
- prekeys are short-term keys, which should be rotated often (weekly to hourly)
- onetime prekeys are keys, which should be used only once.

Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of their:

- public Identity key,
- current prekey, signed using XEDDSA with their identity key
- optionally a batch of One-time public keys.

#### 6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and optionally one One-time public key.

When the necessary key material is available, the initiating party calls `CKM_X3DH_INITIALIZE`, also providing the following additional parameters:

- the initiators identity key
the initiators ephemeral key (a fresh, one-time CKK_EC_MONTGOMERY type key)

**CK_X3DH_INITIATE_PARAMS** is a structure that provides the parameters to the
**CKM_X3DH_INITIALIZE** key exchange mechanism. The structure is defined as follows:

```c
typedef struct CK_X3DH_INITIATE_PARAMS {
    CK_X3DH_KDF_TYPE kdf;
    CK_OBJECT_HANDLE pPeer_identity;
    CK_OBJECT_HANDLE pPeer_prekey;
    CK_BYTE_PTR pPrekey_signature;
    CK_BYTE_PTR pOnetime_key;
    CK_OBJECT_HANDLE pOwn_identity;
    CK_OBJECT_HANDLE pOwn_ephemeral;
} CK_X3DH_INITIATE_PARAMS;
```

Table 92, Extended Triple Diffie-Hellman Initiate Message parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>kdf</td>
<td>CK_X3DH_KDF_TYPE</td>
<td>Key derivation function</td>
</tr>
<tr>
<td>pPeer_identity</td>
<td>Key handle</td>
<td>Peers public Identity key (from the prekey-bundle)</td>
</tr>
<tr>
<td>pPeer_prekey</td>
<td>Key Handle</td>
<td>Peers public prekey (from the prekey-bundle)</td>
</tr>
<tr>
<td>pPrekey_signature</td>
<td>Byte array</td>
<td>XEDDSA signature of PEER_PREKEY (from prekey-bundle)</td>
</tr>
<tr>
<td>pOnetime_key</td>
<td>Byte array</td>
<td>Optional one-time public prekey of peer (from the prekey-bundle)</td>
</tr>
<tr>
<td>pOwn_identity</td>
<td>Key Handle</td>
<td>Initiators Identity key</td>
</tr>
<tr>
<td>pOwn_ephemeral</td>
<td>Key Handle</td>
<td>Initiators ephemeral key</td>
</tr>
</tbody>
</table>

### 6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange

Responding an Extended Triple Diffie-Hellman key exchange is done by executing a
**CKM_X3DH_RESPOND** mechanism. **CK_X3DH_RESPOND_PARAMS** is a structure that provides the
parameters to the **CKM_X3DH_RESPOND** key exchange mechanism. All these parameter should be
supplied by the Initiator in a message to the responder. The structure is defined as follows:

```c
typedef struct CK_X3DH_RESPOND_PARAMS {
    CK_X3DH_KDF_TYPE kdf;
    CK_BYTE_PTR pIdentity_id;
    CK_BYTE_PTR pPrekey_id;
    CK_BYTE_PTR pOnetime_id;
    CK_OBJECT_HANDLE pInitiator_identity;
    CK_BYTE_PTR pInitiator_ephemeral;
} CK_X3DH_RESPOND_PARAMS;
```

Table 93, Extended Triple Diffie-Hellman 1st Message parameters:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>kdf</td>
<td>CK_X3DH_KDF_TYPE</td>
<td>Key derivation function</td>
</tr>
<tr>
<td>pIdentity_id</td>
<td>Byte array</td>
<td>Peers public Identity key identifier (from the prekey-bundle)</td>
</tr>
<tr>
<td>pPrekey_id</td>
<td>Byte array</td>
<td>Peers public prekey identifier (from the prekey-bundle)</td>
</tr>
<tr>
<td>pOnetime_id</td>
<td>Byte array</td>
<td>Optional one-time public prekey of peer (from the prekey-bundle)</td>
</tr>
<tr>
<td>pInitiator_identity</td>
<td>Key handle</td>
<td>Initiators Identity key</td>
</tr>
<tr>
<td>pInitiator_ephemeral</td>
<td>Byte array</td>
<td>Initiators ephemeral key</td>
</tr>
</tbody>
</table>

Where the * _id fields are identifiers marking which key has been used from the prekey-bundle, these identifiers could be the keys themselves.

This mechanism has the following rules about key sensitivity and extractability:

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.

### 6.5.5 Extended Triple Diffie-Hellman parameters

- **CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR**

**CK_X3DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X3DH key agreement schemes. It is defined as follows:

```c
typedef CK_ULONG CK_X3DH_KDF_TYPE;
```

The following table lists the defined functions.

**Table 94, X3DH: Key Derivation Functions**

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_BLAKE2B_256_KDF</td>
</tr>
<tr>
<td>CKD_BLAKE2B_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
</tr>
</tbody>
</table>

---

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.
6.6 Double Ratchet

The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations. The algorithm is described in [DoubleRatchet]. The Signal protocol uses X3DH to exchange a shared secret in the first step, which is then used to derive a Double Ratchet secret key.

Table 95, Double Ratchet Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_X2RATCHET_INITIALIZE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_RESPOND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_ENCRYPT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X2RATCHET_DECRYPT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

6.6.1 Definitions

This section defines the key type “CKK_X2RATCHET” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_X2RATCHET_INITIALIZE
- CKM_X2RATCHET_RESPOND
- CKM_X2RATCHET_ENCRYPT
- CKM_X2RATCHET_DECRYPT

6.6.2 Double Ratchet secret key objects

Double Ratchet secret key objects (object class CKO_SECRET_KEY, key type CKK_X2RATCHET) hold Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the mechanism CKM_X2RATCHET_INITIALIZE or CKM_X2RATCHET_RESPOND. In the Signal protocol these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the common attributes defined for this object class:

Table 96, Double Ratchet Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_X2RATCHET_RK</td>
<td>Byte array</td>
<td>Root key</td>
</tr>
<tr>
<td>CKA_X2RATCHET_HKS</td>
<td>Byte array</td>
<td>Sender Header key</td>
</tr>
<tr>
<td>CKA_X2RATCHET_HKR</td>
<td>Byte array</td>
<td>Receiver Header key</td>
</tr>
<tr>
<td>CKA_X2RATCHET_NHKS</td>
<td>Byte array</td>
<td>Next Sender Header Key</td>
</tr>
<tr>
<td>CKA_X2RATCHET_NHKR</td>
<td>Byte array</td>
<td>Next Receiver Header Key</td>
</tr>
<tr>
<td>CKA_X2RATCHET_CKS</td>
<td>Byte array</td>
<td>Sender Chain key</td>
</tr>
</tbody>
</table>
### 6.6.3 Double Ratchet key derivation

The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the receiving, denoted **CKM_X2RATCHET_INITIALIZE** and **CKM_X2RATCHET_RESPOND**, are the key derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by executing a X3DH key exchange.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Additionally the attribute flags indicating which functions the key supports are also contributed by the mechanism.

For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the **ulMinKeySize** and **ulMaxKeySize** fields accordingly.

- **CK_X2RATCHET_INITIALIZE_PARAMS;**
  **CK_X2RATCHET_INITIALIZE_PARAMS_PTR**

**CK_X2RATCHET_INITIALIZE_PARAMS** provides the parameters to the **CKM_X2RATCHET_INITIALIZE** mechanism. It is defined as follows:

```c
typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
    CK_BYTE_PTR sk;
    CK_OBJECT_HANDLE peer_public_prekey;
    CK_OBJECT_HANDLE peer_public_identity;
    CK_OBJECT_HANDLE own_public_identity;
    CK_BBOOL bEncryptedHeader;
    CK_ULONG eCurve;
    CK_MECHANISM_TYPE aeadMechanism;
    CK_X2RATCHET_KDF_TYPE kdfMechanism;
} CK_X2RATCHET_INITIALIZE_PARAMS;
```

The fields of the structure have the following meanings:

- **sk** the shared secret with peer (derived using X3DH)
- **peers_public_prekey** Peers public prekey which the Initiator used in the X3DH
- **peers_public_identity** Peers public identity which the Initiator used in the X3DH
own_public_identity  Initiators public identity as used in the X3DH
bEncryptedHeader  whether the headers are encrypted
eCurve  255 for curve 25519 or 448 for curve 448
aeadMechanism  a mechanism supporting AEAD encryption
kdfMechanism  a Key Derivation Mechanism, such as CKD_BLAKE2B_512_KDF

•  CK_X2RATCHET_RESPOND_PARAMS;
  CK_X2RATCHET_RESPOND_PARAMS_PTR

CK_X2RATCHET_RESPOND_PARAMS provides the parameters to the
CKM_X2RATCHET_RESPOND mechanism. It is defined as follows:

typedef struct CK_X2RATCHET_RESPOND_PARAMS {
  CK_BYTE_PTR  sk;
  CK_OBJECT_HANDLE  own_prekey;
  CK_OBJECT_HANDLE  initiator_identity;
  CK_OBJECT_HANDLE  own_public_identity;
  CK_BBOOL  bEncryptedHeader;
  CK_ULONG  eCurve;
  CK_MECHANISM_TYPE  aeadMechanism;
  CK_X2RATCHET_KDF_TYPE  kdfMechanism;
}  CK_X2RATCHET_RESPOND_PARAMS;

The fields of the structure have the following meanings:

  sk  shared secret with the Initiator

  own_prekey  Own Prekey pair that the Initiator used

  initiator_identity  Initiators public identity key used

  own_public_identity  as used in the prekey bundle by the initiator in the X3DH

  bEncryptedHeader  whether the headers are encrypted

  eCurve  255 for curve 25519 or 448 for curve 448

  aeadMechanism  a mechanism supporting AEAD encryption

  kdfMechanism  a Key Derivation Mechanism, such as CKD_BLAKE2B_512_KDF

6.6.4 Double Ratchet Encryption mechanism

The Double Ratchet encryption mechanism, denoted CKM_X2RATCHET_ENCRYPT and
CKM_X2RATCHET_DECRYPT, are a mechanisms for single part encryption and decryption based on
the Double Ratchet and its underlying AEAD cipher.
6.6.5 Double Ratchet parameters

- `CK_X2RATCHET_KDF_TYPE`, `CK_X2RATCHET_KDF_TYPE_PTR`

`CK_X2RATCHET_KDF_TYPE` is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X key derivation scheme. It is defined as follows:

```c
typedef CK_ULONG CK_X2RATCHET_KDF_TYPE;
```

The following table lists the defined functions.

Table 97, X2RATCHET: Key Derivation Functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
</tr>
<tr>
<td>CKD_BLAKE2B_256_KDF</td>
</tr>
<tr>
<td>CKD_BLAKE2B_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_256_KDF</td>
</tr>
<tr>
<td>CKD_SHA256_KDF</td>
</tr>
<tr>
<td>CKD_SHA3_512_KDF</td>
</tr>
<tr>
<td>CKD_SHA512_KDF</td>
</tr>
</tbody>
</table>

6.7 Wrapping/unwrapping private keys

Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, short Weierstrass EC private keys and DSA private keys. For wrapping, a private key is BER-encoded according to [PKCS #8] PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the private key. The object identifiers for the required algorithm identifiers are as follows:

```c
rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
```

```c
dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }
```

```c
dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x942(10046) number-type(2) 1 }
```

```c
id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
```

```c
id-dsa OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1
}
```

where

```c
pkcs-1 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1
}
```

```c
pkcs-3 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3
}
```
These parameters for the algorithm identifiers have the following types, respectively:

```
NULL

DHParameter ::= SEQUENCE {
    prime             INTEGER,   -- p
    base              INTEGER,   -- g
    privateValueLength INTEGER OPTIONAL
}

DomainParameters ::= SEQUENCE {
    prime             INTEGER,   -- p
    base              INTEGER,   -- g
    subprime          INTEGER,   -- q
    cofactor          INTEGER OPTIONAL,   -- j
    validationParms   ValidationParms OPTIONAL
}

ValidationParms ::= SEQUENCE {
    Seed                 BIT STRING,   -- seed
    PGenCounter          INTEGER     -- parameter verification
}

Parameters ::= CHOICE {
    ecParameters        ECParameters,
    namedCurve          CURVES.&id({CurveNames}),
    implicitlyCA        NULL
}

Dss-Parms ::= SEQUENCE {
    p INTEGER,
    q INTEGER,
    g INTEGER
}
```

For the X9.42 Diffie-Hellman domain parameters, the `cofactor` and the `validationParms` optional fields should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values are not stored within the token.

For the EC domain parameters, the use of `namedCurve` is recommended over the choice `ecParameters`. The choice `implicitlyCA` must not be used in Cryptoki.

Within the PrivateKeyInfo type:

- RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type requires values to be present for all the attributes specific to Cryptoki's RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key's `CKA_MODULUS`, `CKA_PUBLIC_EXPONENT`, `CKA_PRIVATE_EXPONENT`, `CKA_PRIME_1`, `CKA_PRIME_2`, `CKA_EXPONENT_1`, `CKA_EXPONENT_2`, and `CKA_COEFFICIENT` values, it must not create an RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.

- Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.

Short Weierstrass EC private keys are BER-encoded according to SECG SEC 1 ECPublicKey ASN.1 type:

```asn1
ECPrivateKey ::= SEQUENCE {
    Version INTEGER { ecPrivkeyVer1(1) }
    (ecPrivkeyVer1),
    privateKey OCTET STRING,
    parameters [0] Parameters OPTIONAL,
    publicKey [1] BIT STRING OPTIONAL
}
```

Since the EC domain parameters are placed in the PKCS #8's privateKeyAlgorithm field, the optional parameters field in an ECPublicKey must be omitted. A Cryptoki application must be able to unwrap an ECPublicKey that contains the optional publicKey field; however, what is done with this publicKey field is outside the scope of Cryptoki.

DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is encrypted with the secret key. This encryption is defined in the section for the respective key wrapping mechanism.

Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined for the respective key unwrapping mechanism. The data thereby obtained are parsed as a PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the decrypted data does not parse properly, or its type does not match the key type specified in the template for the new key. The unwrapping mechanism contributes only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will take their default values.

Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

```asn1
DSA OBJECT IDENTIFIER ::= { algorithm 12 }
algorithm OBJECT IDENTIFIER ::= {
    iso(1) identifier-organization(3) oiw(14) secsig(3)
    algorithm(2) }
```

with associated parameters

```asn1
DSAParameters ::= SEQUENCE {
    prime1 INTEGER, -- modulus p
    prime2 INTEGER, -- modulus q
    base INTEGER -- base g
}
```

for wrapping DSA private keys. Note that although the two structures for holding DSA domain parameters appear identical when instances of them are encoded, the two corresponding object identifiers are different.

### 6.8 Generic secret key

**Table 98, Generic Secret Key Mechanisms vs. Functions**
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (arbitrary length)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

These key types are used in several of the mechanisms described in this section.

Table 99, Generic Secret Key Object Attributes

- Refer to Table 11 for footnotes

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

```c
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.
6.8.3 Generic secret key generation

The generic secret key generation mechanism, denoted `CKM_GENERIC_SECRET_KEY_GEN`, is used to generate generic secret keys. The generated keys take on any attributes provided in the template passed to the `C_GenerateKey` call, and the `CKA_VALUE_LEN` attribute specifies the length of the key to be generated.

It does not have a parameter.

The template supplied must specify a value for the `CKA_VALUE_LEN` attribute. If the template specifies an object type and a class, they must have the following values:

```
CK_OBJECT_CLASS = CKO_SECRET_KEY;
CK_KEY_TYPE = CKK_GENERIC_SECRET;
```

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bits.

6.9 HMAC mechanisms

Refer to `RFC2104` and `FIPS 198` for HMAC algorithm description. The HMAC secret key shall correspond to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition).

Such keys, for use with HMAC operations can be created using `C_CreateObject` or `C_GenerateKey`.

The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

6.9.1 General block cipher mechanism parameters

- `CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR`

`CK_MAC_GENERAL_PARAMS` provides the parameters to the general-length MACing mechanisms of the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers. It also provides the parameters to the general-length HMACing mechanisms (i.e., SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1). It holds the length of the MAC that these mechanisms produce. It is defined as follows:

```
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

`CK_MAC_GENERAL_PARAMS_PTR` is a pointer to a `CK_MAC_GENERAL_PARAMS`.

6.10 AES

For the Advanced Encryption Standard (AES) see `[FIPS PUB 197]`.

Table 100, AES Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_MAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_OFB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB64</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB8</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB128</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_CFB1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_XCBC_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_AES_XCBC_MAC_96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.10.1 Definitions

This section defines the key type “CKK_AES” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_AES_KEY_GEN
- CKM_AES_ECB
- CKM_AES_CBC
- CKM_AES_MAC
- CKM_AES_MAC_GENERAL
- CKM_AES_CBC_PAD
- CKM_AES_OFB
- CKM_AES_CFB64
- CKM_AES_CFB8
- CKM_AES_CFB128
- CKM_AES_CFB1
- CKM_AES_XCBC_MAC
- CKM_AES_XCBC_MAC_96

### 6.10.2 AES secret key objects

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

**Table 101, AES Secret Key Object Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN&lt;sup&gt;2,3,6&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The following is a sample template for creating an AES secret key object:
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_AES;
CK_UTF8CHAR label[] = “An AES 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_ENCRYPT, &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 ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
the key type of the secret key object.

6.10.3 AES key generation
The AES key generation mechanism, denoted **CKM_AES_KEY_GEN**, is a key generation mechanism for
NIST’s Advanced Encryption Standard.
It does not have a parameter.
The mechanism generates AES 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 AES 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 AES key sizes, in bytes.

6.10.4 AES-ECB
AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and
decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard 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 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 | 

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

### 6.10.5 AES-CBC

AES-CBC, denoted CKM_AES_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>AES</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

### 6.10.6 AES-CBC with PKCS padding

AES-CBC with PKCS padding, denoted CKM_AES_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced
Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].

It has a parameter, a 16-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.

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.

Constraints on key types and the length of data are summarized in the following table:

Table 104, AES-CBC with PKCS Padding: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>multiple of block size</td>
<td>between 1 and block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>AES</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>AES</td>
<td>multiple of block size</td>
<td>between 1 and block length bytes shorter than input length</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

6.10.7 AES-OFB

AES-OFB, denoted CKM_AES_OFB. It is a mechanism for single and multiple-part encryption and decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

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 105, AES-OFB: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

6.10.8 AES-CFB

Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES. AES-OFB mode is described [NIST sp800-38a].

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

*Table 106, AES-CFB: Key And Data Length*

For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

### 6.10.9 General-length AES-MAC

General-length AES-MAC, denoted **CKM_AES_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on NIST Advanced Encryption Standard as defined in FIPS PUB 197 and data authentication as defined in FIPS PUB 113.

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the supported range of AES key sizes, in bytes.

### 6.10.10 AES-MAC

AES-MAC, denoted **CKM_AES_MAC**, is a special case of the general-length AES-MAC mechanism. AES-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the supported range of AES key sizes, in bytes.

### 6.10.11 AES-XCBC-MAC

AES-XCBC-MAC, denoted **CKM_AES_XCBC_MAC**, is a mechanism for single and multiple part signatures and verification; based on NIST’s Advanced Encryption Standard and [RFC 3566].

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the supported range of AES key sizes, in bytes.
For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of AES key sizes, in bytes.

### 6.10.12 AES-XCBC-MAC-96

AES-XCBC-MAC-96, denoted `CKM_AES_XCBC_MAC_96`, is a mechanism for single and multiple part signatures and verification; based on NIST’s Advanced Encryption Standard and [RFC 3566]. It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>AES</td>
<td>Any</td>
<td>16 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>Any</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of AES key sizes, in bytes.

### 6.11 AES with Counter

The following table shows the supported mechanism and function pairs for single and multiple part signatures and verification:

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_CTR</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 6.11.1 Definitions

Mechanisms:

- CKM_AES_CTR

#### 6.11.2 AES with Counter mechanism parameters

- **CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR**

`CK_AES_CTR_PARAMS` is a structure that provides the parameters to the `CKM_AES_CTR` mechanism. It is defined as follows:

```c
typedef struct CK_AES_CTR_PARAMS {
    CK_ULONG ulCounterBits;
    CK_BYTE cb[16];
} CK_AES_CTR_PARAMS;
```

`ulCounterBits` specifies the number of bits in the counter block (cb) that shall be incremented. This number shall be such that $0 < \text{ulCounterBits} \leq 128$. For any values outside this range the mechanism shall return `CKR_MECHANISM_PARAM_INVALID`.

---

11 August 2022  
Standards Track Work Product  
Copyright © OASIS Open 2022. All Rights Reserved.  
Page 251 of 426
It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

E.g. as defined in [RFC 3686]:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### 6.11.3 AES with Counter Encryption / Decryption

Generic AES counter mode is described in NIST Special Publication 800-38A and in RFC 3686. These describe encryption using a counter block which may include a nonce to guarantee uniqueness of the counter block. Since the nonce is not incremented, the mechanism parameter must specify the number of counter bits in the counter block.

The block counter is incremented by 1 after each block of plaintext is processed. There is no support for any other increment functions in this mechanism.

If an attempt to encrypt/decrypt is made which will cause an overflow of the counter block's counter bits, then the mechanism shall return CKR_DATA_LEN_RANGE. Note that the mechanism should allow the final post increment of the counter to overflow (if it implements it this way) but not allow any further processing after this point. E.g. if ulCounterBits = 2 and the counter bits start as 1 then only 3 blocks of data can be processed.

### 6.12 AES CBC with Cipher Text Stealing CTS

Ref [NIST AES CTS]

This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to the same length of cipher text.

#### Table 112, AES CBC with Cipher Text Stealing CTS Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_CTS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.12.1 Definitions

Mechanisms:

CKM_AES_CTS

---

Standards Track Work Product Copyright © OASIS Open 2022. All Rights Reserved.
6.12.2 AES CTS mechanism parameters

It has a parameter, a 16-byte initialization vector.

Table 113, AES-CTS: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>AES</td>
<td>Any, ≥ block size (16 bytes)</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>AES</td>
<td>any, ≥ block size (16 bytes)</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

6.13 Additional AES Mechanisms

Table 114, Additional AES Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_GCM</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CCM</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_GMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.13.1 Definitions

Mechanisms:
- CKM_AES_GCM
- CKM_AES_CCM
- CKM_AES_GMAC

Generator Functions:
- CKG_NO_GENERATE
- CKG_GENERATE
- CKG_GENERATE_COUNTER
- CKG_GENERATE_RANDOM
- CKG_GENERATE_COUNTER_XOR

6.13.2 AES-GCM Authenticated Encryption / Decryption

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.

Encrypt:
- Set the IV length $ulIvLen$ in the parameter block.
- Set the IV data $plv$ 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 tag length ulTagBits in the parameter block.
- Call C_EncryptInit() for CKM_AES_GCM mechanism with parameters and key K.
- Call C_Encrypt(), or C_EncryptUpdate()*4 C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:
- Set the IV length ulIvLen in the parameter block.
- Set the IV data plv 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 tag length ulTagBits in the parameter block.
- Call C_DecryptInit() for CKM_AES_GCM mechanism with parameters and key K.
- Call C_Decrypt(), or C_DecryptUpdate()*. C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since CKM_AES_GCM is an AEAD cipher, no data should be returned until C_Decrypt() or C_DecryptFinal().

MessageEncrypt:
- Set the IV length ulIvLen in the parameter block.
- Set plv to hold the IV data returned from C_EncryptMessage() and C_EncryptMessageBegin(). If ulIvFixedBits is not zero, then the most significant bits of plv contain the fixed IV. If ivGenerator is set to CKG_NO_GENERATE, plv is an input parameter with the full IV.
- Set the ulIvFixedBits and ivGenerator fields in the parameter block.
- Set the tag length ulTagBits in the parameter block.
- Set pTag to hold the tag data returned from C_EncryptMessage() or the final C_EncryptMessageNext().
- Call C_MessageEncryptInit() for CKM_AES_GCM mechanism key K.
- Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()*. The mechanism parameter is passed to all three of these functions.
- Call C_MessageEncryptFinal() to close the message decryption.

MessageDecrypt:
- Set the IV length ulIvLen in the parameter block.
- Set the IV data plv in the parameter block.
- The ulIvFixedBits and ivGenerator fields are ignored.
- Set the tag length ulTagBits in the parameter block.
- Set the tag data pTag in the parameter block before C_DecryptMessage() or the final 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()\(^6\).

  The mechanism parameter is passed to all three of these functions.

• Call C_MessageDecryptFinal() to close the message decryption.

In \( \text{pIV} \) the least significant bit of the initialization vector is the rightmost bit. \( \text{ulIVLen} \) is the length of the initialization vector in bytes.

On MessageEncrypt, the meaning of \( \text{ivGenerator} \) 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_RAND 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 \( \text{pIV} \).

Modules must implement CKG_GENERATE. Modules may also reject \( \text{ulIVFixedBits} \) values which are too large. Zero is always an acceptable value for \( \text{ulIVFixedBits} \).

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 \( \text{ulTagBits} \) 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 using 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 \( \text{ulDataLen} \) in the parameter block.
- Set the nonce length \( \text{ulNonceLen} \) and the nonce data \( pNonce \) in the parameter block.
- Set the AAD data \( pAAD \) and size \( \text{ulAADLen} \) in the parameter block. \( pAAD \) may be NULL if \( \text{ulAADLen} \) is 0.
- Set the MAC length \( \text{ulMACLen} \) 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 \( \text{ulDataLen} \). The output length will be \( \text{ulDataLen} + \text{ulMACLen} \).

**Decrypt:**

\(^6\) \( **\) 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().

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()\(^*7\). 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.

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()\(^*8\). The mechanism parameter is passed to all three functions.

---

7 “*” indicates 0 or more calls may be made as required
8 “*” indicates 0 or more calls may be made as required
• Call C_MessageDecryptFinal() to close the message decryption.

In pNonce the least significant bit of the nonce is the rightmost bit. ulNonceLen is the length of the nonce in bytes.

On MessageEncrypt, the meaning of nonceGenerator is as follows: CKG_NO_GENERATE means the nonce is passed in on MessageEncrypt and no internal MAC generation is done. CKG_GENERATE means that the non-fixed portion of the nonce is generated by the module internally. The generation method is not defined.

CKG_GENERATE_COUNTER means that the non-fixed portion of the nonce 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 nonce is generated by the module internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in pNonce.

Modules must implement CKG_GENERATE. Modules may also reject ulNonceFixedBits values which are too large. Zero is always an acceptable value for ulNonceFixedBits.

In Encrypt and Decrypt the MAC is appended to the cipher text and the least significant byte of the MAC is the rightmost byte and the MAC bytes are the rightmost ulMACLen bytes. In MessageEncrypt the MAC is returned in the pMAC field of CK_CCM_MESSAGE_PARAMS. In MessageDecrypt the MAC is provided by the pMAC field of CK_CCM_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.

### 6.13.4 AES-GMAC

AES-GMAC, denoted CKM_AES_GMAC, is a mechanism for single and multiple-part signatures and verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism parameters. When GMAC is used with C_Sign or C_Verify, pData points to the AAD. GMAC does not use plaintext or ciphertext.

The signature produced by GMAC, also referred to as a Tag, the tag’s length is determined by the CK_GCM_PARAMS field ulTagBits.

The IV length is determined by the CK_GCM_PARAMS field ulIVLen.

Constraints on key types and the length of data are summarized in the following table:

**Table 115, AES-GMAC: Key And Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>&lt; 2^64</td>
<td>Depends on param’s ulTagBits</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>&lt; 2^64</td>
<td>Depends on param’s ulTagBits</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of AES key sizes, in bytes.

### 6.13.5 AES GCM and CCM Mechanism parameters

♦ **CK_GENERATOR_FUNCTION**

Functions to generate unique IVs and nonces.
typedef CK_ULONG CK_GENERATOR_FUNCTION;

♦ CK_GCM_PARAMS; CK_GCM_PARAMS_PTR

CK_GCM_PARAMS is a structure that provides the parameters to the CKM_AES_GCM mechanism when used for Encrypt or Decrypt. It is defined as follows:

typedef struct CK_GCM_PARAMS {
    CK_BYTE_PTR    pIv;
    CK_ULONG       ulIvLen;
    CK_ULONG       ulIvBits;
    CK_BYTE_PTR    pAAD;
    CK_ULONG       ulAADLen;
    CK_ULONG       ulTagBits;
} CK_GCM_PARAMS;

The fields of the structure have the following meanings:

- **pIv** pointer to initialization vector
- **ulIvLen** 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.
- **ulIvBits** length of initialization vector in bits. Do not use ulIvBits to specify the length of the initialization vector, but ulIvLen instead.
- **pAAD** pointer to additional authentication data. This data is authenticated but not encrypted.
- **ulAADLen** length of pAAD in bytes. The length of the AAD can be any number between 0 and \((2^{32}) - 1\).
- **ulTagBits** length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128.

CK_GCM_PARAMS_PTR is a pointer to a CK_GCM_PARAMS.

♦ CK_GCM_MESSAGE_PARAMS; CK_GCM_MESSAGE_PARAMS_PTR

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:

typedef struct CK_GCM_MESSAGE_PARAMS {
    CK_BYTE_PTR    pIv;
    CK_ULONG       ulIvLen;
    CK_ULONG       ulIvFixedBits;
    CK_GENERATOR_FUNCTION ivGenerator;
    CK_BYTE_PTR    pTag;
    CK_ULONG       ulTagBits;
} CK_GCM_MESSAGE_PARAMS;

The fields of the structure have the following meanings:

- **pIv** pointer to initialization vector
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.

ullvFixedBits number of bits of the original IV to preserve when generating an new IV. These bits are counted from the Most significant bits (to the right).

ivGenerator Function used to generate a new IV. Each IV must be unique for a given session.

pTag location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.

ulTagBits length of authentication tag in bits. Can be any value between 0 and 128.

**CK_GCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_GCM_MESSAGE_PARAMS**.

**CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

**CK_CCM_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism when used for Encrypt or Decrypt. It is defined as follows:

```c
typedef struct CK_CCM_PARAMS {
    CK_ULONG ulDataLen; /*plaintext or ciphertext*/
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceLen;
    CK_BYTE_PTR pAAD;
    CK_ULONG ulAADLen;
    CK_ULONG ulMACLen;
} CK_CCM_PARAMS;
```

The fields of the structure have the following meanings, where L is the size in bytes of the data length’s length (2 ≤ L ≤ 8):

- ulDataLen length of the data where 0 ≤ ulDataLen < \(2^L\).
- pNonce the nonce.
- pAAD Additional authentication data. This data is authenticated but not encrypted.
- ulAADLen length of pAAD in bytes where 0 ≤ ulAADLen ≤ \((2^{32}) - 1\).
- ulMACLen length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

**CK_CCM_PARAMS_PTR** is a pointer to a **CK_CCM_PARAMS**.

**CK_CCM_MESSAGE_PARAMS; CK_CCM_MESSAGE_PARAMS_PTR**

**CK_CCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```c
typedef struct CK_CCM_MESSAGE_PARAMS {
    CK_ULONG ulDataLen; /*plaintext or ciphertext*/
    CK_BYTE_PTR pNonce;
    CK_ULONG ulDataLen;
    CK_BYTE_PTR pAAD;
    CK_ULONG ulAADLen;
    CK_ULONG ulMACLen;
} CK_CCM_MESSAGE_PARAMS;
```
CK_ULONG ulNonceLen;
CK_ULONG ulNonceFixedBits;
CK_GENERATOR_FUNCTION nonceGenerator;
CK_BYTE_PTR pMAC;
CK_ULONG ulMACLen;
} CK_CCM_MESSAGE_PARAMS;

The fields of the structure have the following meanings, where L is the size in bytes of the data length’s length (2 <= L <= 8):

ulDataLen   length of the data where 0 <= ulDataLen < 2^(8L).
pNonce      the nonce.
ulNonceFixedBits number of bits of the original nonce to preserve when generating a new nonce. These bits are counted from the Most significant bits (to the right).
nonceGenerator Function used to generate a new nonce. Each nonce must be unique for a given session.
pMAC        location of the CCM MAC returned on MessageEncrypt, provided on MessageDecrypt.
ulMACLen    length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

CK_CCM_MESSAGE_PARAMS_PTR is a pointer to a CK_CCM_MESSAGE_PARAMS.

6.14 AES CMAC

Table 116, Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR(^1)</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_CMAC_GENERAL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CMAC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

\(^1\) 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

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-MAC 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_AES</td>
<td>any</td>
<td>Block size (16 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_AES</td>
<td>any</td>
<td>Block size (16 bytes)</td>
</tr>
</tbody>
</table>

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
6.15.1 Definitions

This section defines the key type “CKK_AES_XTS” for type CK_KEY_TYPE as used in the
CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_AES_XTS
- CKM_AES_XTS_KEY_GEN

6.15.2 AES-XTS secret key objects

6.15.3 AES-XTS key generation

The double-length AES-XTS key generation mechanism, denoted CKM_AES_XTS_KEY_GEN, is a key
generation mechanism for double-length AES-XTS keys.

The mechanism generates AES-XTS keys with a particular length in bytes as specified in the
CKA_VALUE_LEN attributes of the template for the key.

This mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new
key. Other attributes supported by the double-length AES-XTS 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 AES-XTS key sizes, in bytes.

6.15.4 AES-XTS

AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted CKM_AES_XTS,
is a mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and
64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraintson
key types and the length of data are summarized in the following table:

Table 121, AES-XTS: Key And Data Length
### 6.16 AES Key Wrap

#### Table 122, AES Key Wrap Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_AES_KEY_WRAP</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_KEY_WRAP_PAD</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_KEY_WRAP_KWP</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_AES_KEY_WRAP_PKCS7</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<sup>1</sup>SR = SignRecover, VR = VerifyRecover

#### 6.16.1 Definitions

- **Mechanisms:**
  - CKM_AES_KEY_WRAP
  - CKM_AES_KEY_WRAP_PAD
  - CKM_AES_KEY_WRAP_KWP
  - CKM_AES_KEY_WRAP_PKCS7

#### 6.16.2 AES Key Wrap Mechanism parameters

The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if present, must be a fixed size array of 8 bytes for CKM_AES_KEY_WRAP and CKM_AES_KEY_WRAP_PKCS7, resp. 4 bytes for CKM_AES_KEY_WRAP_KWP; and, if NULL, will use the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

The type of this parameter is CK_BYTE_PTR and the pointer points to the array of bytes to be used as the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM_AES_KEY_WRAP and CKM_AES_KEY_WRAP_PKCS7, resp. 4 for CKM_AES_KEY_WRAP_KWP, and the pointer non-NULL.

#### 6.16.3 AES Key Wrap

The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and single-part encryption and decryption.

**CKM_AES_KEY_WRAP**

The CKM_AES_KEY_WRAP mechanism can wrap a key of any length. A secret key whose length is not a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is defined in Section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the encoded private key will be zero padded to fit if necessary.
The CKM_AES_KEY_WRAP mechanism can only encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm semiblock size.

For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it 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 length specified in the template must not be less than \( n-7 \) bytes, where \( n \) is the length of the wrapped key. In case of a private key, the mechanism parses the encoding as defined in section 6.7 and ignores trailing zero bytes.

**CKM_AES_KEY_WRAP_PAD**

The CKM_AES_KEY_WRAP_PAD mechanism is deprecated. CKM_AES_KEY_WRAP_KWP resp. CKM_AES_KEY_WRAP_PKCS7 shall be used instead.

**CKM_AES_KEY_WRAP_KWP**

The CKM_AES_KEY_WRAP_KWP mechanism can wrap a key or encrypt block of data of any length. The input is zero-padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which produces same results as RFC 5649.

**CKM_AES_KEY_WRAP_PKCS7**

The CKM_AES_KEY_WRAP_PKCS7 mechanism can wrap a key or encrypt a block of data of any length. It does the padding detailed in [PKCS #7] of inputs (keys or data blocks) up to a semiblock size to make it an exact multiple of AES Key Wrap algorithm semiblock size (8bytes), always producing wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of data as defined in Section 6.2 of [AES KEYWRAP].

### 6.17 Key derivation by data encryption – DES & AES

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.

Table 123, Key derivation by data encryption Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### 6.17.1 Definitions

Mechanisms:
typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE iv[8];
    CK_BYTE_PTR pData;
    CK_ULONG length;
} CK_DES_CBC_ENCRYPT_DATA_PARAMS;

typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;

typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE iv[16];
    CK_BYTE_PTR pData;
    CK_ULONG length;
} CK_AES_CBC_ENCRYPT_DATA_PARAMS;

typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR 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>
<thead>
<tr>
<th>Mechanism</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long.</td>
</tr>
<tr>
<td>CKM_DES3_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td>CKM_AES_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td>CKM_DES_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
<tr>
<td>CKM_DES3_CBC_ENCRYPT_DATA</td>
<td>Uses CK_AES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.</td>
</tr>
<tr>
<td>CKM_AES_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES2_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_MAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_MAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.18.1 Definitions

This section defines the key type “CKK_DES2” and “CKK_DES3” for type CK_KEY_TYPE as used in the
CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_DES2_KEY_GEN
CKM_DES3_KEY_GEN
CKM_DES3_ECB
CKM_DES3_CBC
CKM_DES3_MAC
CKM_DES3_MAC_GENERAL
CKM_DES3_CBC_PAD

6.18.2 DES2 secret key objects

DES2 secret key objects (object class CKO_SECRET_KEY, key type CKK_DES2) hold double-length
DES keys. The following table defines the DES2 secret key object attributes, in addition to the common
attributes defined for this object class:

Table 126, DES2 Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE¹,²,⁵,⁷</td>
<td>Byte array</td>
<td>Key value (always 16 bytes long)</td>
</tr>
</tbody>
</table>

¹ Refer to Table 11 for footnotes

DES2 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 DES2 key must have its parity bits properly set). Attempting to create or
unwrap a DES2 key with incorrect parity will return an error.

The following is a sample template for creating a double-length DES secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
```
CK_KEY_TYPE keyType = CKK_DES2;
CK_UTF8CHAR label[] = "A DES2 secret key object";
CK_BYTE value[16] = {...};
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_ENCRYPT, &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 ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

6.18.3 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>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (always 24 bytes long)</td>
</tr>
</tbody>
</table>

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:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES3;
CK_UTF8CHAR label[] = "A DES3 secret key object";
CK_BYTE value[24] = {...};
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_ENCRYPT, &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 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 generated 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>
<thead>
<tr>
<th>Function</th>
<th>Mechansim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt &amp; Decrypt</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Encrypt &amp; Decrypt</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
<tr>
<td>Sign &amp; Verify</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Sign &amp; Verify</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
<tr>
<td>Digest</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Digest</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
<tr>
<td>Gen. Key/Key Pair</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Gen. Key/Key Pair</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
<tr>
<td>Wrap &amp; Unwrap</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Wrap &amp; Unwrap</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
<tr>
<td>Derive</td>
<td>CKM_DES_OFB64 ✓</td>
</tr>
<tr>
<td>Derive</td>
<td>CKM_DES_OFB8 ✓</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

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**

Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**. 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 130, CFB: Key And Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_DES, CKK_DES2, CKK_DES3</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

**6.19 Double and Triple-length DES CMAC**

**Table 131, Double and Triple-length DES CMAC Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DES3_CMAC_GENERAL</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_DES3_CMAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 SR = SignRecover, VR = VerifyRecover.

**6.19.1 Definitions**

**Mechanisms:**

- CKM_DES3_CMAC_GENERAL
- CKM_DES3_CMAC
6.19.2 Mechanism parameters

CKM_DES3_CMAC_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure. 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_DES3</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_DES3</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_DES3</td>
<td>any</td>
<td>Block size (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_DES3</td>
<td>any</td>
<td>Block size (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>CKK_DES2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
6.20.1 Definitions

This section defines the key type “CKK_SHA_1_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA_1
- CKM_SHA_1_HMAC
- CKM_SHA_1_HMAC_GENERAL
- CKM_SHA1_KEY_DERIVATION
- CKM_SHA_1_KEY_GEN

6.20.2 SHA-1 digest

The SHA-1 mechanism, denoted CKM_SHA_1, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-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 135, SHA-1: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>20</td>
</tr>
</tbody>
</table>

6.20.3 General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted CKM_SHA_1_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys and CKK_SHA_1_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 1-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 136, General-length SHA-1-HMAC: Key And Data Length
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA_1_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA_1_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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_SHA1_KEY_GEN`, is a key generation mechanism for NIST's SHA-1-HMAC.

It does not have a parameter.
The mechanism generates SHA-1-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 SHA-1-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_SHA_1_HMAC key sizes, in bytes.

### 6.21 SHA-224

#### Table 137, SHA-224 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA224</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA224_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA224_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 6.21.1 Definitions

This section defines the key type "CKK_SHA224_HMAC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA224
- CKM_SHA224_HMAC
- CKM_SHA224_HMAC_GENERAL
- CKM_SHA224_KEY_DERIVATION
- CKM_SHA224_KEY_GEN

#### 6.21.2 SHA-224 digest

The SHA-224 mechanism, denoted CKM_SHA224, is a mechanism for message digesting, following the Secure Hash Algorithm with a 224-bit message digest defined in FIPS PUB 180-4.

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 138, SHA-224: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

#### 6.21.3 General-length SHA-224-HMAC

The general-length SHA-224-HMAC mechanism, denoted CKM_SHA224_HMAC_GENERAL, is the same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based
on the SHA-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_SHA224_HMAC. FIPS-198 compliant tokens may require the key length to
be at least 14 bytes; that is, half the size of the SHA-224 hash output.
It has a parameter, a **CK_MAC_GENERIC_PARAMS**, which holds the length in bytes of the desired
output. This length should be in the range 1-28 (the output size of SHA-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 will be taken from the start of the full 28-byte HMAC
output.

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret CKK_SHA224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret CKK_SHA224_HMAC</td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
</tbody>
</table>

### 6.21.4 SHA-224-HMAC

The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length
SHA-224-HMAC mechanism.
It has no parameter, and always produces an output of length 28.

### 6.21.5 SHA-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key
derivation mechanism in Section 6.20.5 except that it uses the SHA-224 hash function and the relevant
length is 28 bytes.

### 6.21.6 SHA-224 HMAC key generation

The SHA-224-HMAC key generation mechanism, denoted **CKM_SHA224_KEY_GEN**, is a key
generation mechanism for NIST's SHA224-HMAC.
It does not have a parameter.
The mechanism generates SHA224-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 SHA224-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_SHA224_HMAC** key sizes, in bytes.

### 6.22 SHA-256

Table 140, SHA-256 Mechanisms vs. Functions
### 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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

### 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**
### 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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td><code>CKM_SHA384</code></td>
<td></td>
</tr>
<tr>
<td><code>CKM_SHA384_HMAC_GENERAL</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>CKM_SHA384_HMAC</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>CKM_SHA384_KEY_DERIVATION</code></td>
<td></td>
</tr>
<tr>
<td><code>CKM_SHA384_KEY_GEN</code></td>
<td></td>
</tr>
</tbody>
</table>

### 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.
CKM_SHA384
CKM_SHA384_HMAC
CKM_SHA384_HMAC_GENERAL
CKM_SHA384_KEY_DERIVATION
CKM_SHA384_KEY_GEN

6.23.2 SHA-384 digest

The SHA-384 mechanism, denoted CKM_SHA384, is a mechanism for message digesting, following the Secure Hash Algorithm with a 384-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 144, SHA-384: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

6.23.3 General-length SHA-384-HMAC

The general-length SHA-384-HMAC mechanism, denoted CKM_SHA384_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-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_SHA384_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA-384 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 0-48 (the output size of SHA-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 will be taken from the start of the full 48-byte HMAC output.

Table 145, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA384_HMAC</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
</tbody>
</table>

6.23.4 SHA-384-HMAC

The SHA-384-HMAC mechanism, denoted CKM_SHA384_HMAC, is a special case of the general-length SHA-384-HMAC mechanism. It has no parameter, and always produces an output of length 48.

6.23.5 SHA-384 key derivation

SHA-384 key derivation, denoted CKM_SHA384_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.
6.23.6 SHA-384 HMAC key generation

The SHA-384-HMAC key generation mechanism, denoted \texttt{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 \texttt{CKA_VALUE_LEN} attribute of the template for the key.

The mechanism contributes the \texttt{CKA_CLASS}, \texttt{CKA_KEY_TYPE}, and \texttt{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 \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of \texttt{CKM_SHA384_HMAC} key sizes, in bytes.

### 6.24 SHA-512

#### Table 146, SHA-512 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR$^1$</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.24.1 Definitions

This section defines the key type “CKK_SHA512_HMAC” for type CK_KEY_TYPE as used in the \texttt{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 \texttt{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
<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

### 6.24.3 General-length SHA-512-HMAC

The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_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-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_SHA512_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA-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 0-64 (the output size of SHA-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 will be taken from the start of the full 64-byte HMAC output.

### Table 148, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, <strong>CKK_SHA512_HMAC</strong></td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, <strong>CKK_SHA512_HMAC</strong></td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
</tbody>
</table>

### 6.24.4 SHA-512-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length SHA-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### 6.24.5 SHA-512 key derivation

SHA-512 key derivation, denoted **CKM_SHA512_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.

### 6.24.6 SHA-512 HMAC key generation

The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key generation mechanism for NIST’s SHA512-HMAC.

It does not have a parameter.

The mechanism generates SHA512-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 SHA512-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_SHA512_HMAC** key sizes, in bytes.
### 6.25 SHA-512/224

**Table 149, SHA-512/224 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt</td>
</tr>
<tr>
<td></td>
<td>Decrypt</td>
</tr>
<tr>
<td></td>
<td>Sign</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>Verif</td>
</tr>
<tr>
<td></td>
<td>SR</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>VR</td>
</tr>
<tr>
<td></td>
<td>Digest</td>
</tr>
<tr>
<td></td>
<td>Gen.</td>
</tr>
<tr>
<td></td>
<td>Key/</td>
</tr>
<tr>
<td></td>
<td>Key</td>
</tr>
<tr>
<td></td>
<td>Pair</td>
</tr>
<tr>
<td></td>
<td>Wrap</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>Unwrap</td>
</tr>
<tr>
<td></td>
<td>Derive</td>
</tr>
<tr>
<td>CKM_SHA512_224</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_HMAC</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_224_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_224_KEY_GEN</td>
<td></td>
</tr>
</tbody>
</table>

#### 6.25.1 Definitions

This section defines the key type “CKK_SHA512_224_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

**Mechanisms:**

- CKM_SHA512_224
- CKM_SHA512_224_HMAC
- CKM_SHA512_224_HMAC_GENERAL
- CKM_SHA512_224_KEY_DERIVATION
- CKM_SHA512_224_KEY_GEN

#### 6.25.2 SHA-512/224 digest

The SHA-512/224 mechanism, denoted **CKM_SHA512_224**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 224 bits. **CKM_SHA512_224** is the same as **CKM_SHA512_T** with a parameter value of 224.

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 150, SHA-512/224: Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

#### 6.25.3 General-length SHA-512/224-HMAC

The general-length SHA-512/224-HMAC mechanism, denoted **CKM_SHA512_224_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-512/224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK_SHA512_224_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-512/224 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 0-28 (the output size of SHA-512/224 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 will be taken from the start of the full 28-byte HMAC output.

**Table 151, General-length SHA-384-HMAC: Key And Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, <code>CKK_SHA512_224_HMAC</code></td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, <code>CKK_SHA512_224_HMAC</code></td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
</tbody>
</table>

### 6.25.4 SHA-512/224-HMAC

The SHA-512-HMAC mechanism, denoted `CKM_SHA512_224_HMAC`, is a special case of the general-length SHA-512/224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

### 6.25.5 SHA-512/224 key derivation

The SHA-512/224 key derivation, denoted `CKM_SHA512_224_KEY_DERIVATION`, is the same as the SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/224 hash function and the relevant length is 28 bytes.

### 6.25.6 SHA-512/224 HMAC key generation

The SHA-512/224-HMAC key generation mechanism, denoted `CKM_SHA512_224_KEY_GEN`, is a key generation mechanism for NIST’s SHA512/224-HMAC.

It does not have a parameter. The mechanism generates SHA512/224-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 SHA512/224-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_SHA512_224_HMAC` key sizes, in bytes.

### 6.26 SHA-512/256

**Table 152, SHA-512/256 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512_256</td>
<td>Encrypt, Decrypt, Sign &amp; Verify, SR &amp; VR, Digest, Gen. Key/Key Pair, Wrap &amp; Unwrap, Derive</td>
</tr>
</tbody>
</table>
### 6.26.1 Definitions

This section defines the key type “CKK_SHA512_256_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

#### Mechanisms:
- CKM_SHA512_256
- CKM_SHA512_256_HMAC
- CKM_SHA512_256_KEY_DERIVATION
- CKM_SHA512_256_KEY_GEN

### 6.26.2 SHA-512/256 digest

The SHA-512/256 mechanism, denoted **CKM_SHA512_256**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 256 bits. **CKM_SHA512_256** is the same as **CKM_SHA512** with a parameter value of 256.

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 153, SHA-512/256: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

### 6.26.3 General-length SHA-512/256-HMAC

The general-length SHA-512/256-HMAC mechanism, denoted **CKM_SHA512_256_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-512/256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_SHA512_256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-512/256 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-512/256 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 154, General-length SHA-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret, CKK_SHA512_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret, CKK_SHA512_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
</tbody>
</table>

6.26.4 SHA-512/256-HMAC

The SHA-512-HMAC mechanism, denoted CKM_SHA512_256_HMAC, is a special case of the general-length SHA-512/256-HMAC mechanism.

It has no parameter, and always produces an output of length 32.

6.26.5 SHA-512/256 key derivation

The SHA-512/256 key derivation, denoted CKM_SHA512_256_KEY_DERIVATION, is the same as the SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/256 hash function and the relevant length is 32 bytes.

6.26.6 SHA-512/256 HMAC key generation

The SHA-512/256-HMAC key generation mechanism, denoted CKM_SHA512_256_KEY_GEN, is a key generation mechanism for NIST’s SHA512/256-HMAC.

It does not have a parameter.

The mechanism generates SHA512/256-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 SHA512/256-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_SHA512_256_HMAC key sizes, in bytes.

6.27 SHA-512/t

Table 155, SHA-512/t Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrpyt</td>
</tr>
<tr>
<td>CKM_SHA512_T</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_HMAC_GENERAL</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_HMAC</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_T_KEY_DERIVATION</td>
<td></td>
</tr>
</tbody>
</table>
This section defines the key type "CKK_SHA512_T_HMAC" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:
- CKM_SHA512_T
- CKM_SHA512_T_HMAC
- CKM_SHA512_T_HMAC_GENERAL
- CKM_SHA512_T_KEY_DERIVATION
- CKM_SHA512_T_KEY_GEN

### 6.27.2 SHA-512/t digest

The SHA-512/t mechanism, denoted **CKM_SHA512_T**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to t bits.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of $0 < t < 512$, and $t \neq 384$.

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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>⌈t/8⌉, where 0 &lt; t &lt; 512, and t \neq 384</td>
</tr>
</tbody>
</table>

### 6.27.3 General-length SHA-512/t-HMAC

The general-length SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_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-512/t hash function and length of the output should be in the range $0 \leq t < 512$, and $t \neq 384$.

### 6.27.4 SHA-512/t-HMAC

The SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC**, is a special case of the general-length SHA-512-t-HMAC mechanism.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of $0 \leq t < 512$, and $t \neq 384$. 

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA512_T_KEY_GEN</td>
<td>Encryp Decryp Sign &amp; Verif SR &amp; VR 1 Diges t Gen - Key/ Key Pair Wrap &amp; Unwrap Derive</td>
</tr>
</tbody>
</table>
6.27.5 SHA-512/t key derivation

The SHA-512/t key derivation, denoted `CKM_SHA512_T_KEY_DERIVATION`, is the same as the SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/t hash function and the relevant length is $\lceil t/8 \rceil$ bytes, where $0 < t < 512$, and $t <> 384$.

6.27.6 SHA-512/t HMAC key generation

The SHA-512/t-HMAC key generation mechanism, denoted `CKM_SHA512_T_KEY_GEN`, is a key generation mechanism for NIST's SHA512/t-HMAC.

It does not have a parameter.

The mechanism generates SHA512/t-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 SHA512/t-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_SHA512_T_HMAC` key sizes, in bytes.

6.28 SHA3-224

Table 157, SHA3-224 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_224_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.28.1 Definitions

Mechanisms:

- CKM_SHA3_224
- CKM_SHA3_224_HMAC
- CKM_SHA3_224_HMAC_GENERAL
- CKM_SHA3_224_KEY_DERIVATION
- CKM_SHA3_224_KEY_GEN
- 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>28</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or <code>CKK_SHA3_224_HMAC</code></td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or <code>CKK_SHA3_224_HMAC</code></td>
<td>Any</td>
<td>1-28, depending on parameters</td>
</tr>
</tbody>
</table>

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...
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_224_HMAC key sizes, in bytes.

### 6.29 SHA3-256

**Table 160, SHA3-256 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_SHA3_256</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_256_KEY_DERIVATION</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_256_KEY_GEN</td>
<td></td>
</tr>
</tbody>
</table>

### 6.29.1 Definitions

Mechanisms:

- CKM_SHA3_256
- CKM_SHA3_256_HMAC
- CKM_SHA3_256_HMAC_GENERAL
- CKM_SHA3_256_KEY_DERIVATION
- CKM_SHA3_256_KEY_GEN
- CKK_SHA3_256_HMAC

### 6.29.2 SHA3-256 digest

The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following
the Secure Hash 3 Algorithm with a 256-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 161, SHA3-256: Data Length**

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

### 6.29.3 General-length SHA3-256-HMAC

The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_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 SHA3-256 hash function and length of the output should be in the range 1-32.
The keys it uses are generic secret keys and CKK_SHA3_256_HMAC. FIPS-198 compliant tokens may
require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.
It has a parameter, a \texttt{CK_MACGENERAL\_PARAMS}, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA3-256 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 shall be taken from the start of the full 32-byte HMAC output.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Function} & \textbf{Key type} & \textbf{Data length} & \textbf{Signature length} \\
\hline
C\_Sign & generic secret or \texttt{CK\_SHA3\_256\_HMAC} & Any & 1-32, depending on parameters \\
\hline
C\_Verify & generic secret or \texttt{CK\_SHA3\_256\_HMAC} & Any & 1-32, depending on parameters \\
\hline
\end{tabular}
\end{table}

6.29.4 SHA3-256-HMAC

The SHA-256-HMAC mechanism, denoted \texttt{CKM\_SHA3\_256\_HMAC}, is a special case of the general-length SHA-256-HMAC mechanism.

It has no parameter, and always produces an output of length 32.

6.29.5 SHA3-256 key derivation

SHA-256 key derivation, denoted \texttt{CKM\_SHA3\_256\_KEY\_DERIVATION}, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA3-256 hash function and the relevant length is 32 bytes.

6.29.6 SHA3-256 HMAC key generation

The SHA3-256-HMAC key generation mechanism, denoted \texttt{CKM\_SHA3\_256\_KEY\_GEN}, is a key generation mechanism for NIST’s SHA3-256-HMAC.

It does not have a parameter.

The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the \texttt{CKA\_VALUE\_LEN} attribute of the template for the key.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key. Other attributes supported by the SHA3-256-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 \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of \texttt{CKM\_SHA3\_256\_HMAC} key sizes, in bytes.

6.30 SHA3-384

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\textbf{Mechanism} & \textbf{Encrypt & Decrypt} & \textbf{Sign & Verify} & \textbf{SR & VR} & \textbf{Digest} & \textbf{Gen. Key/Key Pair} & \textbf{Wrap & Unwrap} & \textbf{Derive} \\
\hline
\texttt{CKM\_SHA3\_384} & & & & \checkmark & & & \\
\texttt{CKM\_SHA3\_384\_HMAC\_GENERAL} & \checkmark & & & & & & \\
\texttt{CKM\_SHA3\_384\_HMAC} & \checkmark & & & & & & \\
\texttt{CKM\_SHA3\_384\_KEY\_DERIVATION} & & & & & & & \checkmark \\
\hline
\end{tabular}
\end{table}
Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_384_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

10260 **6.30.1 Definitions**

10261 CKM_SHA3_384

10262 CKM_SHA3_384_HMAC

10263 CKM_SHA3_384_HMAC_GENERAL

10264 CKM_SHA3_384_KEY_DERIVATION

10265 CKM_SHA3_384_KEY_GEN

10266

10267 CKK_SHA3_384_HMAC

10268 **6.30.2 SHA3-384 digest**

10269 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.

10271 It does not have a parameter.

10272 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.

10274 Table 164, SHA3-384: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

10275 **6.30.3 General-length SHA3-384-HMAC**

10276 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.

10278 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).

10279 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

10280 Table 165, General-length SHA3-384-HMAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA3_384_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-48, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_SHA3_384_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA3_512</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC_GENERAL</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SHA3_512_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA3_512_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_SHA3_512_HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_SHA3_512_HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
</tbody>
</table>

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.
The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the SHA3-512-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.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of CKM_SHA3_512_HMAC key sizes, in bytes.

### 6.32 SHAKE

#### Table 169, SHA-512 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHAKE_128_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHAKE_256_KEY_DERIVATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

#### 6.32.1 Definitions

- CKM_SHAKE_128_KEY_DERIVATION
- CKM_SHAKE_256_KEY_DERIVATION

#### 6.32.2 SHAKE Key Derivation

SHAKE-128 and SHAKE-256 key derivation, denoted CKM_SHAKE_128_KEY_DERIVATION and CKM_SHAKE_256_KEY_DERIVATION, implements the SHAKE expansion function defined in FIPS 202 on the input key.

- If no length or key type is provided in the template a CKR_TEMPLATE_INCOMPLETE error is generated.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism shall 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 shall be of the type specified in the template. If it doesn't, an error shall 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 shall 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 shall be set properly.

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 shall 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 shall, 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.
6.33 BLAKE2B-160

Table 170, BLAKE2B-160 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLAKE2B_160</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_HMAC</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_HMAC_GENERAL</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_BLAKE2B_160_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

6.33.1 Definitions

Mechanisms:
- CKM_BLAKE2B_160
- CKM_BLAKE2B_160_HMAC
- CKM_BLAKE2B_160_HMAC_GENERAL
- CKM_BLAKE2B_160_KEY_DERIVE
- CKM_BLAKE2B_160_KEY_GEN
- CKK_BLAKE2B_160_HMAC

6.33.2 BLAKE2B-160 digest

The BLAKE2B-160 mechanism, denoted CKM_BLAKE2B_160, is a mechanism for message digesting, following the Blake2b Algorithm with a 160-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 171, BLAKE2B-160: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>20</td>
</tr>
</tbody>
</table>

6.33.3 General-length BLAKE2B-160-HMAC

The general-length BLAKE2B-160-HMAC mechanism, denoted CKM_BLAKE2B_160_HMAC_GENERAL, is the keyed variant of BLAKE2b-160 and length of the output should be in the range 1-20. The keys it uses are generic secret keys and CKK_BLAKE2B_160_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 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

Table 172, General-length BLAKE2B-160-HMAC: Key And Data Length
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_BLAKE2B_160_HMAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or</td>
<td>Any</td>
<td>1-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td>CKK_BLAKE2B_160_HMAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.33.4 BLAKE2B-160-HMAC

The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the general-length BLAKE2B-160-HMAC mechanism.

It has no parameter, and always produces an output of length 20.

6.33.5 BLAKE2B-160 key derivation

BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the BLAKE2B-160 hash function and the relevant length is 20 bytes.

6.33.6 BLAKE2B-160 HMAC key generation

The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a key generation mechanism for BLAKE2B-160-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-160-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-160-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_160_HMAC** key sizes, in bytes.

6.34 BLAKE2B-256

*Table 173, BLAKE2B-256 Mechanisms vs. Functions*
### 6.34.1 Definitions

#### Mechanisms:

- **CKM_BLAKE2B_256**
- **CKM_BLAKE2B_256_HMAC_GENERAL**
- **CKM_BLAKE2B_256_HMAC**
- **CKM_BLAKE2B_256_KEY_DERIVE**
- **CKM_BLAKE2B_256_KEY_GEN**

#### 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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>32</td>
</tr>
</tbody>
</table>

#### 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**
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_BLAKE2B_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_BLAKE2B_256_HMAC</td>
<td>Any</td>
<td>1-32, depending on parameters</td>
</tr>
</tbody>
</table>

### 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*
6.35.1 Definitions

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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>48</td>
</tr>
</tbody>
</table>

6.35.2 BLAKE2B-384 digest

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
### 6.35.4 BLAKE2B-384-HMAC

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
6.36.1 Definitions

CKM_BLAKE2B_512
CKM_BLAKE2B_512_HMAC_GENERAL
CKM_BLAKE2B_512_HMAC
CKM_BLAKE2B_512_KEY_DERIVE
CKM_BLAKE2B_512_KEY_GEN

6.36.2 BLAKE2B-512 digest

The BLAKE2B-512 mechanism, denoted CKM_BLAKE2B_512, is a mechanism for message digesting, following the Blake2b Algorithm with a 512-bit message digest 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 180, BLAKE2B-512: Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>64</td>
</tr>
</tbody>
</table>

6.36.3 General-length BLAKE2B-512-HMAC

The general-length BLAKE2B-512-HMAC mechanism, denoted CKM_BLAKE2B_512_HMAC_GENERAL, is the keyed variant of the BLAKE2B-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_BLAKE2B_512_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 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 181, General-length BLAKE2B-512-HMAC: Key And Data Length
### Function Table

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret or CKK_BLAKE2B_512_HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret or CKK_BLAKE2B_512_HMAC</td>
<td>Any</td>
<td>1-64, depending on parameters</td>
</tr>
</tbody>
</table>

### 6.36.4 BLAKE2B-512-HMAC

The BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC**, is a special case of the general-length BLAKE2B-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### 6.36.5 BLAKE2B-512 key derivation

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

The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM_BLAKE2B_512_KEY_GEN**, is a key generation mechanism for NIST's BLAKE2B-512-HMAC.

It does not have a parameter.

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.

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.

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.

### 6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

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 #5-1].

**Table 182, PKCS 5 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_PBE_SHA1_DES3_EDE_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_DES2_EDE_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_PBA_SHA1_WITH_SHA1_HMAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.37.1 Definitions

6.37.2 Password-based encryption/authentication mechanism parameters

CK_PBE_PARAMS is a structure which provides all of the necessary information required by the
CKM_PBE mechanisms (see [PKCS #5] and [PKCS #12] for information on the PBE generation
mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

typedef struct CK_PBE_PARAMS {
    CK_BYTE_PTR pInitVector;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG ulPasswordLen;
    CK_ULONG ulIteration;
    CK_BYTE_PTR pSalt;
    CK_ULONG ulSaltLen;
} CK_PBE_PARAMS;

The fields of the structure have the following meanings:

pInitVector pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;

pPassword points to the password to be used in the PBE key generation;

ulPasswordLen length in bytes of the password information;

pSalt points to the salt to be used in the PBE key generation;

ulSaltLen length in bytes of the salt information;

ulIteration number of iterations required for the generation.

CK_PBE_PARAMS_PTR is a pointer to a CK_PBE_PARAMS.
6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters

- **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;**
- **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR**

**CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE** is used to indicate the Pseudo-Random Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

```c
typedef CK_ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;
```

The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

### Table 183, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions

<table>
<thead>
<tr>
<th>PRF Identifier</th>
<th>Value</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA1</td>
<td>0x00000001UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_GOSTR3411</td>
<td>0x00000002UL</td>
<td>This PRF uses GOST R34.11-94 hash to produce secret key value. <em>pPrfData</em> should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. <em>ulPrfDataLen</em> holds encoded OID length in bytes. If <em>pPrfData</em> is set to NULL_PTR, then id-GostR3411-94-CryptoProParamSet parameters will be used (RFC 4357, 11.2), and <em>ulPrfDataLen</em> must be 0.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA224</td>
<td>0x00000003UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA256</td>
<td>0x00000004UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA384</td>
<td>0x00000005UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA512</td>
<td>0x00000006UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA512_224</td>
<td>0x00000007UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
<tr>
<td>CK_PKCS5_PBKD2_HMAC_SHA512_256</td>
<td>0x00000008UL</td>
<td>No Parameter. <em>pPrfData</em> must be NULL and <em>ulPrfDataLen</em> must be zero.</td>
</tr>
</tbody>
</table>

**CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR** is a pointer to a **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE**.
CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE is used to indicate the source of the salt value when deriving a key using PKCS #5 PBKDF2. It is defined as follows:

typedef CK_ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;

The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined sources along with the corresponding data type for the pSaltSourceData field in the CK_PKCS5_PBKDF2_PARAMS2 structure defined below.

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKZ_SALT_SPECIFIED</td>
<td>0x00000001</td>
<td>Array of CK_BYTE containing the value of the salt value.</td>
</tr>
</tbody>
</table>

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR is a pointer to a CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE.

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

CK_PKCS5_PBKDF2_PARAMS2; CK_PKCS5_PBKDF2_PARAMS2_PTR

CK_PKCS5_PBKDF2_PARAMS2 is a structure that provides the parameters to the CKM_PKCS5_PBKDF2 mechanism. The structure is defined as follows:

typedef struct CK_PKCS5_PBKDF2_PARAMS2 {
    CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE saltSource;
    CK_VOID_PTR pSaltSourceData;
    CK_ULONG ulSaltSourceDataLen;
    CK_ULONG iterations;
    CK_PKCS5_PBKDF2_PSEUDO_RANDOM_FUNCTION_TYPE prf;
    CK_VOID_PTR pPrfData;
    CK_ULONG ulPrfDataLen;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG ulPasswordLen;
} CK_PKCS5_PBKDF2_PARAMS2;

The fields of the structure have the following meanings:

- saltSource source of the salt value
- pSaltSourceData data used as the input for the salt source
- ulSaltSourceDataLen length of the salt source input
- iterations number of iterations to perform when generating each block of random data
- prf pseudo-random function used to generate the key
- pPrfData data used as the input for PRF in addition to the salt value
- ulPrfDataLen length of the input data for the PRF
- pPassword points to the password to be used in the PBE key generation
- ulPasswordLen length in bytes of the password information

CK_PKCS5_PBKDF2_PARAMS2_PTR is a pointer to a CK_PKCS5_PBKDF2_PARAMS2.
6.37.4 PKCS #5 PBKDF2 key generation

PKCS #5 PBKDF2 key generation, denoted CKM_PKCS5_PBKD2, is a mechanism used for generating a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

It has a parameter, a CK_PKCS5_PBKD2_PARAMS2 structure. The parameter specifies the salt value source, pseudo-random function, and iteration count used to generate the new key.

Since this mechanism can be used to generate any type of secret key, new key templates must contain the CKA_KEY_TYPE and CKA_VALUE_LEN attributes. If the key type has a fixed length the CKA_VALUE_LEN attribute may be omitted.

6.38 PKCS #12 password-based encryption/authentication mechanisms

The mechanisms in this section are for generating keys and IVs for performing password-based encryption or authentication. The method used to generate keys and IVs is based on a method that was specified in [PKCS #12].

We specify here a general method for producing various types of pseudo-random bits from a password, $p$; a string of salt bits, $s$; and an iteration count, $c$. The “type” of pseudo-random bits to be produced is identified by an identification byte, $ID$, the meaning of which will be discussed later.

Let $H$ be a hash function built around a compression function $f: \mathbb{Z}^m \times \mathbb{Z}^v \rightarrow \mathbb{Z}^w$ (that is, $H$ has a chaining variable and output of length $u$ bits, and the message input to the compression function of $H$ is $v$ bits).

For MD2 and MD5, $u=128$ and $v=512$; for SHA-1, $u=160$ and $v=512$.

We assume here that $u$ and $v$ are both multiples of 8, as are the lengths in bits of the password and salt strings and the number $n$ of pseudo-random bits required. In addition, $u$ and $v$ are of course nonzero.

1. Construct a string, $D$ (the “diversifier”), by concatenating $v/8$ copies of $ID$.
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 salt may be truncated to create $S$). Note that if the salt is the empty string, then so is $S$.
3. Concatenate copies of the password together to create a string $P$ of length $v\lceil p/v \rceil$ bits (the final copy of the password may be truncated to create $P$). Note that if the password is the empty string, then so is $P$.
4. Set $i=S||P$ to be the concatenation of $S$ and $P$.
5. Set $j=n/u$.
6. For $i=1, 2, \ldots, j$, do the following:
   a. Set $A_i=H^c(D||I_i)$, the $c$th hash of $D||I_i$. That is, compute the hash of $D||I_i$; compute the hash of that hash; etc.; continue in this fashion until a total of $c$ hashes have been computed, each on the result of the previous hash.
   b. Concatenate copies of $A_i$ to create a string $B$ of length $v$ bits (the final copy of $A_i$ may be truncated to create $B$).
   c. Treating $I$ as a concatenation $I_0, I_1, \ldots, I_{k-1}$ of $v$-bit blocks, where $k=\lceil s/v \rceil + \lceil p/v \rceil$, modify $I$ by setting $I=(I+\theta B+1) \mod 2^v$ for each $j$. To perform this addition, treat each $v$-bit block as a binary number represented most-significant bit first.
7. Concatenate $A_1, A_2, \ldots, A_j$ together to form a pseudo-random bit string, $A$.
8. Use the first $n$ bits of $A$ as the output of this entire process.

When the password-based encryption mechanisms presented in this section are used to generate a key and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate a key, the identifier byte $ID$ is set to the value 1; to generate an IV, the identifier byte $ID$ is set to the value 2.

When the password based authentication mechanism presented in this section is used to generate a key from a password, salt, and an iteration count, the above algorithm is used. The identifier byte $ID$ is set to 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 \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encryp. &amp; Decryp.</th>
<th>Sign. &amp; Verif.</th>
<th>SR &amp; VR ¹</th>
<th>Digests</th>
<th>Gen. Key / Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SSL3_PRE_MASTER_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS_PRE_MASTER_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Encrypt &amp; Decrypt</td>
<td>Sign &amp; Verify</td>
<td>SR &amp; VR</td>
<td>Digest</td>
<td>Gen · Key / Key Pair</td>
<td>Wrap &amp; Unwrap</td>
<td>Derive</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------</td>
<td>---------------</td>
<td>--------</td>
<td>--------</td>
<td>---------------------</td>
<td>---------------</td>
<td>--------</td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE_DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_KEY_AND_MAC_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SSL3_MD5_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_SHA1_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.39.1 Definitions

**Mechanisms:**

- CKM_SSL3_PRE_MASTER_KEY_GEN
- CKM_TLS_PRE_MASTER_KEY_GEN
- CKM_SSL3_MASTER_KEY_DERIVE
- CKM_SSL3_KEY_AND_MAC_DERIVE
- CKM_SSL3_MASTER_KEY_DERIVE_DH
- CKM_SSL3_MD5_MAC
- CKM_SSL3_SHA1_MAC

### 6.39.2 SSL mechanism parameters

#### CKM_SSL3_RANDOM_DATA

**CKM_SSL3_RANDOM_DATA** is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the CKM_SSL3_MASTER_KEY_DERIVE and the CKM_SSL3_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```c
typedef struct CK_SSL3_RANDOM_DATA {
    CK_BYTE_PTR pClientRandom;
    CKULONG ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CKULONG ulServerRandomLen;
} CK_SSL3_RANDOM_DATA;
```

The fields of the structure have the following meanings:

- `pClientRandom`: pointer to the client’s random data
- `ulClientRandomLen`: length in bytes of the client’s random data
- `pServerRandom`: pointer to the server’s random data
- `ulServerRandomLen`: length in bytes of the server’s random data
CK_SSL3_MASTER_KEY_DERIVE_PARAMS; CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR

CK_SSL3_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the
CKM_SSL3_MASTER_KEY_DERIVE mechanism. It is defined as follows:

typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA    RandomInfo;
    CK_VERSION_PTR         pVersion;
} CK_SSL3_MASTER_KEY_DERIVE_PARAMS;

The fields of the structure have the following meanings:

RandomInfo  client's and server's random data information.
pVersion    pointer to a CK_VERSION structure which receives the SSL
            protocol version information

CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a
CK_SSL3_MASTER_KEY_DERIVE_PARAMS.

CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR

CK_SSL3_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors
after performing a C_DeriveKey function with the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism. It
is defined as follows:

typedef struct CK_SSL3_KEY_MAT_OUT {
    CK_OBJECT_HANDLE  hClientMacSecret;
    CK_OBJECT_HANDLE  hServerMacSecret;
    CK_OBJECT_HANDLE  hClientKey;
    CK_OBJECT_HANDLE  hServerKey;
    CK_BYTE_PTR       pIVClient;
    CK_BYTE_PTR       pIVServer;
} CK_SSL3_KEY_MAT_OUT;

The fields of the structure have the following meanings:

hClientMacSecret  key handle for the resulting Client MAC Secret key
hServerMacSecret  key handle for the resulting Server MAC Secret key
hClientKey        key handle for the resulting Client Secret key
hServerKey        key handle for the resulting Server Secret key
pIVClient         pointer to a location which receives the initialization vector (IV)
                   created for the client (if any)
pIVServer         pointer to a location which receives the initialization vector (IV)
                   created for the server (if any)

CK_SSL3_KEY_MAT_OUT_PTR is a pointer to a CK_SSL3_KEY_MAT_OUT.

CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR

CK_SSL3_KEY_MAT_PARAMS is a structure that provides the parameters to the
CKM_SSL3_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:
typedef struct CK_SSL3_KEY_MAT_PARAMS {
    CK_ULONG ulMacSizeInBits;
    CK_ULONG ulKeySizeInBits;
    CK_ULONG ulIVSizeInBits;
    CK_BBOOL bIsExport;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_SSL3_KEY_MAT_PARAMS;

The fields of the structure have the following meanings:

ulMacSizeInBits: the length (in bits) of the MACing keys agreed upon during the
protocol handshake phase.

ulKeySizeInBits: the length (in bits) of the secret keys agreed upon during the
protocol handshake phase.

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.

bIsExport: a Boolean value which indicates whether the keys have to be
derived for an export version of the protocol.

RandomInfo: client's and server's random data information.

pReturnedKeyMaterial: points to a CK_SSL3_KEY_MAT_OUT structures which receives
the handles for the keys generated and the IVs.

6.39.3 Pre-master key generation

Pre-master key generation in SSL 3.0, denoted CKM_SSL3_PRE_MASTER_KEY_GEN, is a mechanism
which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL
version 3.0 for RSA-like cipher suites.

It has one parameter, a CK_VERSION structure, which provides the client's SSL version number.

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.

The template sent along with this mechanism during a C_GenerateKey call may indicate that the object
class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN
attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
both indicate 48 bytes.

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.

6.39.4 Master key derivation

Master key derivation in SSL 3.0, denoted CKM_SSL3_MASTER_KEY_DERIVE, is a mechanism used
to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce
the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the
value of the client version, which is built into the "pre_master" key as well as a handle to the derived
"master_secret" key.
It has a parameter, a `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 6.39.

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; otherwise they 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`, the key type is `CKK_GENERIC_SECRET`, and the `CKA_VALUE_LEN` attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The `CKA_SENSITIVE` and `CKA_EXTRACTABLE` attributes in the template for the new key can both be specified to be either `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.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure both indicate 48 bytes.

Note that the `CK_VERSION` structure pointed to by the `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure’s `pVersion` field will be modified by the `C_DeriveKey` call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

### 6.39.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in SSL 3.0, denoted `CKM_SSL3_MASTER_KEY_DERIVE_DH`, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key.

It has a parameter, a `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure, which allows for the passing of random data to the token. This structure is defined in Section 6.39. The `pVersion` field of the structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

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.

The template sent along with this mechanism during a `C_DeriveKey` call may indicate that the object class is `CKO_SECRET_KEY`, the key type is `CKK_GENERIC_SECRET`, and the `CKA_VALUE_LEN` attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The `CKA_SENSITIVE` and `CKA_EXTRACTABLE` attributes in the template for the new key can both be specified to be either `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.

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 `ulIVSizeInBits` 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 `ulIVSizeInBits` 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 `NULL_PTR`.

pkcs11-spec-v3.1-cs01
Standards Track Work Product

Copyright © OASIS Open 2022. All Rights Reserved.

11 August 2022
Page 310 of 426
If a call to C_DeriveKey with this mechanism fails, then none of the four keys will be created on the token.

### 6.39.7 MD5 MACing in SSL 3.0

MD5 MACing in SSL3.0, denoted CKM_SSL3_MD5_MAC, is a mechanism for single- and multiple-part signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 186, MD5 MACing in SSL 3.0: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of generic secret key sizes, in bits.

### 6.39.8 SHA-1 MACing in SSL 3.0

SHA-1 MACing in SSL3.0, denoted CKM_SSL3_SHA1_MAC, is a mechanism for single- and multiple-part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 187, SHA-1 MACing in SSL 3.0: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>4-8, depending on parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of generic secret key sizes, in bits.

### 6.40 TLS 1.2 Mechanisms

Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2 mechanisms contains a new value in the parameters structure to specify the hash function.

This section also specifies CKM_TLS12_MAC which should be used in place of CKM_TLS_PRF to calculate the verify_data in the TLS “finished” message.

This section also specifies CKM_TLS_KDF that can be used in place of CKM_TLS_PRF to implement key material exporters.

Table 188, TLS 1.2 Mechanisms vs. Functions
<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_TLS12_MASTER_KEY_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_MASTER_KEY_DERIVE_DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KEY_AND_MAC_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KEY_SAFE_DERIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_TLS12_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_TLS12_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### 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**

```c
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**.
CK_TLS12_KEY_MAT_PARAMS is a structure that provides the parameters to the CKM_TLS12_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_TLS12_KEY_MAT_PARAMS {
    CK_ULONG ulMacSizeInBits;
    CK_ULONG ulKeySizeInBits;
    CK_ULONG ulIVSizeInBits;
    CK_BBOOL bIsExport;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TLS12_KEY_MAT_PARAMS;
```

The fields of the structure have the following meanings:

- **ulMacSizeInBits**: the length (in bits) of the MACing keys agreed upon during the protocol handshake phase. If no MAC key is required, the length should be set to 0.
- **ulKeySizeInBits**: the length (in bits) of the secret keys agreed upon during the protocol handshake phase.
- **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.
- **bIsExport**: must be set to CK_FALSE because export cipher suites must not be used in TLS 1.1 and later.
- **RandomInfo**: client's and server's random data information.
- **pReturnedKeyMaterial**: points to a CK_SSL3_KEY_MAT_OUT structures which receives the handles for the keys generated and the IVs.
- **prfHashMechanism**: base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

CK_TLS12_KEY_MAT_PARAMS_PTR is a pointer to a CK_TLS12_KEY_MAT_PARAMS.

CK_TLS_KDF_PARAMS is a structure that provides the parameters to the CKM_TLS_KDF mechanism. It is defined as follows:

```c
typedef struct CK_TLS_KDF_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BYTE_PTR pLabel;
    CK_ULONG ulLabelLength;
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_BYTE_PTR pContextData;
    CK_ULONG ulContextDataLength;
} CK_TLS_KDF_PARAMS;
```

The fields of the structure have the following meanings:

- **prfMechanism**: the hash mechanism used in the TLS1.2 PRF construct or CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
pLabel     a pointer to the label for this key derivation
ulLabelLength length of the label in bytes
RandomInfo the random data for the key derivation
pContextData a pointer to the context data for this key derivation. NULL_PTR if not present
ulContextDataLength length of the context data in bytes. 0 if not present.

CK_TLS_KDF_PARAMS_PTR is a pointer to a CK_TLS_KDF_PARAMS.

∗ CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR

CK_TLS_MAC_PARAMS is a structure that provides the parameters to the CKM_TLS_MAC mechanism. It is defined as follows:

```c
typedef struct CK_TLS_MAC_PARAMS {
    CK_MECHANISM_TYPE prfHashMechanism;
    CK_ULONG ulMacLength;
    CK_ULONG ulServerOrClient;
} CK_TLS_MAC_PARAMS;
```

The fields of the structure have the following meanings:

- prfHashMechanism the hash mechanism used in the TLS12 PRF construct or CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
- ulMacLength 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.
- ulServerOrClient 1 to use the label "server finished", 2 to use the label "client finished". All other values are invalid.

CK_TLS_MAC_PARAMS_PTR is a pointer to a CK_TLS_MAC_PARAMS.

∗ CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR

CK_TLS_PRF_PARAMS is a structure, which provides the parameters to the CKM_TLS_PRF mechanism. It is defined as follows:

```c
typedef struct CK_TLS_PRF_PARAMS {
    CK_BYTE_PTR pSeed;
    CK_ULONG ulSeedLen;
    CK_BYTE_PTR pLabel;
    CK_ULONG ulLabelLen;
    CK_BYTE_PTR pOutput;
    CK_ULONG_PTR pulOutputLen;
} CK_TLS_PRF_PARAMS;
```

The fields of the structure have the following meanings:

- pSeed pointer to the input seed
- ulSeedLen length in bytes of the input seed
- pLabel pointer to the identifying label
pkcs11-spec-v3.1-cs01
Standards Track Work Product Copyright © OASIS Open 2022. All Rights Reserved.

6.40.3 TLS MAC

The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces the use of the CKM_TLS_PRF function for TLS1.0 and 1.1 and that mechanism is deprecated.

CKM_TLS_MAC takes a parameter of CK_TLS_MAC_PARAMS. To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

In TLS1.0 and 1.1 the "finished" message verify_data (i.e. the output signature from the MAC mechanism) is always 12 bytes. In TLS1.2 the "finished" message verify_data is a minimum of 12 bytes, defaults to 12 bytes, but may be negotiated to longer length.

Table 189, General-length TLS MAC: Key And Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>generic secret</td>
<td>any</td>
<td>&gt;=12 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>&gt;=12 bytes</td>
</tr>
</tbody>
</table>

6.40.4 Master key derivation

Master key derivation in TLS 1.0, denoted CKM_TLS_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 TLS 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.

It has a parameter, a CK_SSL3_MASTER_KEY_DERIVE_PARAMS structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 6.39.

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.

The mechanism also contributes the CKA_ALLOWED_MECHANISMS attribute consisting only of CKM_TLS12_KEY_AND_MAC_DERIVE, CKM_TLS12_KEY_SAFE_DERIVE, CKM_TLS12_KDF and CKM_TLS12_MAC.

The template sent along with this mechanism during a C_DeriveKey call may indicate that the object class is CKO_SECRET_KEY, the key type is CKK_GENERIC_SECRET, and the CKA_VALUE_LEN attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either 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 \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to the same value as its
\texttt{CKA\_SENSITIVE} attribute.

- Similarly, if the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, then the derived key will, too. If the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to the opposite value from its \texttt{CKA\_EXTRACTABLE} attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the \texttt{CK\_MECHANISM\_INFO} structure both indicate 48 bytes.

Note that the \texttt{CK\_VERSION} structure pointed to by the \texttt{CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS} structure’s pVersion field will be modified by the \texttt{C\_DeriveKey} call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte “pre_master” secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

\subsection*{6.40.5 Master key derivation for Diffie-Hellman}

Master key derivation for Diffie-Hellman in TLS 1.0, denoted \texttt{CKM\_TLS\_MASTER\_KEY\_DERIVE\_DH}, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

It has a parameter, a \texttt{CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS} structure, which allows for the passing of random data to the token. This structure is defined in Section 6.39. The pVersion field of the structure must be set to NULL\_PTR since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key (as well as the \texttt{CKA\_VALUE\_LEN} attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The mechanism also contributes the \texttt{CKA\_ALLOWED\_MECHANISMS} attribute consisting only of \texttt{CKM\_TLS\_1\_KEY\_AND\_MAC\_DERIVE}, \texttt{CKM\_TLS\_1\_KEY\_SAFE\_DERIVE}, \texttt{CKM\_TLS\_12\_KDF} and \texttt{CKM\_TLS\_12\_MAC}.

The template sent along with this mechanism during a \texttt{C\_DeriveKey} call may indicate that the object class is \texttt{CKO\_SECRET\_KEY}, the key type is \texttt{CKK\_GENERIC\_SECRET}, and the \texttt{CKA\_VALUE\_LEN} attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

- The \texttt{CKA\_SENSITIVE} and \texttt{CKA\_EXTRACTABLE} attributes in the template for the new key can both be specified to be either \texttt{CK\_TRUE} or \texttt{CK\_FALSE}. If omitted, these attributes each take on some default value.

- If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_FALSE}, then the derived key will as well. If the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to the same value as its \texttt{CKA\_SENSITIVE} attribute.

- Similarly, if the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, then the derived key will, too. If the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_TRUE}, then the derived key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to the opposite value from its \texttt{CKA\_EXTRACTABLE} attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the \texttt{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.
6.40.6 Key and MAC derivation

Key, MAC and IV derivation in TLS 1.0, denoted CKM_TLS_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") (if present) are always given a type of CKK_GENERIC_SECRET. They are flagged as valid for signing and verification.

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.

For CKM_TLS12_KEY_AND_MAC_DERIVE, IVs will be generated and returned if the ulIVSizeInBits 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 ulIVSizeInBits field.

Note Well: CKM_TLS12_KEY_AND_MAC_DERIVE produces both private (key) and public (IV) data. It is possible to "leak" private data by the simple expedient of decreasing the length of private data requested. E.g. Setting uMacSizeInBits and uKeySizeInBits to 0 (or other lengths less than the key size) will result in the private key data being placed in the destination designated for the IV's. Repeated calls with the same master key and same RandomInfo but with differing lengths for the private key material will result in different data being leaked.<

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 NULL_PTR.

If a call to C_DeriveKey with this mechanism fails, then none of the four keys will be created on the token.

6.40.7 CKM_TLS12_KEY_SAFE_DERIVE

CKM_TLS12_KEY_SAFE_DERIVE is identical to CKM_TLS12_KEY_AND_MAC_DERIVE except that it shall never produce IV data, and the ulIVSizeInBits field of CK_TLS12_KEY_MAT_PARAMS is ignored.
and treated as 0. All of the other conditions and behavior described for CKM_TLS12_KEY_AND_MAC_DERIVE, with the exception of the black box warning, apply to this mechanism.

CKM_TLS12_KEY_SAFE_DERIVE is provided as a separate mechanism to allow a client to control the export of IV material (and possible leaking of key material) through the use of the CKA_ALLOWED_MECHANISMS key attribute.

6.40.8 Generic Key Derivation using the TLS PRF

CKM_TLS_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. CKM_TLS_KDF has a parameter of CK_TLS_KDF_PARAMS. If the protocol using this mechanism does not use context information, the pContextData field shall be set to NULL_PTR and the ulContextDataLength field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to CKM_TLS12_KEY_AND_MAC_DERIVE) by first deriving the key stream as a CKK_GENERIC_SECRET of the necessary length and doing subsequent derives against that derived key using the CKM_EXTRACT_KEY_FROM_KEY mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

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.

6.40.9 Generic Key Derivation using the TLS12 PRF

CKM_TLS12_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. CKM_TLS12_KDF has a parameter of CK_TLS12_KDF_PARAMS. If the protocol using this mechanism does not use context information, the pContextData field shall be set to NULL_PTR and the ulContextDataLength field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use CKM_TLS_PRF as the value for prfMechanism in place of a hash mechanism. Note: Although CKM_TLS_PRF is deprecated as a mechanism for C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to CKM_TLS12_KEY_AND_MAC_DERIVE) by first deriving the key stream as a CKK_GENERIC_SECRET of the necessary length and doing subsequent derives against that derived key stream using the CKM_EXTRACT_KEY_FROM_KEY mechanism to split the key stream into the actual operational keys.
The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

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.

### 6.4.1 WTLS

Details can be found in [WTLS].

When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However, since in WTLS the server and client use different sequence numbers, there could be instances (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and server side differ, and hence this motivates the introduced split.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrpt &amp; Decrpt</th>
<th>Sign &amp; Verif y</th>
<th>SR &amp; VR</th>
<th>Dige st</th>
<th>Gen. Key / Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Deriv e</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_WTLS_PRE_MASTER_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_PRF</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.4.1.1 Definitions

Mechanisms:

- CKM_WTLS_PRE_MASTER_KEY_GEN
- CKM_WTLS_MASTER_KEY_DERIVE
- CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
6.41.2 WTLS mechanism parameters

♦ CK_WTLS_RANDOM_DATA; CK_WTLS RANDOM_DATA_PTR

CK_WTLS_RANDOM_DATA is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the CKM_WTLS_MASTER_KEY_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_WTLS_RANDOM_DATA {
    CK_BYTE_PTR pClientRandom;
    CK_ULONG    ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CK_ULONG    ulServerRandomLen;
} CK_WTLS_RANDOM_DATA;
```

The fields of the structure have the following meanings:
- `pClientRandom`: pointer to the client’s random data
- `ulClientRandomLen`: length in bytes of the client’s random data
- `pServerRandom`: pointer to the server’s random data
- `ulServerRandomLen`: length in bytes of the server’s random data

CK_WTLS_RANDOM_DATA_PTR is a pointer to a CK_WTLS_RANDOM_DATA.

♦ CK_WTLS_MASTER_KEY_DERIVE_PARAMS; CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR

CK_WTLS_MASTER_KEY_DERIVE_PARAMS is a structure, which provides the parameters to the CKM_WTLS_MASTER_KEY_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
    CK_MECHANISM_TYPE   DigestMechanism;
    CK_WTLS_RANDOM_DATA RandomInfo;
    CK_BYTE_PTR         pVersion;
} CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:
- `DigestMechanism`: the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
- `RandomInfo`: Client’s and server’s random data information
- `pVersion`: pointer to a CK_BYTE which receives the WTLS protocol version information

CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a CK_WTLS_MASTER_KEY_DERIVE_PARAMS.
CK_WTLS_PRF_PARAMS is a structure, which provides the parameters to the CKM_WTLS_PRF mechanism. It is defined as follows:

typedef struct CK_WTLS_PRF_PARAMS {
    CK_MECHANISM_TYPE DigestMechanism;
    CK_BYTE_PTR pSeed;
    CK_ULONG ulSeedLen;
    CK_BYTE_PTR pLabel;
    CK_ULONG ulLabelLen;
    CK_BYTE_PTR pOutput;
    CK_ULONG_PTR pulOutputLen;
} CK_WTLS_PRF_PARAMS;

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

CK_WTLS_PRF_PARAMS_PTR is a pointer to a CK_WTLS_PRF_PARAMS.

CK_WTLS_KEY_MAT_OUT is a structure that contains the resulting key handles and initialization vectors after performing a C_DeriveKey function with the CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE or with the CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

typedef struct CK_WTLS_KEY_MAT_OUT {
    CK_OBJECT_HANDLE hMacSecret;
    CK_OBJECT_HANDLE hKey;
    CK_BYTE_PTR pIV;
} CK_WTLS_KEY_MAT_OUT;

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)

CK_WTLS_KEY_MAT_OUT_PTR is a pointer to a CK_WTLS_KEY_MAT_OUT.
CK_WTLS_KEY_MAT_PARAMS is a structure that provides the parameters to the
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE and the
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```c
typedef struct CK_WTLS_KEY_MAT_PARAMS {
    CK_MECHANISM_TYPE       DigestMechanism;
    CK_ULONG                ulMacSizeInBits;
    CK_ULONG                ulKeySizeInBits;
    CK_ULONG                ulIVSizeInBits;
    CK_ULONG                ulSequenceNumber;
    CK_BBOOL                bIsExport;
    CK_WTLS_RANDOM_DATA     RandomInfo;
    CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
} CK_WTLS_KEY_MAT_PARAMS;
```

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])
- **ulMaxSizeInBits**
  - the length (in bits) of the MACing key agreed upon during the
    protocol handshake phase
- **ulKeySizeInBits**
  - the length (in bits) of the secret key agreed upon during the
    handshake phase
- **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.
- **ulSequenceNumber**
  - the current sequence number used for records sent by the client
    and server respectively
- **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 CKA_KEY_TYPE or the CKA_VALUE_LEN attribute).
- **RandomInfo**
  - client's and server's random data information
- **pReturnedKeyMaterial**
  - points to a CK_WTLS_KEY_MAT_OUT structure which receives
    the handles for the keys generated and the IV

CK_WTLS_KEY_MAT_PARAMS_PTR is a pointer to a CK_WTLS_KEY_MAT_PARAMS.

### 6.41.3 Pre master secret key generation for RSA key exchange suite

Pre master secret key generation for the RSA key exchange suite in WTLS denoted
CKM_WTLS_PRE_MASTER_KEY_GEN, is a mechanism, which generates a variable length secret key.
It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This
mechanism returns a handle to the pre master secret key.

It has one parameter, a CK_BYTE, which provides the client's WTLS version.
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.
6.41.4 Master secret key derivation

Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.

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.

The template sent along with this mechanism during a **C_DeriveKey** 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 20. 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 have the same value as its **CKA_SENSITIVE** attribute.

Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE, then the derived key will be CK_FALSE, and the derived key’s **CKA_VALUE_LEN** 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 20 bytes.

Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure’s **pVersion** field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token.

The **pVersion** field of the structure must be set to NULL_PTR since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.
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.

The template sent along with this mechanism during a C_DeriveKey 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 20. 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.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure both indicate 20 bytes.

Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

6.41.6 WTLS PRF (pseudorandom function)

PRF (pseudo random function) in WTLS, denoted CKM_WTLS_PRF, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a CK_WTLS_PRF_PARAMS structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a C_DeriveKey function call, which means the template shall be a NULL_PTR. For most key-derivation mechanisms, C_DeriveKey returns a single key handle as a result of a successful completion. However, since the CKM_WTLS_PRF mechanism returns the requested number of output bytes in the CK_WTLS_PRF_PARAMS structure specified as the mechanism parameter, the parameter phKey passed to C_DeriveKey is unnecessary, and should be a NULL_PTR. If a call to C_DeriveKey with this mechanism fails, then no output will be generated.

6.41.7 Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite 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 IV created.

It has a parameter, a CK_WTLS_KEY_MAT_PARAMS structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.
This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of CKO_SECRET_KEY.

The MACing key (server write MAC secret) is always given a type of CKK_GENERIC_SECRET. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a C_DeriveKey function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the ulIVSizeInBits field of the CK_WTLS_KEY_MAT_PARAMS field has a nonzero value. If it is generated, its length in bits will agree with the value in the ulIVSizeInBits field.

Both 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 that differ from those held by the base key.

Note that the CK_WTLS_KEY_MAT_OUT structure pointed to by the CK_WTLS_KEY_MAT_PARAMS structure’s pReturnedKeyMaterial field will be modified by the C_DeriveKey call. In particular, the two key handle fields in the CK_WTLS_KEY_MAT_OUT structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the CK_WTLS_KEY_MAT_OUT structure’s pIV field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV 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_WTLS_SERVER_KEY_AND_MAC_DERIVE mechanism returns all of its key handles in the CK_WTLS_KEY_MAT_OUT structure pointed to by the CK_WTLS_KEY_MAT_PARAMS structure specified as the mechanism parameter, the parameter phKey passed to C_DeriveKey is unnecessary, and should be a NULL_PTR.

If a call to C_DeriveKey with this mechanism fails, then none of the two keys will be created.

### 6.41.8 Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite 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 IV created.

It has a parameter, a CK_WTLS_KEY_MAT_PARAMS structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of CKO_SECRET_KEY.

The MACing key (client write MAC secret) is always given a type of CKK_GENERIC_SECRET. It is flagged as valid for signing, verification and derivation operations.

The other key (client write key) is typed according to information found in the template sent along with this mechanism during a C_DeriveKey function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (client write IV) will be generated and returned if the ulIVSizeInBits field of the CK_WTLS_KEY_MAT_PARAMS field has a nonzero value. If it is generated, its length in bits will agree with the value in the ulIVSizeInBits field.

Both 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 that differ from those held by the base key.
Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure’s `pReturnedKeyMaterial` field will be modified by the **C_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure’s `pIV` field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV 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_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter `phKey` passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

If a call to **C_DeriveKey** with this mechanism fails, then none of the two keys will be created.

### 6.42 SP 800-108 Key Derivation

NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a Feedback Mode KDF and a Double Pipeline Mode KDF.

This section defines a unique mechanism for each type of KDF. These mechanisms can be used to derive one or more symmetric keys from a single base symmetric key.

The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the PRFs accept two pieces of input; a base key and some input data. The base key is taken from the `hBaseKey` parameter to **C_Derive**. The input data is constructed from an iteration variable (internally defined by the KDF/PRF) and the data provided in the **CK_PRF_DATA_PARAM** array that is part of the mechanism parameter.

#### Table 191, SP800-108 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SP800_108_COUNTER_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SP800_108_FEEDBACK_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SP800_108_DOUBLE_PIPELINE_KDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

For these mechanisms, the `ulMinKeySize` and `ulMaxKeySize` fields of the **CK_MECHANISM_INFO** structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms support multiple PRF types and key types; as such the values reported by `ulMinKeySize` and `ulMaxKeySize` specify the minimum and maximum supported base key size when all PRF and keys types are considered. For example, a Cryptoki implementation may support `CKK_GENERIC_SECRET` keys that can be as small as 8-bits in length and therefore `ulMinKeySize` could report 8-bits. However, for an AES-CMAC PRF the base key must be of type `CKK_AES` and must be either 16-bytes, 24-bytes or 32-bytes in lengths and therefore the value reported by `ulMinKeySize` could be misleading. Depending on the PRF type selected, additional key size restrictions may apply.

### 6.42.1 Definitions

#### Mechanisms:

```
CKM_SP800_108_COUNTER_KDF
CKM_SP800_108_FEEDBACK_KDF
```

---

11592 11593 11594 11595 11596 11597 11598 11599 11600 11601 11602 11603 11604 11605 11606 11607 11608 11609 11610 11611 11612 11613 11614 11615 11616 11617 11618 11619 11620 11621 11622 11623 11624 11625 11626 11627 11628 11629
Data Field Types:
- CK_SP800_108_ITERATION_VARIABLE
- CK_SP800_108_COUNTER
- CK_SP800_108_DKM_LENGTH
- CK_SP800_108_BYTE_ARRAY

DKM Length Methods:
- CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS
- CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS

6.42.2 Mechanism Parameters

◆ CK_SP800_108_PRF_TYPE

The CK_SP800_108_PRF_TYPE field of the mechanism parameter is used to specify the type of PRF that is to be used. It is defined as follows:

```
typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;
```

The CK_SP800_108_PRF_TYPE field reuses the existing mechanisms definitions. The following table lists the supported PRF types:

Table 192, SP800-108 Pseudo Random Functions

<table>
<thead>
<tr>
<th>Pseudo Random Function Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SHA_1_HMAC</td>
</tr>
<tr>
<td>CKM_SHA224_HMAC</td>
</tr>
<tr>
<td>CKM_SHA256_HMAC</td>
</tr>
<tr>
<td>CKM_SHA384_HMAC</td>
</tr>
<tr>
<td>CKM_SHA512_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_224_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_256_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_384_HMAC</td>
</tr>
<tr>
<td>CKM_SHA3_512_HMAC</td>
</tr>
<tr>
<td>CKM_DES3_CMAC</td>
</tr>
<tr>
<td>CKM_AES_CMAC</td>
</tr>
</tbody>
</table>

◆ CK_PRF_DATA_TYPE

Each mechanism parameter contains an array of CK_PRF_DATA_PARAM structures. The CK_PRF_DATA_PARAM structure contains CK_PRF_DATA_TYPE field. The CK_PRF_DATA_TYPE field is used to identify the type of data identified by each CK_PRF_DATA_PARAM element in the array. Depending on the type of KDF used, some data field types are mandatory, some data field types are optional and some data field types are not allowed. These requirements are defined on a per-mechanism basis in the sections below. The CK_PRF_DATA_TYPE is defined as follows:

```
typedef CK_ULONG CK_PRF_DATA_TYPE;
```
The following table lists all of the supported data field types:

### Table 193, SP800-108 PRF Data Field Types

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>Identifies the iteration variable defined internally by the KDF.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>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.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>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.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>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.</td>
</tr>
</tbody>
</table>

#### CK_PRF_DATA_PARAM

CK_PRF_DATA_PARAM is used to define a segment of input for the PRF. Each mechanism parameter supports an array of CK_PRF_DATA_PARAM structures. The CK_PRF_DATA_PARAM is defined as follows:

```
typedef struct CK_PRF_DATA_PARAM
{
    CK_PRF_DATA_TYPE    type;
    CK_VOID_PTR         pValue;
    CK_ULONG            ulValueLen;
} CK_PRF_DATA_PARAM;
```

The fields of the CK_PRF_DATA_PARAM structure have the following meaning:

- type defines the type of data pointed to by pValue
- pValue pointer to the data defined by type
- ulValueLen size of the data pointed to by pValue

If the type field of the CK_PRF_DATA_PARAM structure is set to CK_SP800_108_ITERATION_VARIABLE, then pValue must be set the appropriate value for the KDF’s iteration variable type. For the Counter Mode KDF, pValue must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and ulValueLen must be set to sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, pValue must be set to NULL_PTR and ulValueLen must be set to 0.

If the type field of the CK_PRF_DATA_PARAM structure is set to CK_SP800_108_COUNTER, then pValue must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and ulValueLen must be set to sizeof(CK_SP800_108_COUNTER_FORMAT).
If the type field of the `CK_PRF_DATA_PARAM` structure is set to `CK_SP800_108_DKM_LENGTH` then `pvValue` must be assigned a valid `CK_SP800_108_DKM_LENGTH_FORMAT_PTR` and `ulValueLen` must be set to `sizeof(CK_SP800_108_DKM_LENGTH_FORMAT)`.  

If the type field of the `CK_PRF_DATA_PARAM` structure is set to `CK_SP800_108_BYTE_ARRAY`, then `pvValue` must be assigned a valid `CK_BYTE_PTR` value and `ulValueLen` must be set to a non-zero length.

**CK_SP800_108_COUNTER_FORMAT**

`CK_SP800_108_COUNTER_FORMAT` is used to define the encoding format for a counter value. The `CK_SP800_108_COUNTER_FORMAT` is defined as follows:

```c
typedef struct CK_SP800_108_COUNTER_FORMAT
{
    CK_BBOOL    bLittleEndian;
    CK_ULONG    ulWidthInBits;
} CK_SP800_108_COUNTER_FORMAT;
```

The fields of the `CK_SP800_108_COUNTER_FORMAT` structure have the following meaning:

- `bLittleEndian` defines if the counter should be represented in Big Endian or Little Endian format
- `ulWidthInBits` defines the number of bits used to represent the counter value

**CK_SP800_108_DKM_LENGTH_METHOD**

`CK_SP800_108_DKM_LENGTH_METHOD` is used to define how the DKM length value is calculated. The `CK_SP800_108_DKM_LENGTH_METHOD` type is defined as follows:

```c
typedef CK_ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

The following table lists all of the supported DKM Length Methods:

<table>
<thead>
<tr>
<th>DKM Length Method Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS</code></td>
<td>Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF.</td>
</tr>
<tr>
<td><code>CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**CK_SP800_108_DKM_LENGTH_FORMAT**

`CK_SP800_108_DKM_LENGTH_FORMAT` is used to define the encoding format for the DKM length value. The `CK_SP800_108_DKM_LENGTH_FORMAT` is defined as follows:

```c
typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```
The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:

dkmLengthMethod defines the method used to calculate the DKM length value
bLittleEndian defines if the DKM length value should be represented in Big Endian or Little Endian format
ulWidthInBits defines the number of bits used to represent the DKM length value

♦ CK_DERIVED_KEY

CK_DERIVED_KEY is used to define an additional key to be derived as well as provide a CK_OBJECT_HANDLE_PTR to receive the handle for the derived keys. The CK_DERIVED_KEY is defined as follows:

typedef struct CK_DERIVED_KEY
{
    CK_ATTRIBUTE_PTR    pTemplate;
    CK_ULONG             ulAttributeCount;
    CK_OBJECT_HANDLE_PTR phKey;
} CK_DERIVED_KEY;

typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR

The fields of the CK_DERIVED_KEY structure have the following meaning:
pTemplate pointer to a template that defines a key to derive
ulAttributeCount number of attributes in the template pointed to by pTemplate
phKey pointer to receive the handle for a derived key

♦ CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR

CK_SP800_108_KDF_PARAMS is a structure that provides the parameters for the CKM_SP800_108_COUNTER_KDF and CKM_SP800_108_DOUBLE_PIPELINE_KDF mechanisms.

typedef struct CK_SP800_108_KDF_PARAMS
{
    CK_SP800_108_PRF_TYPE    prfType;
    CK_ULONG                 ulNumberOfDataParams;
    CK_PRF_DATA_PARAM_PTR    pDataParams;
    CK_ULONG                 ulAdditionalDerivedKeys;
    CK_DERIVED_KEY_PTR      pAdditionalDerivedKeys;
} CK_SP800_108_KDF_PARAMS;
typedef CK_SP800_108_KDF_PARAMS CK_PTR
CK_SP800_108_KDF_PARAMS_PTR;

The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:

prfType       type of PRF
ulNumberOfDataParams number of elements in the array pointed to by pDataParams
pDataParams   an array of CK_PRF_DATA_PARAM structures. The array defines
               input parameters that are used to construct the “data” input to the
               PRF.
ulAdditionalDerivedKeys number of additional keys that will be derived and the number of
                           elements in the array pointed to by pAdditionalDerivedKeys. If
                           pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
                           be set to 0.
pAdditionalDerivedKeys   an array of CK_DERIVED_KEY structures. If
                           ulAdditionalDerivedKeys is set to 0, this parameter must be set to
                           NULL_PTR

♣ CK_SP800_108_FEEDBACK_KDF_PARAMS,
CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR

typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
{
    CK_SP800_108_PRF_TYPE     prfType;
    CK_ULONG                  ulNumberOfDataParams;
    CK_PRF_DATA_PARAM_PTR     pDataParams;
    CK_ULONG                  ulIVLen;
    CK_BYTE_PTR              pIV;
    CK_ULONG                  ulAdditionalDerivedKeys;
    CK_DERIVED_KEY_PTR       pAdditionalDerivedKeys;
} CK_SP800_108_FEEDBACK_KDF_PARAMS;

typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;

The fields of the CK_SP800_108_FEEDBACK_KDF_PARAMS structure have the following meaning:

prfType       type of PRF
ulNumberOfDataParams number of elements in the array pointed to by pDataParams
pDataParams   an array of CK_PRF_DATA_PARAM structures. The array defines
               input parameters that are used to construct the “data” input to the
               PRF.
ulIVLen   the length in bytes of the IV. If pIV is set to NULL_PTR, this
           parameter must be set to 0.
pIV       an array of bytes to be used as the IV for the feedback mode KDF.
           This parameter is optional and can be set to NULL_PTR. If ulIVLen
           is set to 0, this parameter must be set to NULL_PTR.
6.42.3 Counter Mode KDF

The SP800-108 Counter Mode KDF mechanism, denoted `CKM_SP800_108_COUNTER_KDF`, represents the KDF defined SP800-108 section 5.1. `CKM_SP800_108_COUNTER_KDF` is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a `CK_SP800_108_KDF_PARAMS` structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable for this KDF type is a counter. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</td>
</tr>
<tr>
<td>CK_SP800_108_COUNTER</td>
<td>This data field type is invalid for this KDF type.</td>
</tr>
<tr>
<td>CK_SP800_108_DKM_LENGTH</td>
<td>This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.</td>
</tr>
<tr>
<td>CK_SP800_108_BYTE_ARRAY</td>
<td>This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.</td>
</tr>
</tbody>
</table>

SP800-108 limits the amount of derived keying material that can be produced by a Counter Mode KDF by limiting the internal loop counter to \((2^r-1)\), where “\(r\)” is the number of bits used to represent the counter. Therefore the maximum number of bits that can be produced is \((2^r-1)h\), where “\(h\)” is the length in bits of the output of the selected PRF.

6.42.4 Feedback Mode KDF

The SP800-108 Feedback Mode KDF mechanism, denoted `CKM_SP800_108_FEEDBACK_KDF`, represents the KDF defined SP800-108 section 5.2. `CKM_SP800_108_FEEDBACK_KDF` is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a `CK_SP800_108_FEEDBACK_KDF_PARAMS` structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 196, Feedback Mode data field requirements
**Data Field Identifier** | **Description**
--- | ---
CK_SP800_108_ITERATION_VARIABLE | This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data.
| The iteration variable is defined as K(i-1) in section 5.2 of SP800-108.
| The size, format and value of this data input is defined by the internal KDF structure and PRF output.
| Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.

CK_SP800_108_COUNTER | This data field type is optional. This data field type identifies the location of the counter in the constructed PRF input data.
| Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.
| If specified, only one instance of this type may be specified.

CK_SP800_108_DKM_LENGTH | This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data.
| Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.
| If specified, only one instance of this type may be specified.

CK_SP800_108_BYTE_ARRAY | This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data.
| This standard does not restrict the number of instances of this data type.

SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF by limiting the internal loop counter to \((2^{32} - 1)\). Therefore the maximum number of bits that can be produced is \((2^{32} - 1)h\), where “h” is the length in bits of the output of the selected PRF.

### 6.42.5 Double Pipeline Mode KDF

The SP800-108 Double Pipeline Mode KDF mechanism, denoted CKM_SP800_108_DOUBLE_PIPELINE_KDF, represents the KDF defined SP800-108 section 5.3. CKM_SP800_108_DOUBLE_PIPELINE_KDF is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a CK_SP800_108_KDF_PARAMS structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

**Table 197, Double Pipeline Mode data field requirements**

<table>
<thead>
<tr>
<th>Data Field Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_SP800_108_ITERATION_VARIABLE</td>
<td>This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable is defined as A(i) in section 5.3 of SP800-108.</td>
</tr>
</tbody>
</table>
The size, format and value of this data input is defined by the internal KDF structure and PRF output. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.

**CK_SP800_108_COUNTER**
This data field type is optional. This data field type identifies the location of the counter in the constructed PRF input data. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. If specified, only one instance of this type may be specified.

**CK_SP800_108_DKM_LENGTH**
This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.

**CK_SP800_108_BYTE_ARRAY**
This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.

SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode KDF by limiting the internal loop counter to \((2^{32} - 1)\). Therefore the maximum number of bits that can be produced is \((2^{32} - 1)h\), where “h” is the length in bits of the output of the selected PRF.

The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method used to construct the PRF input data; the data/values identified by the array of **CK_PRF_DATA_PARAM** structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3, the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not included in IV construction process. All other data field types are included in the construction process.

### 6.42.6 Deriving Additional Keys

The KDFs defined in this section can be used to derive more than one symmetric key from the base key. The **C_Derive** function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

To derive additional keys, the mechanism parameter structure can be filled in with one or more **CK_DERIVED_KEY** structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key defined by the **C_Derive** function parameters is always derived before the keys defined by the **CK_DERIVED_KEY** array that is part of the mechanism parameter. The additional keys that are defined by the **CK_DERIVED_KEY** array are derived in the order they are defined in the array. That is to say that the derived keying material produced by the KDF is processed from left to right, and bytes are assigned first to the key defined by the **C_Derive** function parameters, and then bytes are assigned to the keys that are defined by the **CK_DERIVED_KEY** array in the order they are defined in the array.

Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration will produce enough keying material for the key being derived. Other times, additional internal iterations are performed to produce multiple segments which are concatenated together to produce enough keying material for the derived key(s).

When deriving multiple keys, no key can be created using part of a segment that was used for another key. All keys must be created from disjoint segments. For example, if the parameters are defined such
that a 48-byte key (defined by the C_Derive function parameters) and a 16-byte key (defined by the content of CK_DERIVED_KEY) are to be derived using CKM_SHA256_HMAC as a PRF, three internal iterations of the KDF will be performed and three segments of PRF output will be produced. The first segment and half of the second segment will be used to create the 48-byte key and the third segment will be used to create the 16-byte key.

<table>
<thead>
<tr>
<th>3 KDF Segments of Output:</th>
<th>32-byte segment</th>
<th>32-byte segment</th>
<th>32-byte segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Derived Keys:</td>
<td>48-byte key</td>
<td>unused</td>
<td>16-byte key</td>
</tr>
</tbody>
</table>

In the above example, if the CK_SP800_108_DKM_LENGTH data field type is specified with method CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, then the DKM length value will be 512 bits. If the CK_SP800_108_DKM_LENGTH data field type is specified with method CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS, then the DKM length value will be 768 bits.

When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall be derived. If the failure was caused by the content of a specific key’s template (ie the template defined by the content of pTemplate), the corresponding phKey value will be set to CK_INVALID_HANDLE to identify the offending template.

6.42.7 Key Derivation Attribute Rules

The CKM_SP800_108_COUNTER_KDF, CKM_SP800_108_FEEDBACK_KDF and CKM_SP800_108_DOUBLE_PIPELINE_KDF mechanisms have the following rules about key sensitivity and extractability:

- The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key(s) 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.

6.42.8 Constructing PRF Input Data

SP800-108 defines the PRF input data for each KDF at a high level using terms like “label”, “context”, “separator”, “counter”…etc. The value, formatting and order of the input data is not strictly defined by SP800-108, instead it is described as being defined by the “encoding scheme”.

To support any encoding scheme, these mechanisms construct the PRF input data from from the array of CK_PRF_DATA_PARAM structures in the mechanism parameter. All of the values defined by the CK_PRF_DATA_PARAM array are concatenated in the order they are defined and passed in to the PRF as the data parameter.

6.42.8.1 Sample Counter Mode KDF

SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

```
PRF (Ki, [i]2 || Label || 0x00 || Context || [L]2)
```

Section 5.1 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 following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
CK ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
CK ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
    = (CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16);

CK_PRF_DATA_PARAM dataParams[] =
{   CK_SP800_108_ITERATION_VARIABLE,
    &counterFormat, sizeof(counterFormat) },
{   CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
{   CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
{   CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
{   CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
}

CK_SP800_108_KDF_PARAMS kdfParams =
{   CKM_AES_CMAC,
    DIM(dataParams),
    &dataParams,
    0, /* no addition derived keys */
    NULL /* no addition derived keys */
};

CK_MECHANISM = mechanism
{   CK_SP800_108_COUNTER_KDF,
    &kdfParams,
    sizeof(kdfParams)
};

hBaseKey = GetBaseKeyHandle(....);

rv = C_DeriveKey(
    hSession,
    &mechanism,
    hBaseKey,
    &derivedKeyTemplate,
    DIM(derivedKeyTemplate),
    &hDerivedKey);

6.42.8.2 Sample SCP03 Counter Mode KDF

The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

PRF (Ki, Label || 0x00 || [L]2 || [i]2 || Context)
SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```c
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0x0f, 0xed, 0xbe , 0xef};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] =
{  
  { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
  { CK_SP800_108_BYTE_ARRAY, [0x00], 1 },
  { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
  { CK_SP800_108_ITERATION_VARIABLE,
    &counterFormat, sizeof(counterFormat) },
  { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
};

CK_SP800_108_KDF_PARAMS kdfParams =
{  
  CKM_AES_CMAC,
  DIM(dataParams),
  &dataParams,
  0,         /* no addition derived keys */
  NULL       /* no addition derived keys */
};

CK_MECHANISM = mechanism  
{  
  CKM_SP800_108_COUNTER_KDF,
  &kdfParams,
  sizeof(kdfParams)
};

hBaseKey = GetBaseKeyHandle(....);

rv = C_DeriveKey(  
  hSession,
  &mechanism,
  hBaseKey,
  &derivedKeyTemplate,
  DIM(derivedKeyTemplate),
  &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:
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.

```c
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
CK_ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0x0e, 0xde, 0xbe, 0xef};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat =
    {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] =
    {
        { CK_SP800_108_ITERATION_VARIABLE, &counterFormat, sizeof(counterFormat) },
        { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
        { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
        { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
        { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
    };

CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =
    {
        CKM_AES_CMAC,
        DIM(dataParams),
        &dataParams,
        ulFeedbackIVLen,
        baFeedbackIV,
        0, /* no addition derived keys */
        NULL /* no addition derived keys */
    };

CK_MECHANISM = mechanism
    { CKM_SP800_108_FEEDBACK_KDF,
      &kdfParams,
      sizeof(kdfParams)
    };

hBaseKey = GetBaseKeyHandle(....);

rv = C_DeriveKey(
    hSession,
    &mechanism,
    hBaseKey,
    ....);
```
6.42.8.4 Sample Double-Pipeline Mode KDF

SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following PRF inputs:

PRF (KI, A(i-1))
PRF (KI, K(i-1) {|| [i]2 || Label || 0x00 || Context || [L]2)}

Section 5.3 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 so it is left out in this example. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```c
#define DIM(a) (sizeof((a)) / sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xef, 0xed, 0xbe, 0xef};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] =
{ CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
{ CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
{ CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
{ CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }

CK_SP800_108_KDF_PARAMS kdfParams =
{ CKM_AES_CMAC,
  DIM(dataParams),
  &dataParams,
  0, /* no addition derived keys */
  NULL /* no addition derived keys */
};

CK_MECHANISM = mechanism
{ CKM_SP800_108_DOUBLE_PIPELINE_KDF,
  &kdfParams,
  sizeof(kdfParams) }

hBaseKey = GetBaseKeyHandle(....);
rv = C_DeriveKey(
  hSession,
  &derivedKeyTemplate,
  hDerivedKey);
```
&mechanism, hBaseKey, &derivedKeyTemplate, DIM(derivedKeyTemplate), &hDerivedKey);

6.43 Miscellaneous simple key derivation mechanisms

Table 198, Miscellaneous simple key derivation Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_CONCATENATE_BASE_AND_KEY</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CONCATENATE_BASE_AND_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CONCATENATE_DATA_AND_BASE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_XOR_BASE_AND_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_EXTRACT_KEY_FROM_KEY</td>
<td>✓</td>
</tr>
</tbody>
</table>

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:

```c
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.
CK_EXTRACT_PARAMS provides the parameter to the CKM_EXTRACT_KEY_FROM_KEY mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:

```c
typedef CK_ULONG CK_EXTRACT_PARAMS;
```

CK_EXTRACT_PARAMS_PTR is a pointer to a CK_EXTRACT_PARAMS.

### 6.43.3 Concatenation of a base key and another key

This mechanism, denoted CKM_CONCATENATE_BASE_AND_KEY, derives a secret key from the concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys specified are concatenated together in a buffer.

This mechanism takes a parameter, a CK_OBJECT_HANDLE. This handle produces the key value information which is appended to the end of the base key's value information (the base key is the key whose handle is supplied as an argument to C_DeriveKey).

For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the values of the two original keys.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the two original keys' values, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If either of the two original keys has its CKA_SENSITIVE attribute set to 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 either of the two original keys 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 both of the original keys have their CKA_ALWAYS_SENSITIVE attributes set to CK_TRUE.
- Similarly, the derived key's CKA_NEVER_EXTRACTABLE attribute is set to CK_TRUE if and only if both of the original keys have their CKA_NEVER_EXTRACTABLE attributes set to CK_TRUE.

### 6.43.4 Concatenation of a base key and data

This mechanism, denoted CKM_CONCATENATE_BASE_AND_DATA, derives a secret key by concatenating data onto the end of a specified secret key.
This mechanism takes a parameter, a `CK_KEY_DERIVATION_STRING_DATA` structure, which specifies the length and value of the data which will be appended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the value of the original key and the data.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the original key's value and the data, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its `CKA_SENSITIVE` attribute set to 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.

### 6.43.5 Concatenation of data and a base key

This mechanism, denoted `CKM_CONCATENATE_DATA_AND_BASE`, derives a secret key by prepending data to the start of a specified secret key.

This mechanism takes a parameter, a `CK_KEY_DERIVATION_STRING_DATA` structure, which specifies the length and value of the data which will be prepended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.
If the requested type of key requires more bytes than are available by concatenating the data and the
original key's value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its CKA_SENSITIVE attribute set to 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.

6.43.6 XORing of a key and data

XORing key derivation, denoted CKM_XOR_BASE_AND_DATA, is a mechanism which provides the
capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle
and some data.

This mechanism takes a parameter, a CK_KEY_DERIVATION_STRING_DATA structure, which
specifies the data with which to XOR the original key's value.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
the value of the derived key will be taken from a buffer containing the string 0x88888888.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a
generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of
the original key.

- If no key type is provided in the template, but a length is, then the key produced by this mechanism
will be a generic secret key of the specified length.

- If no length is provided in the template, but a key type is, then that key type must have a well-defined
length. If it does, then the key produced by this mechanism will be of the type specified in the
template. If it doesn't, an error will be returned.

- If both a key type and a length are provided in the template, the length must be compatible with that
key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
properly.

If the requested type of key requires more bytes than are available by taking the shorter of the data and
the original key's value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its CKA_SENSITIVE attribute set to 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.
6.43.7 Extraction of one key from another key

Extraction of one key from another key, denoted \texttt{CKM\_EXTRACT\_KEY\_FROM\_KEY}, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a \texttt{CK\_EXTRACT\_PARAMS}, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value \texttt{0x329F84A9}. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the value of the parameter to the \texttt{CKM\_EXTRACT\_KEY\_FROM\_KEY} mechanism is 21).

1. We write the key’s value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this binary string as holding the 32 bits of the key, labeled as $b_0$, $b_1$, ..., $b_{31}$.

2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit $b_{21}$. We obtain the binary string 1001 0101 0010 0110.

3. The value of the new key is thus \texttt{0x9526}.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key’s value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

- If no length or key type is provided in the template, then an error will be returned.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its \texttt{CKA\_SENSITIVE} attribute set to \texttt{CK\_TRUE}, so does the derived key. If not, then the derived key’s \texttt{CKA\_SENSITIVE} attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its \texttt{CKA\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}, so does the derived key. If not, then the derived key’s \texttt{CKA\_EXTRACTABLE} attribute is set either from the supplied template or from a default value.
- The derived key’s \texttt{CKA\_ALWAYS\_SENSITIVE} attribute is set to \texttt{CK\_TRUE} if and only if the base key has its \texttt{CKA\_ALWAYS\_SENSITIVE} attribute set to \texttt{CK\_TRUE}.
- Similarly, the derived key’s \texttt{CKA\_NEVER\_EXTRACTABLE} attribute is set to \texttt{CK\_TRUE} if and only if the base key has its \texttt{CKA\_NEVER\_EXTRACTABLE} attribute set to \texttt{CK\_TRUE}.

6.44 CMS

Table 199, CMS Mechanisms vs. Functions
<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CMS_SIG</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.44.1 Definitions

#### Mechanisms:

- CKM_CMS_SIG

### 6.44.2 CMS Signature Mechanism Objects

These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG mechanism object attributes represent information about supported CMS signature attributes in the token. They are only present on tokens supporting the CKM_CMS_SIG mechanism, but must be present on those tokens.

**Table 200, CMS Signature Mechanism Object Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_REQUIRED_CMS_ATTRIBUTES</td>
<td>Byte array</td>
<td>Attributes the token always will include in the set of CMS signed attributes</td>
</tr>
<tr>
<td>CKA_DEFAULT_CMS_ATTRIBUTES</td>
<td>Byte array</td>
<td>Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application</td>
</tr>
<tr>
<td>CKA_SUPPORTED_CMS_ATTRIBUTES</td>
<td>Byte array</td>
<td>Attributes the token may include in the set of CMS signed attributes upon request by the application</td>
</tr>
</tbody>
</table>

The contents of each byte array will be a DER-encoded list of CMS Attributes with optional accompanying values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-encoded. The list of attributes is defined in ASN.1 as:

```
Attributes ::= SET SIZE (1..MAX) OF Attribute
Attribute ::= SEQUENCE {
  attrType    OBJECT IDENTIFIER,
  attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
            OPTIONAL
}
```

The client may not set any of the attributes.

### 6.44.3 CMS mechanism parameters

- **CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR**

CK_CMS_SIG_PARAMS is a structure that provides the parameters to the CKM_CMS_SIG mechanism. It is defined as follows:

```
typedef struct CK_CMS_SIG_PARAMS {
  CK_OBJECT_HANDLE _certificateHandle;
  CK_MECHANISM_PTR pSigningMechanism;
  CK_MECHANISM_PTR pDigestMechanism;
} CK_CMS_SIG_PARAMS;
```
The fields of the structure have the following meanings:

- `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.

- `pSigningMechanism`: Mechanism to use when signing a constructed CMS `SignedAttributes` value. E.g. `CKM_SHA1_RSA_PKCS`.

- `pDigestMechanism`: Mechanism to use when digesting the data. Value shall be `NULL_PTR` when the digest mechanism to use follows from the `pSigningMechanism` parameter.

- `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.

- `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.

- `ulRequestedAttributesLen`: Length in bytes of the value pointed to by `pRequestedAttributes`.

- `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`.

- `ulRequiredAttributesLen`: Length in bytes of the value pointed to by `pRequiredAttributes`.

### 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
present through the parameters pRequiredAttributes. The signature is computed in accordance with the
parameter pSigningMechanism.

When this mechanism is used in successful calls to C_Sign or C_SignFinal, the pSignature return value
will point to a DER-encoded value of type SignerInfo. SignerInfo is defined in ASN.1 as follows (for a
complete definition of all fields and types, see RFC 2630):

\[
\text{SignerInfo ::= SEQUENCE { }
\text{  version CMSVersion, }
\text{  sid SignerIdentifier, }
\text{  digestAlgorithm DigestAlgorithmIdentifier, }
\text{  signedAtts [0] IMPLICIT SignedAttributes OPTIONAL, }
\text{  signatureAlgorithm SignatureAlgorithmIdentifier, }
\text{  signature SignatureValue, }
\text{  unsignedAtts [1] IMPLICIT UnsignedAttributes }
\text{ OPTIONAL } }
\]

The certificateHandle parameter, when set, helps the token populate the sid field of the SignerInfo value.
If certificateHandle is NULL_PTR the choice of a suitable certificate reference in the SignerInfo result
value is left to the token (the token could, e.g., interact with the user).

This mechanism shall not be used in calls to C_Verify or C_VerifyFinal (use the pSigningMechanism
mechanism instead).

For the pRequiredAttributes field, the token may have to interact with the user to find out whether to
accept a proposed value or not. The token should never accept any proposed attribute values without
some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings
and not direct interaction). If a user rejects proposed values, or the signature request as such, the value
CKR_FUNCTION_REJECTED shall be returned.

When possible, applications should use the CKM_CMS_SIG mechanism when generating CMS-
compatible signatures rather than lower-level mechanisms such as CKM_SHA1_RSA_PKCS. This is
especially true when the signatures are to be made on content that the token is able to present to a user.
Exceptions may include those cases where the token does not support a particular signing attribute. Note
however that the token may refuse usage of a particular signature key unless the content to be signed is
known (i.e. the CKM_CMS_SIG mechanism is used).

When a token does not have presentation capabilities, the PKCS #11-aware application may avoid
sending the whole message to the token by electing to use a suitable signature mechanism (e.g.
CKM_RSA_PKCS) as the pSigningMechanism value in the CK_CMS_SIG_PARAMS structure, and
digesting the message itself before passing it to the token.

PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to
provide messages to be signed by the token in a format possible for the token to present to the user.
Tokens that receive multipart MIME-messages for which only certain parts are possible to present may
fail the signature operation with a return value of CKR_DATA_INVALID, but may also choose to add a
signing attribute indicating which parts of the message were possible to present.

### 6.45 Blowfish

Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times.
The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex
initialization phase required before any encryption can take place, the actual encryption of data is very
efficient on large microprocessors. See [BLOWFISH] for details.

Table 201, Blowfish Mechanisms vs. Functions
Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_BLOWFISH_CBC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BLOWFISH_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

### 6.45.1 Definitions

This section defines the key type “CKK_BLOWFISH” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_BLOWFISH_KEY_GEN
- CKM_BLOWFISH_CBC
- CKM_BLOWFISH_CBC_PAD

### 6.45.2 BLOWFISH secret key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value the key can be any length up to 448 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BLOWFISH;
CK_UTF8CHAR label[] = "A blowfish secret key object";
CK_BYTE value[16] = {...};
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_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```
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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Determined by type of key being unwrapped or <code>CKA_VALUE_LEN</code></td>
</tr>
</tbody>
</table>

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.
Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Between 1 and block length block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>BLOWFISH</td>
<td>Any</td>
<td>Input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>BLOWFISH</td>
<td>Multiple of block size</td>
<td>Between 1 and block length block size bytes shorter than input length</td>
</tr>
</tbody>
</table>

### 6.46 Twofish

Twofish is a secret key block cipher. See [TWOFISH] for details.

#### 6.46.1 Definitions

This section defines the key type “CKK_TWOFISH” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_TWOFISH_KEY_GEN
- CKM_TWOFISH_CBC
- CKM_TWOFISH_CBC_PAD

#### 6.46.2 Twofish secret key objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Byte array</td>
<td>Key value 128-, 192-, or 256-bit key</td>
</tr>
<tr>
<td>CKA_VALUE_LEN&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_TWOFISH;
CK_UTF8CHAR label[] = "A twofish secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
```
6.46.3 Twofish key generation

The Twofish key generation mechanism, denoted `CKM_TWOFISH_KEY_GEN`, is a key generation mechanism Twofish.

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.

6.46.4 Twofish -CBC

Twofish-CBC, denoted `CKM_TWOFISH_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 16-byte initialization vector.

6.46.5 Twofish-CBC with PKCS padding

Twofish-CBC-PAD, denoted `CKM_TWOFISH_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 16-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.

6.47 CAMELLIA

Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.

Camellia is described e.g. in IETF RFC 3713.

Table 206, Camellia Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encryp / Decryp</th>
<th>Sign &amp; Verif</th>
<th>SR &amp; VR</th>
<th>Diges t</th>
<th>Gen - Key / Key Pair</th>
<th>Wrap &amp; Unwra p</th>
<th>Deriv e</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CAMELLIA_KEY_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_CAMELLIA_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 206, Camellia Mechanisms vs. Functions
### 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>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.

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

```c
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[] = {
```
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 and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of Camellia key sizes, in bytes.

### 6.47.5 Camellia-CBC

Camellia-CBC, denoted `CKM_CAMELLIA_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of Camellia key sizes, in bytes.

### 6.47.6 Camellia-CBC with PKCS padding

Camellia-CBC with PKCS padding, denoted `CKM_CAMELLIA_CBC_PAD`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia; cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].
It has a parameter, a 16-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.

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.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>between 1 and block size bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_CAMELLIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_CAMELLIA</td>
<td>multiple of block size</td>
<td>between 1 and block length bytes shorter than input length</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of Camellia key sizes, in bytes.

### 6.47.7 CAMELLIA with Counter mechanism parameters

- **CK_CAMELLIA_CTR_PARAMS; CK_CAMELLIA_CTR_PARAMS_PTR**

  `CK_CAMELLIA_CTR_PARAMS` is a structure that provides the parameters to the `CKM_CAMELLIA_CTR` mechanism. It is defined as follows:

  ```c
  typedef struct CK_CAMELLIA_CTR_PARAMS {
      CK_ULONG ulCounterBits;
      CK_BYTE cb[16];
  } CK_CAMELLIA_CTR_PARAMS;
  ```

  `ulCounterBits` specifies the number of bits in the counter block (`cb`) that shall be incremented. This number shall be such that `0 < ulCounterBits <= 128`. For any values outside this range the mechanism shall return `CKR_MECHANISM_PARAM_INVALID`.

  It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter bits are the least significant bits of the counter block (`cb`). They are a big-endian value usually starting with 1. The rest of `cb` is for the nonce, and maybe an optional IV.

  E.g. as defined in [RFC 3686]:

  ```c
  ulCounterBits specifies the number of bits in the counter block (`cb`) that shall be incremented. This
  number shall be such that `0 < ulCounterBits <= 128`. For any values outside this range the mechanism
  shall return `CKR_MECHANISM_PARAM_INVALID`.
  ```
Standards Track
pkcs11
specify the supported range of C
For this mechanism, the
Table
Constraints on key types and the length of data are summarized in the following table:
MAC mechanism. C
C
C
6.47.9 Camellia-MAC
Camellia-MAC, denoted by CKM_CAMELLIA_MAC, is a special case of the general-length Camellia-
MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
length.
It does not have a parameter.
Constraints on key types and the length of data are summarized in the following table:
Table 212, Camellia-MAC: Key and Data Length
Function | Key type  | Data length | Signature length  
---------|-----------|-------------|-------------------
C_Sign   | CKK_CAMELLIA | any         | ½ block size (8 bytes) 
C_Verify | CKK_CAMELLIA | any         | ½ block size (8 bytes) 

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Camellia key sizes, in bytes.

6.47.8 General-length Camellia-MAC
General-length Camellia-MAC, denoted CKM_CAMELLIA_MAC_GENERAL, is a mechanism for single-
and multiple-part signatures and verification, based on Camellia and data authentication as defined in.[CAMELLIA]
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 Camellia cipher block produced in the MACing process.
Constraints on key types and the length of data are summarized in the following table:
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 

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Camellia key sizes, in bytes.

This construction permits each packet to consist of up to $2^{32} - 1$ blocks = 4,294,967,295 blocks = 68,719,476,720 octets.

CK_CAMELLIA_CTR_PARAMS_PTR is a pointer to a CK_CAMELLIA_CTR_PARAMS.

6.47.8 General-length Camellia-MAC
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

```c
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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_CAMELLIA_ECB_ENCRYPT_DATA</td>
<td></td>
</tr>
<tr>
<td>CKM_CAMELLIA_CBC_ENCRYPT_DATA</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_ARIA_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_ARIA_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_ARIA_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_ARIA_CBC_PAD</td>
<td>✓</td>
</tr>
</tbody>
</table>
### 6.49.1 Definitions

This section defines the key type “CKK_ARIA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

#### Mechanisms:
- CKM_ARIA_KEY_GEN
- CKM_ARIA_ECB
- CKM_ARIA_CBC
- CKM_ARIA_MAC
- CKM_ARIA_MAC_GENERAL
- CKM_ARIA_CBC_PAD

### 6.49.2 Aria secret key objects

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

#### Table 215, ARIA Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ARIA;
CK_UTF8CHAR label[] = "An ARIA 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_ENCRYPT, &true, sizeof(true)},
...}
```
{CKA_VALUE, value, sizeof(value)}

6.49.3 ARIA key generation
The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism for Aria.
It does not have a parameter.
The mechanism generates ARIA 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 ARIA 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 ARIA key sizes, in bytes.

6.49.4 ARIA-ECB
ARIA-ECB, denoted CKM_ARIA_ECB, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Aria 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>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of block size</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of ARIA key sizes, in bytes.

6.49.5 ARIA-CBC
ARIA-CBC, denoted CKM_ARIA_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.
It has a parameter, a 16-byte initialization vector.

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 217, ARIA-CBC: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C.WrapKey</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>input length rounded up to multiple of the block size</td>
<td></td>
</tr>
<tr>
<td>C.UnwrapKey</td>
<td>CKK_ARIA</td>
<td>multiple of block size</td>
<td>determined by type of key being unwrapped or CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of Aria key sizes, in bytes.

6.49.6 ARIA-CBC with PKCS padding

ARIA-CBC with PKCS padding, denoted CKM_ARIA_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].

It has a parameter, a 16-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.

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.

Constraints on key types and the length of data are summarized in the following table:

Table 218, ARIA-CBC with PKCS Padding: Key and Data Length
For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of ARIA key sizes, in bytes.

### 6.49.7 General-length ARIA-MAC

General-length ARIA-MAC, denoted `CKM_ARIA_MAC_GENERAL`, is a mechanism for single- and multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

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 ARIA cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>1-block size, as specified in parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of ARIA key sizes, in bytes.

### 6.49.8 ARIA-MAC

ARIA-MAC, denoted by `CKM_ARIA_MAC`, is a special case of the general-length ARIA-MAC mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_ARIA</td>
<td>any</td>
<td>½ block size (8 bytes)</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of ARIA key sizes, in bytes.

### 6.50 Key derivation by data encryption - ARIA

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.
6.50.1 Definitions

Mechanisms:

CKM_ARIA_ECB_ENCRYPT_DATA
CKM_ARIA_CBC_ENCRYPT_DATA

typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE      iv[16];
    CK_BYTE_PTR  pData;
    CK_ULONG     length;
} CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;

typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;

6.50.2 Mechanism Parameters

Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

Table 221, Mechanism Parameters for Aria-based key derivation

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_ARIA_ECB_ENCRYPT_DATA</td>
<td>Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</td>
</tr>
<tr>
<td>CKM_ARIA_CBC_ENCRYPT_DATA</td>
<td>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.</td>
</tr>
</tbody>
</table>

6.51 SEED

SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It has a 128-bit key size and a 128-bit block size.

Its specification has been published as Internet [RFC 4269].

RFCs have been published defining the use of SEED in


TLS cipher suites that use SEED include:

CipherSuite TLS_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x96};
CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x97};
CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x98};
CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x99};
CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x9A};
CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00, 0x9B};

As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as in a MAC algorithm such as HMAC.
OIDs have been published for all these uses. A list may be seen at
http://www.alvestrand.no/objectid/1.2.410.200004.1.html

Table 222, SEED Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encryp &amp; Decryp</th>
<th>Sign &amp; Verif</th>
<th>SR &amp; VR</th>
<th>Diges</th>
<th>Gen &amp; Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SEED_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SEED_ECB</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC_PAD</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_MAC_GENERAL</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_MAC</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_ECB_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SEED_CBC_ENCRYPT_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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:

- CKM_SEED_KEY_GEN
- CKM_SEED_ECB
- CKM_SEED_CBC
- CKM_SEED_MAC
- CKM_SEED_MAC_GENERAL
- CKM_SEED_CBC_PAD

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:
Table 223, SEED Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key value (always 16 bytes long)</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes.

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

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SEED;
CK_UTF8CHAR label[] = "A SEED 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_VALUE, value, sizeof(value)}
};
```

6.51.3 SEED key generation

The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism for SEED.

It does not have a parameter.

The mechanism generates SEED keys.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the SEED 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.

6.51.4 SEED-ECB

SEED-ECB, denoted CKM_SEED_ECB, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

It does not have a parameter.

6.51.5 SEED-CBC

SEED-CBC, denoted CKM_SEED_CBC, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

6.51.6 SEED-CBC with PKCS padding

SEED-CBC with PKCS padding, denoted CKM_SEED_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].

It has a parameter, a 16-byte initialization vector.
6.51.7 General-length SEED-MAC

General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on SEED and data authentication.

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 cipher block produced in the MACing process.

6.51.8 SEED-MAC

SEED-MAC, denoted by **CKM_SEED_MAC**, is a special case of the general-length SEED-MAC mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

6.52 Key derivation by data encryption - SEED

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.

6.52.1 Definitions

Mechanisms:

- **CKM_SEED_ECB_ENCRYPT_DATA**
- **CKM_SEED_CBC_ENCRYPT_DATA**

```c
typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE iv[16];
    CK_BYTE_PTR pData;
    CK_ULONG length;
} CK_SEED_CBC_ENCRYPT_DATA_PARAMS;
```

6.52.2 Mechanism Parameters

<table>
<thead>
<tr>
<th><strong>CKM_SEED_ECB_ENCRYPT_DATA</strong></th>
<th>Uses <strong>CK_KEY_DERIVATION_STRING_DATA</strong> structure. Parameter is the data to be encrypted and must be a multiple of 16 long.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CKM_SEED_CBC_ENCRYPT_DATA</strong></td>
<td>Uses <strong>CK_SEED_CBC_ENCRYPT_DATA_PARAMS</strong>. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.</td>
</tr>
</tbody>
</table>
6.53 OTP

6.53.1 Usage overview
OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases can be categorized according to the type of sought functionality.

6.53.2 Case 1: Generation of OTP values

Figure 2: Retrieving OTP values through C_Sign
Figure 2 shows an integration of PKCS #11 into an application that needs to authenticate users holding OTP tokens. In this particular example, a connected hardware token is used, but a software token is equally possible. The application invokes C_Sign to retrieve the OTP value from the token. In the example, the application then passes the retrieved OTP value to a client API that sends it via the network to an authentication server. The client API may implement a standard authentication protocol such as RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's ACE/Agent® software.
6.53.3 Case 2: Verification of provided OTP values

Figure 3: Server-side verification of OTP values

Figure 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. In this case, a server application invokes `C_Verify` with the received OTP value as the signature value to be verified.

6.53.4 Case 3: Generation of OTP keys

```plaintext
Client Application
   ↙ C_GenerateKey()

PKCS #11 Library
   ↙

Internal Token API
   ↙

Token (or software version thereof)
```
Figure 4 shows an integration of PKCS #11 into an application that generates OTP keys. The application invokes \texttt{C\_GenerateKey} to generate an OTP key of a particular type on the token. The key may subsequently be used as a basis to generate OTP values.

### 6.53.5 OTP objects

#### 6.53.5.1 Key objects

OTP key objects (object class \texttt{CKO\_OTP\_KEY}) hold secret keys used by OTP tokens. The following table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys, all of which are inherited by this class:

\textit{Table 225: Common OTP key attributes}
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_OTP_FORMAT</td>
<td>CK_ULONG</td>
<td>Format of OTP values produced with this key:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_FORMAT_DECIMAL = Decimal (default) (UTF8-encoded)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_FORMAT_HEXADECIMAL = Hexadecimal (UTF8-encoded)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_FORMAT_ALPHANUMERIC = Alphanumeric (UTF8-encoded)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_FORMAT_BINARY = Only binary values.</td>
</tr>
<tr>
<td>CKA_OTP_LENGTH</td>
<td>CK_ULONG</td>
<td>Default length of OTP values (in the CKA_OTP_FORMAT) produced with this key.</td>
</tr>
<tr>
<td>CKA_OTP_USER_FRIENDLY_MODE</td>
<td>CK_BBOOL</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_OTP_CHALLENGE_REQUIREMENT</td>
<td>CK_ULONG</td>
<td>Parameter requirements when generating or verifying OTP values with this key:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_MANDATORY = A challenge must be supplied.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_OPTIONAL = A challenge may be supplied but need not be.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_IGNORED = A challenge, if supplied, will be ignored.</td>
</tr>
<tr>
<td>CKA_OTP_TIME_REQUIREMENT</td>
<td>CK_ULONG</td>
<td>Parameter requirements when generating or verifying OTP values with this key:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_MANDATORY = A time value must be supplied.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_OPTIONAL = A time value may be supplied but need not be.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CK_OTP_PARAM_IGNORED = A time value, if supplied, will be ignored.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| CKA_OTP_COUNTER_REQUIREMENT | 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 | 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. ulValueLen = 0). |
| CKA_OTP_TIME | RFC 2279 string | Value of the associated internal UTC time in the form YYYYMMDDhhmss. Default value is empty (i.e. ulValueLen = 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. ulValueLen = 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. ulValueLen = 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. ulValueLen = 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. ulValueLen = 0). |
| CKA_VALUE | Byte array | Value of the key. |
| CKA_VALUE_LEN | CK_ULONG | Length in bytes of key value. |

13039 Refer to Table 11 for footnotes
Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-PKCS #11 application should therefore always consult the state of the CKA_OTP_PIN_REQUIREMENT attribute before each call to C_SignInit, as the value of this attribute may change dynamically.

For OTP tokens with multiple keys, the keys may be enumerated using C_FindObjects. The CKA_OTP_SERVICE_IDENTIFIER and/or the CKA_OTP_SERVICE_LOGO attribute may be used to distinguish between keys. The actual choice of key for a particular operation is however application-specific and beyond the scope of this document.

For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

### 6.53.6 OTP-related notifications

This document extends the set of defined notifications as follows:

- **CKN_OTP_CHANGED** Cryptoki is informing the application that the OTP for a key on a connected token just changed. This notification is particularly useful when applications wish to display the current OTP value for time-based mechanisms.

### 6.53.7 OTP mechanisms

The following table shows, for the OTP mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

**Table 226: OTP mechanisms vs. applicable functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SECURID KEY_GEN</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_SECURID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_HOTP KEY_GEN</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_HOTP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ACTI KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_ACTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

The remainder of this section will present in detail the OTP mechanisms and the parameters that are supplied to them.

### 6.53.7.1 OTP mechanism parameters

#### CK_OTP_PARAM_TYPE

**CK_OTP_PARAM_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

```c
typedef CK_ULONG CK_OTP_PARAM_TYPE;
```

The following **CK_OTP_PARAM_TYPE** types are defined:

**Table 227, OTP parameter types**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CK_OTP_PIN</td>
<td>RFC 2279 string</td>
<td>A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_CHALLENGE</td>
<td>Byte array</td>
<td>Challenge to use when computing or verifying challenge-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_TIME</td>
<td>RFC 2279 string</td>
<td>UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_COUNTER</td>
<td>Byte array</td>
<td>Counter value to use when computing or verifying counter-based OTP values.</td>
</tr>
<tr>
<td>CK_OTP_FLAGS</td>
<td>CK_FLAGS</td>
<td>Bit flags indicating the characteristics of the sought OTP as defined below.</td>
</tr>
<tr>
<td>CK_OTP_OUTPUT_LENGTH</td>
<td>CK_ULONG</td>
<td>Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.</td>
</tr>
<tr>
<td>CK_OTP_OUTPUT_FORMAT</td>
<td>CK_ULONG</td>
<td>Returned OTP format (allowed values are the same as for CKA_OTP_FORMAT). This parameter is only intended for C_Sign 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.</td>
</tr>
<tr>
<td>CK_OTP_VALUE</td>
<td>Byte array</td>
<td>An actual OTP value. This parameter type is intended for C_Sign output, see paragraphs below.</td>
</tr>
</tbody>
</table>

The following table defines the possible values for the CK_OTP_FLAGS type:

<table>
<thead>
<tr>
<th>Bit flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 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.

9 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.
<table>
<thead>
<tr>
<th>Bit flag</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKF_EXCLUDE_TIME</td>
<td>0x00000002</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_COUNTER</td>
<td>0x00000004</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_CHALLENGE</td>
<td>0x00000008</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_EXCLUDE_PIN</td>
<td>0x00000010</td>
<td>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.</td>
</tr>
<tr>
<td>CKF_USER_FRIENDLY_OTP</td>
<td>0x00000020</td>
<td>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.</td>
</tr>
</tbody>
</table>

Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be value in setting the CKF_USER_FRIENDLY_OTP flag (assuming CKA_OTP_USER_FRIENDLY_MODE is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF_USER_FRIENDLY_OTP flag.

**CK_OTP_PARAM; CK_OTP_PARAM_PTR**

CK_OTP_PARAM is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:
typedef struct CK_OTP_PARAM {
    CK_OTP_PARAM_TYPE type;
    CK_VOID_PTR pValue;
    CK_ULONG ulValueLen;
} CK_OTP_PARAM;

The fields of the structure have the following meanings:

- type: the parameter type
- pValue: pointer to the value of the parameter
- ulValueLen: length in bytes of the value

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).

CK_OTP_PARAM_PTR is a pointer to a CK_OTP_PARAM.

**CK_OTP_PARAMS; CK_OTP_PARAMS_PTR**

CK_OTP_PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

typedef struct CK_OTP_PARAMS {
    CK_OTP_PARAM_PTR pParams;
    CK_ULONG ulCount;
} CK_OTP_PARAMS;

The fields of the structure have the following meanings:

- pParams: pointer to an array of OTP parameters
- ulCount: the number of parameters in the array

CK_OTP_PARAMS_PTR is a pointer to a CK_OTP_PARAMS.

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 CKA_OTP_\_X\_REQUIREMENT key attributes for the identified key, where X is PIN, CHALLENGE, TIME, or COUNTER.

For example, if CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_MANDATORY, then the CK_OTP_TIME parameter shall be present. If CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_OPTIONAL, then a CK_OTP_TIME parameter may be present. If it is not present, then the library may collect it (during the C_Sign call). If CKA_OTP_TIME_REQUIREMENT = CK_OTP_PARAM_IGNORED, then a provided CK_OTP_TIME parameter will always be ignored.

Additionally, a provided CK_OTP_TIME parameter will always be ignored if CKF_EXCLUDE_TIME is set in a CK_OTP_FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (CKR_MECHANISM_PARAM_INVALID) to set the CKF_EXCLUDE_TIME flag when the CKA_OTP_TIME_REQUIREMENT attribute is CK_OTP_PARAM_MANDATORY.

The above discussion holds for all CKA_OTP_\_X\_REQUIREMENT attributes (i.e., CKA_OTP_PIN_REQUIREMENT, CKA_OTP_CHALLENGE_REQUIREMENT, CKA_OTP_COUNTER_REQUIREMENT, CKA_OTP_TIME_REQUIREMENT). A library may set a particular CKA_OTP_\_X\_REQUIREMENT attribute to CK_OTP_PARAM_OPTIONAL even if it is required 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.
In addition, applications may use the CK_OTP_FLAGS, the CK_OTP_OUTPUT_FORMAT and the CKA_OTP_LENGTH parameters to set additional parameters.

**CK_OTP_SIGNATURE_INFO, CK_OTP_SIGNATURE_INFO_PTR**

**CK_OTP_SIGNATURE_INFO** is a structure that is returned by all OTP mechanisms in successful calls to **C_Sign** (C_SignFinal). The structure informs applications of actual parameter values used in particular OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key belongs to the class CKO_OTP_KEY and is defined as follows:

```c
typedef struct CK_OTP_SIGNATURE_INFO {
    CK_OTP_PARAM_PTR pParams;
    CK_ULONG ulCount;
} CK_OTP_SIGNATURE_INFO;
```

The fields of the structure have the following meanings:

- `pParams`: pointer to an array of OTP parameter values
- `ulCount`: the number of parameters in the array

After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the `pSignature` parameter will be set to point to a **CK_OTP_SIGNATURE_INFO** structure. One of the parameters in this structure will be the OTP value itself, identified with the **CK_OTP_VALUE** tag. Other parameters may be present for informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP validations, authentication protocols may permit authenticating parties to send some or all of these parameters in addition to OTP values themselves. Applications should therefore check for their presence in returned **CK_OTP_SIGNATURE_INFO** values whenever such circumstances apply.

Since **C_Sign** and **C_SignFinal** follows the convention described in Section 5.2 on producing output, a call to **C_Sign** (or **C_SignFinal**) with `pSignature` set to NULL_PTR will return (in the `pulSignatureLen` parameter) the required number of bytes to hold the **CK_OTP_SIGNATURE_INFO** structure as well as all the data in all its **CK_OTP_PARAM** components. If an application allocates a memory block based on this information, it shall therefore not subsequently de-allocate components of such a received value but rather de-allocate the complete **CK_OTP_PARAMS** structure itself. A Cryptoki library that is called with a non-NULL `pSignature` pointer will assume that it points to a contiguous memory block of the size indicated by the `pulSignatureLen` parameter.

When verifying an OTP value using an OTP mechanism, `pSignature` shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure returned by a call to **C_Sign**. The **CK_OTP_PARAM** value supplied in the **C_VerifyInit** call sets the values to use in the verification operation.

**CK_OTP_SIGNATURE_INFO_PTR** points to a **CK_OTP_SIGNATURE_INFO**.

### 6.53.8 RSA SecurID

#### 6.53.8.1 RSA SecurID secret key objects

RSA SecurID secret key objects (object class CKO_OTP_KEY, key type CKK_SECURID) hold RSA SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_OTP_TIME_INTERVAL¹</td>
<td>CK_ULONG</td>
<td>Interval between OTP values produced with this key, in seconds. Default is 60.</td>
</tr>
</tbody>
</table>

¹ Refer to Table 11 for footnotes

The following is a sample template for creating an RSA SecurID secret key object:
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_SECURID;
CK_DATE endDate = {...};
CK_UTF8CHAR label[] = "RSA SecurID secret key object";
CK_BYTE keyId[] = {...};
CK_ulong outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ulong outputLength = 6;
CK_ulong needPIN = CK_OTP_PARAM_MANDATORY;
CK_ulong timeInterval = 60;
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label) - 1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
    {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
    {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
    {CKA_OTP_TIME_INTERVAL, &timeInterval,
        sizeof(timeInterval)},
    {CKA_VALUE, value, sizeof(value)}
};

6.53.8.2 RSA SecurID key generation

The RSA SecurID key generation mechanism, denoted CKM_SECURID_KEY_GEN, is a key generation
mechanism for the RSA SecurID algorithm.

It does not have a parameter.

The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the
template for the key.

The mechanism contributes at least the CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE_LEN, and
CKA_VALUE attributes to the new key. Other attributes supported by the RSA SecurID key type 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 SecurID key sizes, in bytes.

6.53.8.3 SecurID OTP generation and validation

CKM_SECURID is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a CK_OTP_PARAMS structure as a parameter.

When signing or verifying using the CKM_SECURID mechanism, pData shall be set to NULL_PTR and
ulDataLen shall be set to 0.
6.53.8.4 Return values

Support for the CKM_SECRUID mechanism extends the set of return values for C_Verify with the following values:

- CKR_NEW_PIN_MODE: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.
- CKR_NEXT_OTP: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

6.53.9 OATH HOTP

6.53.9.1 OATH HOTP secret key objects

HOTP secret key objects (object class CKO_OTP_KEY, key type CKK_HOTP) hold generic secret keys and associated counter values.

The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.

For HOTP keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK_OTP_COUNTER** value in a **CK_OTP_PARAM** structure.

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

```c
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_HOTP;
CK_UTF8CHAR label[] = "HOTP secret key object";
CK_BYTE keyId[] = {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
    {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
    {CKA_OTP_COUNTER, counterValue, sizeof(counterValue)},
    {CKA_VALUE, value, sizeof(value)}
};
```
6.53.9.2 HOTP key generation

The HOTP key generation mechanism, denoted \texttt{CKM\_HOTP\_KEY\_GEN}, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, \texttt{CKA\_OTP\_COUNTER}, \texttt{CKA\_VALUE} and \texttt{CKA\_VALUE\_LEN} attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of HOTP key sizes, in bytes.

6.53.9.3 HOTP OTP generation and validation

\texttt{CKM\_HOTP} is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

As for the \texttt{CKM\_SECURID} mechanism, when signing or verifying using the \texttt{CKM\_HOTP} mechanism, \texttt{pData} shall be set to \texttt{NULL\_PTR} and \texttt{ulDataLen} shall be set to 0.

For verify operations, the counter value \texttt{CK\_OTP\_COUNTER} must be provided as a \texttt{CK\_OTP\_PARAM} parameter to \texttt{C\_VerifyInit}. When verifying an OTP value using the \texttt{CKM\_HOTP} mechanism, \texttt{pSignature} shall be set to the OTP value itself, e.g. the value of the \texttt{CK\_OTP\_VALUE} component of a \texttt{CK\_OTP\_PARAM} structure in the case of an earlier call to \texttt{C\_Sign}.

6.53.10 ActivIdentity ACTI

6.53.10.1 ACTI secret key objects

ACTI secret key objects (object class \texttt{CKO\_OTP\_KEY}, key type \texttt{CKK\_ACTI}) hold ActivIdentity ACTI secret keys.

For ACTI keys, the \texttt{CKA\_OTP\_COUNTER} value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for the \texttt{CK\_OTP\_COUNTER} value in the \texttt{CK\_OTP\_PARAM} structure.

The \texttt{CKA\_OTP\_COUNTER} value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its \texttt{CKA\_SENSITIVE} attribute set to \texttt{CK\_TRUE} or its \texttt{CKA\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}.

The \texttt{CKA\_OTP\_TIME} value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its \texttt{CKA\_SENSITIVE} attribute set to \texttt{CK\_TRUE} or its \texttt{CKA\_EXTRACTABLE} attribute set to \texttt{CK\_FALSE}.

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

\begin{verbatim}
CK\_OBJECT\_CLASS class = CKO\_OTP\_KEY;
CK\_KEY\_TYPE keyType = CKK\_ACTI;
CK\_UTF8\_CHAR label[] = "ACTI secret key object";
CK\_BYTE keyId[] = {...};
CK\_ULONG outputFormat = CK\_OTP\_FORMAT\_DECIMAL;
CK\_ULONG outputLength = 6;
CK\_DATE endDate = {...};
CK\_BYTE counterValue[8] = {0};
\end{verbatim}
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
    {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
    {CKA_OTP_COUNTER, counterValue, sizeof(counterValue)},
    {CKA_VALUE, value, sizeof(value)}
};

6.53.10.2 ACTI key generation

The ACTI key generation mechanism, denoted CKM_ACTI_KEY_GEN, is a key generation mechanism for the ACTI algorithm. It does not have a parameter. The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the key. The mechanism contributes at least the CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE and CKA_VALUE_LEN attributes to the new key. Other attributes supported by the ACTI key type 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 ACTI key sizes, in bytes.

6.53.10.3 ACTI OTP generation and validation

CKM_ACTI is the mechanism for the retrieval and verification of ACTI OTP values. The mechanism takes a pointer to a CK_OTP_PARAMS structure as a parameter. When signing or verifying using the CKM_ACTI mechanism, pData shall be set to NULL_PTR and ulDataLen shall be set to 0. When verifying an OTP value using the CKM_ACTI mechanism, pSignature shall be set to the OTP value itself, e.g. the value of the CK_OTP_VALUE component of a CK_OTP_PARAM structure in the case of an earlier call to C_Sign.
6.54 CT-KIP

6.54.1 Principles of Operation

Figure 5: PKCS #11 and CT-KIP integration

Figure 5 shows an integration of PKCS #11 into an application that generates cryptographic keys through the use of CT-KIP. The application invokes `C_DeriveKey` to derive a key of a particular type on the token. The key may subsequently be used as a basis to e.g., generate one-time password values. The application communicates with a CT-KIP server that participates in the key derivation and stores a copy of the key in its database. The key is transferred to the server in wrapped form, after a call to `C_WrapKey`. The server authenticates itself to the client and the client verifies the authentication by calls to `C_Verify`.

6.54.2 Mechanisms

The following table shows, for the mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 230: CT-KIP Mechanisms vs. applicable functions
The remainder of this section will present in detail the mechanisms and the parameters that are supplied to them.

### 6.54.3 Definitions

#### Mechanisms:
- CKM_KIP_DERIVE
- CKM_KIP_WRAP
- CKM_KIP_MAC

#### 6.54.4 CT-KIP Mechanism parameters

The CT-KIP parameters are defined as follows:

```c
typedef struct CK_KIP_PARAMS {
    CK_MECHANISM_PTR  pMechanism;
    CK_OBJECT_HANDLE  hKey;
    CK_BYTE_PTR       pSeed;
    CK_ULONG          ulSeedLen;
} CK_KIP_PARAMS;
```

The fields of the structure have the following meanings:

- `pMechanism`: pointer to the underlying cryptographic mechanism (e.g. AES, SHA-256)
- `hKey`: handle to a key that will contribute to the entropy of the derived key (CKM_KIP_DERIVE) or will be used in the MAC operation (CKM_KIP_MAC)
- `pSeed`: pointer to an input seed
- `ulSeedLen`: length in bytes of the input seed

**CK_KIP_PARAMS_PTR** is a pointer to a **CK_KIP_PARAMS** structure.

### 6.54.5 CT-KIP key derivation

The CT-KIP key derivation mechanism, denoted **CKM_KIP_DERIVE**, is a key derivation mechanism that is capable of generating secret keys of potentially any type, subject to token limitations.

It takes a parameter of type **CK_KIP_PARAMS** which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. In particular, when the `hKey` parameter is a handle...
13392 to an existing key, that key will be used in the key derivation in addition to the \textit{hBaseKey} of \textit{C\_DeriveKey}.

13393 The \textit{pSeed} parameter may be used to seed the key derivation operation.

13394 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the
13395 template for the key.

13396 The mechanism contributes the \textbf{CKA\_CLASS} and \textbf{CKA\_VALUE} attributes to the new key. Other
13397 attributes supported by the key type may be specified in the template for the key, or else will be assigned
13398 default initial values. Since the mechanism is generic, the \textbf{CKA\_KEY\_TYPE} attribute should be set in the
13399 template, if the key is to be used with a particular mechanism.

13400 \textbf{6.54.6 CT-KIP key wrap and key unwrap}

13401 The CT-KIP key wrap and unwrap mechanism, denoted \textbf{CKM\_KIP\_WRAP}, is a key wrap mechanism that
13402 is capable of wrapping and unwrapping generic secret keys.

13403 It takes a parameter of type \textbf{CK\_KIP\_PARAMS}, which allows for the passing of the desired underlying
13404 cryptographic mechanism as well as some other data. It does not make use of the \textit{hKey} parameter of
13405 \textbf{CK\_KIP\_PARAMS}.

13406 \textbf{6.54.7 CT-KIP signature generation}

13407 The CT-KIP signature (MAC) mechanism, denoted \textbf{CKM\_KIP\_MAC}, is a mechanism used to produce a
13408 message authentication code of arbitrary length. The keys it uses are secret keys.

13409 It takes a parameter of type \textbf{CK\_KIP\_PARAMS}, which allows for the passing of the desired underlying
13410 cryptographic mechanism as well as some other data. The mechanism does not make use of the \textit{pSeed}
13411 and the \textit{ulSeedLen} parameters of \textbf{CT\_KIP\_PARAMS}.

13412 This mechanism produces a MAC of the length specified by \textit{pulSignatureLen} parameter in calls to
13413 \textbf{C\_Sign}.

13414 If a call to \textbf{C\_Sign} with this mechanism fails, then no output will be generated.

13415 \textbf{6.55 GOST 28147-89}

13416 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

13417 \textit{Table 231, GOST 28147-89 Mechanisms vs. Functions}

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_GOST28147_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GOST28147_ECB</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147_MAC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_GOST28147_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

13419

\textbf{6.55.1 Definitions}

13420 This section defines the key type “CKK\_GOST28147” for type \textbf{CK\_KEY\_TYPE} as used in the
13421 \textbf{CKA\_KEY\_TYPE} attribute of key objects and domain parameter objects.
Mechanisms:

- CKM_GOST28147_KEY_GEN
- CKM_GOST28147_ECB
- CKM_GOST28147
- CKM_GOST28147_MAC
- CKM_GOST28147_KEY_WRAP

6.55.2 GOST 28147-89 secret key objects

GOST 28147-89 secret key objects (object class CKO_SECRET_KEY, key type CKK_GOST28147) hold GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE₂₃₄₆₇</td>
<td>Byte array</td>
<td>32 bytes in little endian order</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS¹³⁵</td>
<td>Byte array</td>
<td>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</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The following is a sample template for creating a GOST 28147-89 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = “A GOST 28147-89 secret key object”;
CK_BYTE value[32] = {...};
CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
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_ENCRYPT, &true, sizeof(true)},
    {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},
    {CKAVALUE, value, sizeof(value)}
};
```

6.55.3 GOST 28147-89 domain parameter objects

GOST 28147-89 domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_GOST28147) hold GOST 28147-89 domain parameters.

The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the common attributes defined for this object class:
Table 233, GOST 28147-89 Domain Parameter Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE†</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type Gost28147-89-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID†</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

† Refer to Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST 28147-89 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic parameters object";
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[] = {
    0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,
    0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0xb9, 0x02, 0x98, 0x75, 0x61, 0x3b,
    0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0xc0, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,
    0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0x8b, 0xe8, 0xb4, 0x2c, 0x77, 0xe7,
    0xcd, 0x46, 0x6a, 0xa0, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0x99, 0xa4,
    0x0f, 0x02, 0x21, 0x65, 0x8b, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,
    0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,
    0x00, 0x05, 0x00
};
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_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.55.4 GOST 28147-89 key generation

The GOST 28147-89 key generation mechanism, denoted CKM_GOST28147_KEY_GEN, is a key generation mechanism for GOST 28147-89.

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 GOST 28147-89 key type may be specified for objects of object class CKO_SECRET_KEY.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO are not used.
6.55.5 GOST 28147-89-ECB

GOST 28147-89-ECB, denoted CKM_GOST28147_ECB, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 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 (C_WrapKey), 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 so that the resulting length is a multiple of the block size.

For unwrapping (C_UnwrapKey), the mechanism decrypts the wrapped key, and contributes the result as the CKA_VALUE attribute of the new key.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_GOST28147</td>
<td>Any</td>
<td>Input length rounded up to multiple of block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Determined by type of key being unwrapped</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

6.55.6 GOST 28147-89 encryption mode except ECB

GOST 28147-89 encryption mode except ECB, denoted CKM_GOST28147, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2. Encryption’s parameters are specified in object identifier of attribute CKA_GOST28147_PARAMS.

It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero initialization vector is used.

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 (C_WrapKey), the mechanism encrypts the value of the CKA_VALUE attribute of the key that is wrapped.

For unwrapping (C_UnwrapKey), the mechanism decrypts the wrapped key, and contributes the result as the CKA_VALUE attribute of the new key.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Same as input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>CKK_GOST28147</td>
<td>Any</td>
<td>Input length rounded up to multiple of block size</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_GOST28147</td>
<td>Multiple of block size</td>
<td>Determined by type of key being unwrapped</td>
</tr>
</tbody>
</table>
For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure are not used.

**6.55.7 GOST 28147-89-MAC**

GOST 28147-89-MAC, denoted \texttt{CKM\_GOST28147\_MAC}, is a mechanism for data integrity and authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3. MACing parameters are specified in object identifier of attribute \texttt{CKA\_GOST28147\_PARAMS}.

The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block produced in the MACing process.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

Constraints on key types and the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_GOST28147</td>
<td>Any</td>
<td>4 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_GOST28147</td>
<td>Any</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the \textit{ulMinKeySize} and \textit{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure are not used.

**6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89**

GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by \texttt{CKM\_GOST28147\_KEY\_WRAP}, is a mechanism for key wrapping; and key unwrapping, based on GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89.

For wrapping (\texttt{C\_WrapKey}), the mechanism first computes MAC from the value of the \texttt{CKA\_VALUE} attribute of the key that is wrapped and then encrypts in ECB mode the value of the \texttt{CKA\_VALUE} attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

For unwrapping (\texttt{C\_UnwrapKey}), the mechanism first decrypts in ECB mode the 32 bytes of the key that was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is disallowed. The mechanism contributes the result as the \texttt{CKA\_VALUE} attribute of the unwrapped key.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

Constraints on key types and the length of data are summarized in the following table:
Table 237, GOST 28147-89 keys as KEK: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_WrapKey</td>
<td>CKK_GOST28147</td>
<td>32 bytes</td>
<td>36 bytes</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>CKK_GOST28147</td>
<td>32 bytes</td>
<td>36 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

6.56 GOST R 34.11-94

GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit message digest defined in [GOST R 34.11-94].

Table 238, GOST R 34.11-94 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrpt</td>
</tr>
<tr>
<td>CKM_GOSTR3411</td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3411_HMAC</td>
<td>✓</td>
</tr>
</tbody>
</table>

6.56.1 Definitions

This section defines the key type “CKK_GOSTR3411” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of domain parameter objects.

Mechanisms:

CKM_GOSTR3411
CKM_GOSTR3411_HMAC

6.56.2 GOST R 34.11-94 domain parameter objects

GOST R 34.11-94 domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_GOSTR3411) hold GOST R 34.11-94 domain parameters.

The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 239, GOST R 34.11-94 Domain Parameter Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type GostR3411-94-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes
For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3411;
CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic parameters object";
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
                 0x1e, 0x00};
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_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

**6.56.3 GOST R 34.11-94 digest**

GOST R 34.11-94 digest, denoted **CKM_GOSTR3411**, is a mechanism for message digesting based on GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

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.

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>
<thead>
<tr>
<th>Function</th>
<th>Input length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>Any</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure 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*

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK GENERIC_SECRET or CKK_GOST28147</td>
<td>Any</td>
<td>32 byte</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK GENERIC_SECRET or CKK_GOST28147</td>
<td>Any</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

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].

*Table 242, GOST R34.10-2001 Mechanisms vs. Functions*

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_GOSTR3410_KEY_PAIR_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GOSTR3410</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3410_WITH_GOSTR341</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_GOSTR3410_KEY_WRAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

1 Single-part operations only

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:
### 6.57.2 GOST R 34.10-2001 public key objects

GOST R 34.10-2001 public key objects (object class `CKO_PUBLIC_KEY`, key type `CKK_GOSTR3410`) hold GOST R 34.10-2001 public keys.

The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>64 bytes for public key; 32 bytes for each coordinates X and Y of Elliptic Curve point P(X, Y) in little endian order</td>
</tr>
<tr>
<td>CKA_GOSTR3410_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
<tr>
<td>CKA_GOSTR3411_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS</td>
<td>Byte array</td>
<td>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</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

The following is a sample template for creating an GOST R 34.10-2001 public key object:

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";
CK_BYTE gostR3410params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] =
```
{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};

CK_BYTE gost28147params_oid[] =
{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
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_GOSTR3410_PARAMS, gostR3410params_oid,
        sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
        sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid,
        sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};

6.57.3 GOST R 34.10-2001 private key objects

GOST R 34.10-2001 private key objects (object class CKO_PRIVATE_KEY, key type CKK_GOSTR3410) hold GOST R 34.10-2001 private keys.

The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE^{1,4,6,7}</td>
<td>Byte array</td>
<td>32 bytes for private key in little endian order</td>
</tr>
<tr>
<td>CKA_GOSTR3410_PARAMS^{1,4,6}</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID</td>
</tr>
<tr>
<td>CKA_GOSTR3411_PARAMS^{1,4,6,8}</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID</td>
</tr>
<tr>
<td>CKA_GOST28147_PARAMS^{4,6,8}</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the data object type of GOST 28147-89.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
</tbody>
</table>

13699 Refer to Table 11 for footnotes

13700 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.

13704 The following is a sample template for creating an GOST R 34.10-2001 private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE gostR3410params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[32] = {...};
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_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_GOSTR3410_PARAMS, gostR3410params_oid, sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411_PARAMS, gostR3411params_oid, sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid, sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```
6.57.4 GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class CKO_DOMAIN_PARAMETERS, key type CKK_GOSTR3410) hold GOST R 34.10-2001 domain parameters.

The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE¹</td>
<td>Byte array</td>
<td>DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type GostR3410-2001-ParamSetParameters)</td>
</tr>
<tr>
<td>CKA_OBJECT_ID¹</td>
<td>Byte array</td>
<td>DER-encoding of the object identifier indicating the domain parameters</td>
</tr>
</tbody>
</table>

¹ Refer to Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that CKA_VALUE attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```c
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic parameters object";
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_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```
6.57.5 GOST R 34.10-2001 mechanism parameters

♦ CK_GOSTR3410_KEY_WRAP_PARAMS

CK_GOSTR3410_KEY_WRAP_PARAMS is a structure that provides the parameters to the CKM_GOSTR3410_KEY_WRAP mechanism. It is defined as follows:

```c
typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {
    CK_BYTE_PTR      pWrapOID;
    CK_ULONG         ulWrapOIDLen;
    CK_BYTE_PTR      pUKM;
    CK_ULONG         ulUKMLen;
    CK_OBJECT_HANDLE hKey;
} CK_GOSTR3410_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

- **pWrapOID** 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.

- **ulWrapOIDLen** length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89.

- **pUKM** 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-NUL_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.

- **ulUKMLen** length of UKM data. If pUKM-pointer is different from NULL_PTR then equal to 8.

- **hKey** 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.

♦ CK_GOSTR3410_DERIVE_PARAMS

CK_GOSTR3410_DERIVE_PARAMS is a structure that provides the parameters to the CKM_GOSTR3410_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_GOSTR3410_DERIVE_PARAMS {
    CK_EC_KDF_TYPE kdf;
    CK_BYTE_PTR    pPublicData;
    CK_ULONG       ulPublicDataLen;
} CK_GOSTR3410_DERIVE_PARAMS;
```
The fields of the structure have the following meanings:

- `kdf`: additional key diversification algorithm identifier. Possible values are CKD_NULL and CKD_CPDIFFERENTIATE_KDF. In case of CKD_NULL, the result of the key derivation function described in [RFC 4357], section 5.2 is used directly; in case of CKD_CPDIFFERENTIATE_KDF, the resulting key value is additionally processed with algorithm from [RFC 4357], section 6.5.
- `pPublicData`: pointer to data with public key of a receiver
- `ulPublicDataLen`: length of data with public key of a receiver (must be 64)
- `pUKM`: pointer to a UKM data
- `ulUKMLen`: length of UKM data in bytes (must be 8)

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 them is 32 bytes long and represented in little endian order.

CK_GOSTR3410_DERIVE_PARAMS_PTR is a pointer to a CK_GOSTR3410_DERIVE_PARAMS.

### 6.57.6 GOST R 34.10-2001 key pair generation

The GOST R 34.10-2001 key pair generation mechanism, denoted CKM_GOSTR3410_KEY_PAIR_GEN, is a key pair generation mechanism for GOST R 34.10-2001.

This mechanism does not have a parameter.

The mechanism generates GOST R 34.10-2001 public/private key pairs with particular GOST R 34.10-2001 domain parameters, as specified in the `CKA_GOSTR3410_PARAMS`, `CKA_GOSTR3411_PARAMS`, and `CKA_GOST28147_PARAMS` attributes of the template for the public key. Note that `CKA_GOST28147_PARAMS` attribute may not be present in the template.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, and `CKA_VALUE` attributes to the new public key and the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_VALUE`, and `CKA_GOSTR3410_PARAMS`, `CKA_GOSTR3411_PARAMS`, `CKA_GOST28147_PARAMS` attributes to the new private key.

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure are not used.

### 6.57.7 GOST R 34.10-2001 without hashing

The GOST R 34.10-2001 without hashing mechanism, denoted CKM_GOSTR3410, is a mechanism for single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

This mechanism does not have a parameter.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values s and r', both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.
The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

### Table 246, GOST R 34.10-2001 without hashing: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign(^1)</td>
<td>CKK_GOSTR3410</td>
<td>32 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>CKK_GOSTR3410</td>
<td>32 bytes</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

#### 6.57.8 GOST R 34.10-2001 with GOST R 34.11-94

The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM_GOSTR3410_WITH_GOSTR3411**, is a mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are specified in object identifier of attribute **CKA_GOSTR3411_PARAMS** must be used.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values \(s\) and \(r'\), both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is signed or should be signed message of any length. Single- and multiple-part signature operations are available.

### Table 247, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>CKK_GOSTR3410</td>
<td>Any</td>
<td>64 bytes</td>
</tr>
<tr>
<td>C_Verify</td>
<td>CKK_GOSTR3410</td>
<td>Any</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

#### 6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by **CKM_GOSTR3410_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used. Encrypted key is a DER-encoded structure of ASN.1 GostR3410-KeyTransport type [RFC 4490] section 4.2.

It has a parameter, a **CK_GOSTR3410_KEY_WRAP_PARAMS** structure defined in section 6.57.5.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA_VALUE** attribute of the new key.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used.

#### 6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys

Common key derivation, denoted **CKM_GOSTR3410_DERIVE**, is a mechanism for key derivation with assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object
class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation from [RFC 4357] (section 5.2) must be used. The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other attributes must be specified in a template for creating private key object.

### 6.58 ChaCha20

ChaCha20 is a secret-key stream cipher described in [CHACHA].

**Table 248, ChaCha20 Mechanisms vs. Functions**

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; 1 VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_CHACHA20_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CHACHA20</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 6.58.1 Definitions

This section defines the key type “CKK_CHACHA20” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_CHACHA20_KEY_GEN
- CKM_CHACHA20

### 6.58.2 ChaCha20 secret key objects

ChaCha20 secret key objects (object class CKO_SECRET_KEY, key type CKK_CHACHA20) hold ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the common attributes defined for this object class:

**Table 249, ChaCha20 Secret Key Object**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE(^{1,4,6,7})</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN(^{2,3})</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a ChaCha20 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CHACHA20;
CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
CK_BYTE value[32] = {...};
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_ENCCRYPT, &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 ChaCha20 secret key object's CKA_VALUE attribute.

### 6.58.3 ChaCha20 mechanism parameters

#### CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR

CK_CHACHA20_PARAMS provides the parameters to the CKM_CHACHA20 mechanism. It is defined as follows:

```c
typedef struct CK_CHACHA20_PARAMS {
    CK_BYTE_PTR pBlockCounter;
    CK_ULONG blockCounterBits;
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceBits;
} CK_CHACHA20_PARAMS;
```

The fields of the structure have the following meanings:

- `pBlockCounter` pointer to block counter
- `ulblockCounterBits` length of block counter in bits (can be either 32 or 64)
- `pNonce` nonce (This should be never re-used with the same key.)
- `ulNonceBits` length of nonce in bits (is 64 for original, 96 for IETF and 192 for xchacha20 variant)

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

CK_CHACHA20_PARAMS_PTR is a pointer to CK_CHACHA20_PARAMS.

### 6.58.4 ChaCha20 key generation

The ChaCha20 key generation mechanism, denoted CKM_CHACHA20_KEY_GEN, is a key generation mechanism for ChaCha20.

It does not have a parameter.

The mechanism generates ChaCha20 keys of 256 bits.

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. As a practical matter, the key size for ChaCha20 is fixed at 256 bits.

### 6.58.5 ChaCha20 mechanism

ChaCha20, denoted CKM_CHACHA20, is a mechanism for single and multiple-part encryption and decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and handling of their nonces, affecting the safety of using random nonces and the maximum size that can be encrypted safely.
Chacha20 has a parameter, `CK_CHACHA20_PARAMS`, which indicates the nonce and initial block counter value.

Constraints on key types and the length of input and output data are summarized in the following table:

### Table 250, ChaCha20: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>ChaCha20</td>
<td>Any / only up to 256 GB in case of IETF variant</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>ChaCha20</td>
<td>Any / only up to 256 GB in case of IETF variant</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of ChaCha20 key sizes, in bits.

### Table 251, ChaCha20: Nonce and block counter lengths

<table>
<thead>
<tr>
<th>Variant</th>
<th>Nonce</th>
<th>Block counter</th>
<th>Maximum message</th>
<th>Nonce generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>64 bit</td>
<td>64 bit</td>
<td>Virtually unlimited</td>
<td><code>1st msg: nonce0=random, n^th msg: nonce_{n-1}++</code></td>
</tr>
<tr>
<td>IETF</td>
<td>96 bit</td>
<td>32 bit</td>
<td>Max ~256 GB</td>
<td><code>1st msg: nonce0=random, n^th msg: nonce_{n-1}++</code></td>
</tr>
<tr>
<td>XChaCha20</td>
<td>192 bit</td>
<td>64 bit</td>
<td>Virtually unlimited</td>
<td>Each nonce can be randomly generated.</td>
</tr>
</tbody>
</table>

Nonces must not ever be reused with the same key. However due to the birthday paradox the first two variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.

Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

### 6.59 Salsa20

Salsa20 is a secret-key stream cipher described in [SALSA].

### Table 252, Salsa20 Mechanisms vs. Functions
Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_SALSA20_KEYGEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SALSA20</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

6.59.1 Definitions

This section defines the key type “CKK_SALSA20” and “CKK_SALSA20” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SALSA20_KEYGEN
CKM_SALSA20

6.59.2 Salsa20 secret key objects

Salsa20 secret key objects (object class CKO_SECRET_KEY, key type CKK_SALSA20) hold Salsa20 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common attributes defined for this object class:

Table 253, ChaCha20 Secret Key Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a Salsa20 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SALSA20;
CK_UTF8CHAR label[] = “A Salsa20 secret key object”;
CK_BYTE value[32] = {...};
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_ENCRYPT, &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 ChaCha20 secret key object’s CKA_VALUE attribute.
6.59.3 Salsa20 mechanism parameters

♦ CK_SALSA20_PARAMS; CK_SALSA20_PARAMS_PTR

CK_SALSA20_PARAMS provides the parameters to the CKM_SALSA20 mechanism. It is defined as follows:

```c
typedef struct CK_SALSA20_PARAMS {
    CK_BYTE_PTR pBlockCounter;
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceBits;
} CK_SALSA20_PARAMS;
```

The fields of the structure have the following meanings:

- **pBlockCounter**: pointer to block counter (64 bits)
- **pNonce**: nonce
- **ulNonceBits**: size of the nonce in bits (64 for classic and 192 for XSalsa20)

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

CK_SALSA20_PARAMS_PTR is a pointer to CK_SALSA20_PARAMS.

6.59.4 Salsa20 key generation

The Salsa20 key generation mechanism, denoted CKM_SALSA20_KEY_GEN, is a key generation mechanism for Salsa20.

It does not have a parameter.

The mechanism generates Salsa20 keys of 256 bits.

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. As a practical matter, the key size for Salsa20 is fixed at 256 bits.

6.59.5 Salsa20 mechanism

Salsa20, denoted CKM_SALSA20, is a mechanism for single and multiple-part encryption and decryption based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and handling of their nonces, affecting the safety of using random nonces.

Salsa20 has a parameter, CK_SALSA20_PARAMS, which indicates the nonce and initial block counter value.

Constraints on key types and the length of input and output data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Encrypt</td>
<td>Salsa20</td>
<td>Any</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>Salsa20</td>
<td>Any</td>
<td>Same as input length</td>
<td>No final part</td>
</tr>
</tbody>
</table>
For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of ChaCha20 key sizes, in bits.

Table 255, Salsa20: Nonce sizes

<table>
<thead>
<tr>
<th>Variant</th>
<th>Nonce</th>
<th>Maximum message</th>
<th>Nonce generation</th>
</tr>
</thead>
</table>
| original  | 64 bit| Virtually unlimited| 1st msg: nonce = random
|           |       |                   | n^n msg: nonce = n++ |
| XSalsa20  | 192 bit| Virtually unlimited| Each nonce can be randomly generated. |

Nonces must not ever be reused with the same key. However due to the birthday paradox the original variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages. Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

6.60 Poly1305

Poly1305 is a message authentication code designed by D.J Bernstein [POLY1305]. Poly1305 takes a 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

Table 256, Poly1305 Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_POLY1305_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_POLY1305</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.60.1 Definitions

This section defines the key type “CKK_POLY1305” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_POLY1305_KEY_GEN
- CKM_POLY1305

6.60.2 Poly1305 secret key objects

Poly1305 secret key objects (object class CKO_SECRET_KEY, key type CKK_POLY1305) hold Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the common attributes defined for this object class:

Table 257, Poly1305 Secret Key Object
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE^1,4,8,7</td>
<td>Byte array</td>
<td>Key length is fixed at 256 bits. Bit length restricted to a byte array.</td>
</tr>
<tr>
<td>CKA_VALUE_LEN^2,3</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

The following is a sample template for creating a Poly1305 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_POLY1305;
CK_UTF8CHAR label[] = "A Poly1305 secret key object";
CK_BYTE value[32] = {...};
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_SIGN, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.60.3 Poly1305 mechanism

Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key and arbitrary length input.

It has no parameters.

Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

Table 258, Poly1305: Key and Data Length

<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Data length</th>
<th>Signature Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign</td>
<td>Poly1305</td>
<td>Any</td>
<td>128 bits</td>
</tr>
<tr>
<td>C_Verify</td>
<td>Poly1305</td>
<td>Any</td>
<td>128 bits</td>
</tr>
</tbody>
</table>

6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption

The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

Table 259, Poly1305 Mechanisms vs. Functions
### 6.61.1 Definitions

**Mechanisms:**
- CKM_CHACHA20_POLY1305
- CKM_SALSA20_POLY1305

### 6.61.2 Usage

Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].

To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAMS for Encrypt, Decrypt and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS for MessageEncrypt, and MessageDecrypt.

#### Encrypt:
- Set the Nonce length \(ul\text{NonceLen}\) in the parameter block. (this affects which variant of ChaCha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data \(p\text{Nonce}\) in the parameter block.
- Set the AAD data \(p\text{AAD}\) and size \(ul\text{AADLen}\) in the parameter block. \(p\text{AAD}\) may be NULL if \(ul\text{AADLen}\) is 0.
- Call C_EncryptInit() for CKM_CHACHA20_POLY1305 or CKM_SALSA20_POLY1305 mechanism with parameters and key \(K\).
- Call C_Encrypt(), or C_EncryptUpdate()\(^{10}\) C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

#### Decrypt:
- Set the Nonce length \(ul\text{NonceLen}\) in the parameter block. (this affects which variant of ChaCha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data \(p\text{Nonce}\) in the parameter block.
- Set the AAD data \(p\text{AAD}\) and size \(ul\text{AADLen}\) in the parameter block. \(p\text{AAD}\) may be NULL if \(ul\text{AADLen}\) is 0.
- Call C_DecryptInit() for CKM_CHACHA20_POLY1305 or CKM_SALSA20_POLY1305 mechanism with parameters and key \(K\).
- Call C_Decrypt(), or C_DecryptUpdate()\(^{11}\) C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since CKM_CHACHA20_POLY1305 and CKM_SALSA20_POLY1305 are AEAD ciphers, no data should be returned until C_Decrypt() or C_DecryptFinal().

---

\(^{10}\) "\(^{**}\) indicates 0 or more calls may be made as required

\(^{11}\) Added padding
MessageEncrypt:::

- Set the Nonce length \textit{ulNonceLen} in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data \textit{pNonce} in the parameter block.
- Set \textit{pTag} to hold the tag data returned from \texttt{C\_EncryptMessage()} or the final \texttt{C\_EncryptMessageNext()}.
- Call \texttt{C\_MessageEncryptInit()} for \texttt{CKM\_CHACHA20\_POLY1305} or \texttt{CKM\_SALSA20\_POLY1305} mechanism with key \textit{K}.
- Call \texttt{C\_EncryptMessage()}, or \texttt{C\_EncryptMessageBegin} followed by \texttt{C\_EncryptMessageNext()}\textsuperscript{11}. The mechanism parameter is passed to all three of these functions.
- Call \texttt{C\_MessageEncryptFinal()} to close the message decryption.

MessageDecrypt:

- Set the Nonce length \textit{ulNonceLen} in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data \textit{pNonce} in the parameter block.
- Set the tag data \textit{pTag} in the parameter block before \texttt{C\_DecryptMessage} or the final \texttt{C\_DecryptMessageNext()}.
- Call \texttt{C\_MessageDecryptInit()} for \texttt{CKM\_CHACHA20\_POLY1305} or \texttt{CKM\_SALSA20\_POLY1305} mechanism with key \textit{K}.
- Call \texttt{C\_DecryptMessage()}, or \texttt{C\_DecryptMessageBegin} followed by \texttt{C\_DecryptMessageNext()}\textsuperscript{12}. The mechanism parameter is passed to all three of these functions.
- Call \texttt{C\_MessageDecryptFinal()} to close the message decryption.

\textit{ulNonceLen} is the length of the nonce in bits.

In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in the \textit{pTag} field of \texttt{CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS}. In MessageDecrypt the tag is provided by the \textit{pTag} field of \texttt{CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS}. The application must provide 16 bytes of space for the tag.

The key type for \textit{K} must be compatible with \texttt{CKM\_CHACHA20} or \texttt{CKM\_SALSA20} respectively and the \texttt{C\_EncryptInit}/\texttt{C\_DecryptInit} calls shall behave, with respect to \textit{K}, as if they were called directly with \texttt{CKM\_CHACHA20} or \texttt{CKM\_SALSA20}, \textit{K} and NULL parameters.

Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose the block counter, as the AEAD construction is based on a message metaphor in which random access is not needed.

\textsuperscript{11} **"** indicates 0 or more calls may be made as required

\textsuperscript{12} **"** indicates 0 or more calls may be made as required
6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

♦ CK_SALSA20_CHACHA20_POLY1305_PARAMS;
CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR

CK_SALSA20_CHACHA20_POLY1305_PARAMS is a structure that provides the parameters to the
CKM_CHACHA20_POLY1305 and CKM_SALSA20_POLY1305 mechanisms. It is defined as follows:

typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceLen;
    CK_BYTE_PTR pAAD;
    CK_ULONG ulAADLen;
} CK_SALSA20_CHACHA20_POLY1305_PARAMS;

The fields of the structure have the following meanings:

pNonce        nonce (This should be never re-used with the same key.)
ulNonceLen    length of nonce in bits (is 64 for original, 96 for IETF (only for
              chaacha20) and 192 for xchaacha20/xsalsa20 variant)

CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR is a pointer to
CK_SALSA20_CHACHA20_POLY1305_PARAMS.

♦ CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR

CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS is a structure that provides the parameters to the
CKM_CHACHA20_POLY1305 mechanism. It is defined as follows:

typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
    CK_BYTE_PTR pNonce;
    CK_ULONG ulNonceLen;
    CK_BYTE_PTR pTag;
} CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;

The fields of the structure have the following meanings:

pNonce        pointer to nonce
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)

pTag          location of the authentication tag which is returned on
              MessageEncrypt, and provided on MessageDecrypt.

CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR is a pointer to
CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS.
6.62 HKDF Mechanisms

Details for HKDF key derivation mechanisms can be found in [RFC 5869].

Table 260, HKDF Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_HKDF_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_HKDF_DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_HKDF_KEY_GEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

6.62.1 Definitions

Mechanisms:

- CKM_HKDF_DERIVE
- CKM_HKDF_DATA
- CKM_HKDF_KEY_GEN

Key Types:

- CKK_HKDF

6.62.2 HKDF mechanism parameters

**CK_HKDF_PARAMS; CK_HKDF_PARAMS_PTR**

CK_HKDF_PARAMS is a structure that provides the parameters to the CKM_HKDF_DERIVE and CKM_HKDF_DATA mechanisms. It is defined as follows:

```c
typedef struct CK_HKDF_PARAMS {
    CK_BBOOL bExtract;
    CK_BBOOL bExpand;
    CK_MECHANISM_TYPE prfHashMechanism;
    CK_ULONG ulSaltType;
    CK_BYTE_PTR pSalt;
    CK_ULONG ulSaltLen;
    CK_OBJECT_HANDLE hSaltKey;
    CK_BYTE_PTR pInfo;
    CK_ULONG ulInfoLen;
} CK_HKDF_PARAMS;
```

The fields of the structure have the following meanings:

- `bExtract`: execute the extract portion of HKDF.
- `bExpand`: execute the expand portion of HKDF.
- `prfHashMechanism`: base hash used for the HMAC in the underlying HKDF operation.
- `ulSaltType`: specifies how the salt for the extract portion of the KDF is supplied.
CKF_HKDF_SALT_NULL no salt is supplied.

CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with length ulSaltLen.

CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey.

 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.

Table 261, **CKM_NULL** Mechanisms vs. Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR¹</th>
<th>Digest</th>
<th>Gen. Key/ Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_NULL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

¹SR = SignRecover, VR = VerifyRecover

6.63.1 Definitions

Mechanisms:

- **CKM_NULL**

6.63.2 **CKM_NULL** mechanism parameters

CKM_NULL does not have a parameter.

When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

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.

When used for digesting data, the input data is copied to the message digest.

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.

When used for deriving a key, the derived key has the same value as the base key.

6.64 IKE Mechanisms

Table 262, **IKE** Mechanisms vs. Functions
### Functions

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Encrypt &amp; Decrypt</th>
<th>Sign &amp; Verify</th>
<th>SR &amp; VR</th>
<th>Digest</th>
<th>Gen. Key/Key Pair</th>
<th>Wrap &amp; Unwrap</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_IKE2_PRF_PLUS_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_IKE_PRF_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_IKE1_PRF_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_IKE1_EXTENDED_DERIVE</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.64.1 Definitions

Mechanisms:

- CKM_IKE2_PRF_PLUS_DERIVE
- CKM_IKE_PRF_DERIVE
- CKM_IKE1_PRF_DERIVE
- CKM_IKE1_EXTENDED_DERIVE

### 6.64.2 IKE mechanism parameters

**CK_IKE2_PRF_PLUS_DERIVE_PARAMS;**

**CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR**

CK_IKE2_PRF_PLUS_DERIVE_PARAMS is a structure that provides the parameters to the CKM_IKE2_PRF_PLUS_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_IKE2_PRF_PLUS_DERIVE_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BBOOL bHasSeedKey;
    CK_OBJECT_HANDLE hSeedKey;
    CK_BYTE_PTR pSeedData;
    CK_ULONG ulSeedDataLen;
} CK_IKE2_PRF_PLUS_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- prfMechanism: underlying MAC mechanism used to generate the prf
- bHasSeedKey: hSeed key is present
- hSeedKey: optional seed from key
- pSeedData: optional seed from data
- ulSeedDataLen: length of optional seed data. If no seed data is present this value is 0

**CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR** is a pointer to a CK_IKE2_PRF_PLUS_DERIVE_PARAMS.
CK_IKE_PRF_DERIVE_PARAMS; CK_IKE_PRF_DERIVE_PARAMS_PTR

CK_IKE_PRF_DERIVE_PARAMS is a structure that provides the parameters to the CKM_IKE_PRF_DERIVE mechanism. It is defined as follows:

typedef struct CK_IKE_PRF_DERIVE_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BBOOL bDataAsKey;
    CK_BBOOL bRekey;
    CK_BYTE_PTR pNi;
    CK_ULONG ulNiLen;
    CK_BYTE_PTR pNr;
    CK_ULONG ulNrLen;
    CK_OBJECT_HANDLE hNewKey;
} CK_IKE_PRF_DERIVE_PARAMS;

The fields of the structure have the following meanings:
prfMechanism    underlying MAC mechanism used to generate the prf
bDataAsKey      Ni||Nr is used as the key for the prf rather than baseKey
bRekey          rekey operation. hNewKey must be present
pNi             Ni value
ulNiLen          length of Ni
pNr             Nr value
ulNrLen          length of Nr
hNewKey          New key value to drive the rekey.

CK_IKE_PRF_DERIVE_PARAMS_PTR is a pointer to a CK_IKE_PRF_DERIVE_PARAMS.

CK_IKE1_PRF_DERIVE_PARAMS; CK_IKE1_PRF_DERIVE_PARAMS_PTR

CK_IKE1_PRF_DERIVE_PARAMS is a structure that provides the parameters to the CKM_IKE1_PRF_DERIVE mechanism. It is defined as follows:

typedef struct CK_IKE1_PRF_DERIVE_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BBOOL bHasPrevKey;
    CK_OBJECT_HANDLE hKeygxy;
    CK_OBJECT_HANDLE hPrevKey;
    CK_BYTE_PTR pCKYi;
    CK_ULONG ulCKYiLen;
    CK_BYTE_PTR pCKYr;
    CK_ULONG ulCKYrLen;
    CK_BYTE keyNumber;
} CK_IKE1_PRF_DERIVE_PARAMS;

The fields of the structure have the following meanings:
prfMechanism    underlying MAC mechanism used to generate the prf
bHasPrevKey  hPrevKey is present
hKeygxy  handle to the exchanged g^xy key
hPrevKey  handle to the previously derived key
pCKYi  CKYi value
ulCKYiLen  length of CKYi
pCKYr  CKYr value
ulCKYrLen  length of CKYr
keyNumber  unique number for this key derivation

CK_IKE1_PRF_DERIVE_PARAMS_PTR is a pointer to a CK_IKE1_PRF_DERIVE_PARAMS.

♦ CK_IKE1_EXTENDED_DERIVE_PARAMS;
CK_IKE1_EXTENDED_DERIVE_PARAMS_PTR

CK_IKE1_EXTENDED_DERIVE_PARAMS is a structure that provides the parameters to the
CKM_IKE1_EXTENDED_DERIVE mechanism. It is defined as follows:

typedef struct CK_IKE1_EXTENDED_DERIVE_PARAMS {
  CK_MECHANISM_TYPE  prfMechanism;
  CK_BBOOL  bHasKeygxy;
  CK_OBJECT_HANDLE  hKeygxy;
  CK_BYTE_PTR  pExtraData;
  CK_ULONG  ulExtraDataLen;
} CK_IKE1_EXTENDED_DERIVE_PARAMS;

The fields of the structure have the following meanings:

prfMechanism  underlying MAC mechanism used to generate the prf
bHasKeygxy  hKeygxy key is present
hKeygxy  optional key g^xy
pExtraData  optional extra data
ulExtraDataLen  length of optional extra data. If no extra data is present this value is
0

CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR is a pointer to a
CK_IKE2_PRF_PLUS_DERIVE_PARAMS.

6.64.3 IKE PRF DERIVE

The IKE PRF Derive mechanism denoted CKM_IKE_PRF_DERIVE is used in IPSEC both IKEv1 and
IKEv2 to generate an initial key that is used to generate additional keys. It takes a
CK_IKE_PRF_DERIVE_PARAMS as a mechanism parameter. baseKey is the base key passed into
C_DeriveKey. baseKey must be of type CKK_GENERIC_SECRET if bDataAsKey is TRUE and the key
type of the underlying prf if bDataAsKey is FALSE. hNewKey must be of type CKK_GENERIC_SECRET.
Depending on the parameter settings, it generates keys with a CKA_VALUE of:

1. prf(pNi|pNr, baseKey); (bDataAsKey=TRUE, bRekey=FALSE)
2. prf(baseKey, pNi|pNr); (bDataAsKey=FALSE, bRekey=FALSE)
3. prf(baseKey, ValueOf(hNewKey)| pNi | pNr); (bDataAsKey=FALSE, bRekey=TRUE)

The resulting output key is always the length of the underlying prf. The combination of
bDataAsKey=TRUE and bRekey=TRUE is not allowed. If both are set, CKR_ARGUMENTS_BAD is
returned.

Case 1 is used in
a. ikev2 (RFC 5996) baseKey is called g^ir, the output is called SKEYSEED
b. ikev1 (RFC 2409) baseKey is called g^ir, the output is called SKEYID

Case 2 is used in ikev1 (RFC 2409) inkey is called pre-shared-key, output is called SKEYID
Case 3 is used in ikev2 (RFC 5996) rekey case, baseKey is SK_d, hNewKey is g^ir (new), the output is
called SKEYSEED. The derived key will have a length of the length of the underlying prf. If
CKA_VALUE_LEN is specified, it must equal the underlying prf or CKR_KEY_SIZE_RANGE is returned.
If CKA_KEY_TYPE is not specified in the template, it will be the underlying key type of the prf.

6.64.4 IKEv1 PRF DERIVE

The IKEv1 PRF Derive mechanism denoted CKM_IKE1_PRF_DERIVE is used in IPSEC IKEv1 to
generate various additional keys from the initial SKEYID. It takes a CK_IKE1_PRF_DERIVE_PARAMS
as a mechanism parameter. SKEYID is the base key passed into C_DeriveKey.

This mechanism derives a key with CKA_VALUE set to either:
prf(baseKey, ValueOf(hKeygxy) || pCKYi || pCKYr || key_number)
or
prf(baseKey, ValueOf(hPrevKey) || ValueOf(hKeygxy) || pCKYi || pCKYr || key_number)
depending on the state of bHasPrevKey.
The key type of baseKey must be the key type of the prf, and the key type of hKeygxy must be
CKK_GENERIC_SECRET. The key type of hPrevKey can be any key type.

This is defined in RFC 2409. For each of the following keys.
baseKey is SKEYID, hKeygxy is g^xy
for outKey = SKEYID_d, bHasPrevKey = false, key_number = 0
for outKey = SKEYID_a, hPrevKey = SKEYID_d, key_number = 1
for outKey = SKEYID_e, hPrevKey = SKEYID_a, key_number = 2
If CKA_VALUE_LEN is not specified, the resulting key will be the length of the prf. If CKA_VALUE_LEN
is greater then the prf, CKR_KEY_SIZE_RANGE is returned. If it is less the key is truncated taking the
left most bytes. The value CKA_KEY_TYPE must be specified in the template or
CKR_TEMPLATE_INCOMPLETE is returned.

6.64.5 IKEv2 PRF PLUS DERIVE

The IKEv2 PRF PLUS Derive mechanism denoted CKM_IKE2_PRF_PLUS_DERIVE is used in IPSEC
IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a
CK_IKE2_PRF_PLUS_DERIVE_PARAMS as a mechanism parameter. SKEYSEED is the base key
passed into C_DeriveKey. The key type of baseKey must be the key type of the underlying prf. This
mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient
bytes to cover all additional keys. The application will then use CKM_EXTRACT_KEY_FROM_KEY
several times to pull out the various keys. CKA_VALUE_LEN must be set in the template and its value
must not be bigger than 255 times the size of the prf function output or \texttt{CKR\_KEY\_SIZE\_RANGE} will be returned. If \texttt{CKA\_KEY\_TYPE} is not specified, the output key type will be \texttt{CKK\_GENERIC\_SECRET}.

This mechanism derives a key with a \texttt{CKA\_VALUE} of (from RFC 5996):

\[
\text{prfplus} = T1 \mid T2 \mid T3 \mid T4 \mid \ldots \mid Tn
\]

where:

\[
\begin{align*}
T1 &= \text{prf}(K, S \mid 0x01) \\
T2 &= \text{prf}(K, T1 \mid S \mid 0x02) \\
T3 &= \text{prf}(K, T3 \mid S \mid 0x03) \\
T4 &= \text{prf}(K, T4 \mid S \mid 0x04) \\
\vdots \\
Tn &= \text{prf}(K, T(n-1) \mid n)
\end{align*}
\]

\[
K = \text{baseKey}, \ S = \text{valueOf(hSeedKey)} \mid pSeedData
\]

6.64.6 IKEv1 Extended Derive

The IKE Extended Derive mechanism denoted \texttt{CKM\_IKE1\_EXTENDED\_DERIVE} is used in IPSEC IKEv1 to derive longer keys than \texttt{CKM\_IKE1\_EXTENDED\_DERIVE} can from the initial SKEYID. It is used to support RFC 2409 appendix B and RFC 2409 section 5.5 (Quick Mode). It takes a \texttt{CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS} as a mechanism parameter. SKEYID is the base key passed into \texttt{C\_DeriveKey}. \texttt{CKA\_VALUE\_LEN} must be set in the template and its value must not be bigger than 255 times the size of the prf function output or \texttt{CKR\_KEY\_SIZE\_RANGE} will be returned. If \texttt{CKA\_KEY\_TYPE} is not specified, the output key type will be \texttt{CKK\_GENERIC\_SECRET}. The key type of SKEYID must be the key type of the prf, and the key type of \texttt{hKeygxy} (if present) must be \texttt{CKK\_GENERIC\_SECRET}.

This mechanism derives a key with \texttt{CKA\_VALUE} (from RFC 2409 appendix B and section 5.5):

\[
K_a = K1 \mid K2 \mid K3 \mid K4 \mid \ldots \mid Kn
\]

where:

\[
\begin{align*}
K1 &= \text{prf}(K, \text{valueOf(hKeygxy)}\mid pExtraData) \text{ or } \text{prf}(K,0x00) \text{ if } bHashKeygxy \text{ is FALSE and } ulExtraData \\
&\text{is } 0 \\
K2 &= \text{prf}(K, K1\mid pExtraData) \\
K3 &= \text{prf}(K, K2\mid pExtraData) \\
K4 &= \text{prf}(K, K3\mid pExtraData) \\
\vdots \\
K_n &= \text{prf}(K, K(n-1)\mid pExtraData)
\end{align*}
\]

\[
K = \text{baseKey}
\]

If \texttt{CKA\_VALUE\_LEN} is less than or equal to the prf length and bHasKeygxy is CK\_FALSE, then the new key is simply the base key truncated to \texttt{CKA\_VALUE\_LEN} (specified in RFC 2409 appendix B). Otherwise the prf is executed and the derived keys value is \texttt{CKA\_VALUE\_LEN} bytes of the resulting prf.

6.65 HSS

HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm defined in [RFC 8554] and [NIST 802-208].
6.65.1 Definitions
This section defines the key type \texttt{CKK\_HSS} for type \texttt{CK\_KEY\_TYPE} as used in the \texttt{CKA\_KEY\_TYPE} attribute of key objects and domain parameter objects.

Mechanisms:
- \texttt{CKM\_HSS\_KEY\_PAIR\_GEN}
- \texttt{CKM\_HSS}

6.65.2 HSS public key objects
HSS public key objects (object class \texttt{CKO\_PUBLIC\_KEY}, key type \texttt{CKK\_HSS}) hold HSS public keys.

The following table defines the HSS public key object attributes, in addition to the common attributes defined for this object class:

Table 264, HSS Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CKA_HSS_LEVELS}</td>
<td>\texttt{CK_ULONG}</td>
<td>The number of levels in the HSS scheme.</td>
</tr>
<tr>
<td>\texttt{CKA_HSS_LMS_TYPE}</td>
<td>\texttt{CK_ULONG}</td>
<td>The encoding for the Merkle tree heights of the top level LMS tree in the hierarchy.</td>
</tr>
<tr>
<td>\texttt{CKA_HSS_LMOTS_TYPE}</td>
<td>\texttt{CK_ULONG}</td>
<td>The encoding for the Winternitz parameter of the one-time-signature scheme of the top level LMS tree.</td>
</tr>
<tr>
<td>\texttt{CKA_VALUE}</td>
<td>Byte array</td>
<td>XDR-encoded public key as defined in \cite{RFC8554}.</td>
</tr>
</tbody>
</table>

Refer to Table 11 for footnotes

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

\begin{verbatim}
    CK\_OBJECT\_CLASS keyClass = CKO\_PUBLIC\_KEY;
    CK\_KEY\_TYPE keyType = CKK\_HSS;
    CK\_UTF8CHAR label[] = "An HSS public key object";
    CK\_BYTE value[] = {...};
\end{verbatim}
CK_BBOOL true = CK_TRUE;
CK_BBOOL false = CK_FALSE;

CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &false, sizeof(false)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_VALUE, value, sizeof(value)},
    {CKA_VERIFY, &true, sizeof(true)}
};

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
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_HSS_LEVELS&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>The number of levels in the HSS scheme.</td>
</tr>
<tr>
<td>CKA_HSS_LMS_TYPES&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>CK_ULONG_PTR</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_HSS_LMOTS_TYPES&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>CK_ULONG_PTR</td>
<td>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.</td>
</tr>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Byte array</td>
<td>Vendor defined, must include state information. Note that exporting this value is dangerous as it would allow key reuse.</td>
</tr>
<tr>
<td>CKA_HSS_KEYS_REMAINING&lt;sup&gt;2,4&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>The minimum of the following two values: 1) The number of one-time private keys remaining; 2) 2^32-1</td>
</tr>
</tbody>
</table>

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:

```c
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,...};
CK_ULONG lmotsTypes[] = {123,...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_BBOOL false = CK_FALSE;
CK_ATTRIBUTE template[] = {
```
{CKA_CLASS, &keyClass, sizeof(keyClass)},
{CKA_KEY_TYPE, &keyType, sizeof(keyType)},
{CKA_TOKEN, &true, sizeof(true)},
{CKA_LABEL, label, sizeof(label)-1},
{CKA_SENSITIVE, &true, sizeof(true)},
{CKA_EXTRACTABLE, &false, sizeof(true)},
{CKA_HSS_LEVELS, &hssLevels, sizeof(hssLevels)},
{CKA_HSS_LMS_TYPES, lmsTypes, sizeof(lmsTypes)},
{CKA_HSS_LMOTS_TYPES, lmotsTypes, sizeof(lmotsTypes)},
{CKA_VALUE, value, sizeof(value)},
{CKA_SIGN, &true, sizeof(true)}

CKA_SENSITIVE MUST be true, CKA_EXTRACTABLE MUST be false, and CKA_COPYABLE MUST be false for this key.

6.65.4 HSS key pair generation

The HSS key pair generation mechanism, denoted CKM_HSS_KEY_PAIR_GEN, is a key pair generation mechanism for HSS.

This mechanism does not have a parameter.

The mechanism generates HSS public/private key pairs for the scheme specified by the CKA_HSS_LEVELS, CKA_HSS_LMS_TYPES, and CKA_HSS_LMOTS_TYPES attributes of the template for the private key.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_HSS_LEVELS, CKA_HSS_LMS_TYPE, CKA_HSS_LMOTS_TYPE, and CKA_VALUE attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE, and CKA_HSS_KEYS_REMAINING attributes to the new private key.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure are not used and must be set to 0.

6.65.5 HSS without hashing

The HSS without hashing mechanism, denoted CKM_HSS, is a mechanism for single-part signatures and verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value, which may be of any length; it does not compute the hash value.)

This mechanism does not have a parameter.

For the purposes of these mechanisms, an HSS signature is a byte string with length depending on CKA_HSS_LEVELS, CKA_HSS_LMS_TYPES, CKA_HSS_LMOTS_TYPES as described in the following table.

Table 266, HSS without hashing: Key and Data Length
<table>
<thead>
<tr>
<th>Function</th>
<th>Key type</th>
<th>Input length</th>
<th>Output length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Sign¹</td>
<td>HSS Private Key</td>
<td>any</td>
<td>1296-74988²</td>
</tr>
<tr>
<td>C_Verify¹</td>
<td>HSS Public Key</td>
<td>any, 1296-74988²</td>
<td>N/A</td>
</tr>
</tbody>
</table>

¹ Single-part operations only.

² \(4+(\text{levels}-1)\times56+\text{levels} \times \left(8+\left(36+32 \times p\right)+h \times 32\right)\) where \(p\) has values \((265, 133, 67, 34)\) for lmots type \((W1, W2, W4, W8)\) and \(h\) is the number of levels in the LMS Merkle trees.

For this mechanism, the \(ul\text{MinKeySize}\) and \(ul\text{MaxKeySize}\) fields of the \texttt{CK\_MECHANISM\_INFO} structure are not used and must be set to 0.

If the number of signatures is exhausted, \texttt{CKR\_KEY\_EXHAUSTED} will be returned.
7 PKCS #11 Implementation Conformance

7.1 PKCS#11 Consumer Implementation Conformance

An implementation is a conforming PKCS#11 Consumer if the implementation meets the conditions specified in one or more consumer profiles specified in [PKCS11-Prof].

A PKCS#11 consumer implementation SHALL be a conforming PKCS#11 Consumer.

If a PKCS#11 consumer implementation claims support for a particular consumer profile, then the implementation SHALL conform to all normative statements within the clauses specified for that profile and for any subclauses to each of those clauses.

7.2 PKCS#11 Provider Implementation Conformance

An implementation is a conforming PKCS#11 Provider if the implementation meets the conditions specified in one or more provider profiles specified in [PKCS11-Prof].

A PKCS#11 provider implementation SHALL be a conforming PKCS#11 Provider.
Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

<table>
<thead>
<tr>
<th>Salutation</th>
<th>First Name</th>
<th>Last Name</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr.</td>
<td>Warren</td>
<td>Armstrong</td>
<td>QuintessenceLabs Pty Ltd.</td>
</tr>
<tr>
<td></td>
<td>Anthony</td>
<td>Berglas</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>Dieter</td>
<td>Bong</td>
<td>Utimaco IS GmbH</td>
</tr>
<tr>
<td>Mr.</td>
<td>Roland</td>
<td>Bramm</td>
<td>PrimeKey Solutions AB</td>
</tr>
<tr>
<td></td>
<td>Andrew</td>
<td>Byrne</td>
<td>Dell</td>
</tr>
<tr>
<td></td>
<td>Hamish</td>
<td>Cameron</td>
<td>nCipher</td>
</tr>
<tr>
<td></td>
<td>Kenli</td>
<td>Chong</td>
<td>QuintessenceLabs Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>Justin</td>
<td>Corlett</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
<tr>
<td></td>
<td>Xuelei</td>
<td>Fan</td>
<td>Oracle</td>
</tr>
<tr>
<td>Mr.</td>
<td>Jan</td>
<td>Friedel</td>
<td>Oracle</td>
</tr>
<tr>
<td>Ms.</td>
<td>Susan</td>
<td>Gleeson</td>
<td>Oracle</td>
</tr>
<tr>
<td>Mr.</td>
<td>Thomas</td>
<td>Hardjono</td>
<td>M.I.T.</td>
</tr>
<tr>
<td>Mrs.</td>
<td>Jane</td>
<td>Harnad</td>
<td>OASIS</td>
</tr>
<tr>
<td></td>
<td>David</td>
<td>Horton</td>
<td>Dell</td>
</tr>
<tr>
<td></td>
<td>Tim</td>
<td>Hudson</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>Gershon</td>
<td>Janssen</td>
<td>Individual</td>
</tr>
<tr>
<td>Mr.</td>
<td>Jakub</td>
<td>Jelen</td>
<td>Red Hat</td>
</tr>
<tr>
<td>Dr.</td>
<td>Mark</td>
<td>Joseph</td>
<td>P6R, Inc</td>
</tr>
<tr>
<td>Mr.</td>
<td>Paul</td>
<td>King</td>
<td>nCipher</td>
</tr>
<tr>
<td>Ms.</td>
<td>Dina</td>
<td>Kurktchi-Nimeh</td>
<td>Oracle</td>
</tr>
<tr>
<td></td>
<td>John</td>
<td>Leiseboer</td>
<td>QuintessenceLabs Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>John</td>
<td>Leser</td>
<td>Oracle</td>
</tr>
<tr>
<td></td>
<td>Chris</td>
<td>Malafis</td>
<td>Red Hat</td>
</tr>
<tr>
<td>Dr.</td>
<td>Michael</td>
<td>Markowitz</td>
<td>Information Security Corporation</td>
</tr>
<tr>
<td>Mr.</td>
<td>Scott</td>
<td>Marshall</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
<tr>
<td>Salutation</td>
<td>First Name</td>
<td>Last Name</td>
<td>Company</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Mr.</td>
<td>Chris</td>
<td>Meyer</td>
<td>Utimaco IS GmbH</td>
</tr>
<tr>
<td>Mr.</td>
<td>Darren</td>
<td>Moffat</td>
<td>Oracle</td>
</tr>
<tr>
<td>Dr.</td>
<td>Florian</td>
<td>Poppa</td>
<td>QuintessenceLabs Pty Ltd.</td>
</tr>
<tr>
<td></td>
<td>Roland</td>
<td>Reichenberg</td>
<td>Utimaco IS GmbH</td>
</tr>
<tr>
<td>Mr.</td>
<td>Robert</td>
<td>Relyea</td>
<td>Red Hat</td>
</tr>
<tr>
<td>Mr.</td>
<td>Jonathan</td>
<td>Schulze-Hewett</td>
<td>Information Security Corporation</td>
</tr>
<tr>
<td>Mr.</td>
<td>Greg</td>
<td>Scott</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>Martin</td>
<td>Shannon</td>
<td>QuintessenceLabs Pty Ltd.</td>
</tr>
<tr>
<td>Mr.</td>
<td>Oscar</td>
<td>So</td>
<td>Individual</td>
</tr>
<tr>
<td></td>
<td>Patrick</td>
<td>Steuer</td>
<td>IBM</td>
</tr>
<tr>
<td>Mr.</td>
<td>Gerald</td>
<td>Stueve</td>
<td>Fornetix</td>
</tr>
<tr>
<td></td>
<td>Jim</td>
<td>Susoy</td>
<td>P6R, Inc</td>
</tr>
<tr>
<td>Mr.</td>
<td>Sander</td>
<td>Temme</td>
<td>nCipher</td>
</tr>
<tr>
<td>Mr.</td>
<td>Manish</td>
<td>Upasani</td>
<td>Utimaco IS GmbH</td>
</tr>
<tr>
<td>Mr.</td>
<td>Charles</td>
<td>White</td>
<td>Fornetix</td>
</tr>
<tr>
<td>Ms.</td>
<td>Magda</td>
<td>Zdunkiewicz</td>
<td>Cryptsoft Pty Ltd.</td>
</tr>
</tbody>
</table>
Appendix B. Manifest constants

The definitions for manifest constants specified in this document can be found in the Additional artifacts following normative computer language definition files: section.

- pkcs11.h:
- pkcs11f.h:
- pkcs11t.h:
### Appendix C. Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSD02 WD01</td>
<td>02 December 2020</td>
<td>Dieter Bong &amp; Tony Cox</td>
<td>- Merged Base Changes made compared to Committee Specification &amp; Current</td>
</tr>
<tr>
<td></td>
<td>12 May 2022</td>
<td></td>
<td>Mechanisms forming new “PKCS#11 CSD01, as working draft of Committee</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Specification v3.1 CSD02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Editorial changes resolving comments by Paul Knight, OASIS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CKA_DERIVE TEMPLATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Reference [PKCS11-curr] replaced by reference within document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Correction of typos</td>
</tr>
<tr>
<td>WD02</td>
<td>04 December 2020</td>
<td>Dieter Bong &amp; Tony Cox</td>
<td>- Removed section 4.9.1 (covered in 6.1.3)</td>
</tr>
<tr>
<td>WD03</td>
<td>4 March 2021</td>
<td>Dieter Bong &amp; Tony Cox</td>
<td>- Section 6.3.8 2nd paragraph replace “Edwards” by “Montgomery”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Revised Note in § 5.2</td>
</tr>
<tr>
<td>WD04</td>
<td>1 June 2021</td>
<td>Daniel Minder &amp; Dieter Bong</td>
<td>- Fixed several references and typos</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Moved CKM_SHA224_RSA_PKCS and CKM_SHA224_RSA_PKCS_PSS from table 137 to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>table 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Fixed the typo and added the wording wrt. CKA_VALUE_LEN in sections 6.64.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and 6.64.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Section 4.9 Private key objects: replaced &quot;&lt;this version&gt;&quot; by “PKCS #11 V2.40”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Section 6.65 updated to HSS proposal dd. 12 May 2021</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Section 6.3: deprecation notice for CKM_ECDH_AES_KEY_WRAP</td>
</tr>
<tr>
<td>Revision</td>
<td>Date</td>
<td>Editor</td>
<td>Changes Made</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| WD05     | 15 July 2021   | Dieter Bong & Tony Cox| - Section 6.64: change the non-existing error CKR_KEY_RANGE_ERROR to CKR_KEY_SIZE_RANGE  
|          |                |                       | - Section 6.64.2: typo corrected: CK_IKE_PRF_PARAMS -> CK_IKE_PRF_DERIVE_PARAMS  
|          |                |                       | - Section 6.64.5: improved wording for IKE v2 key derivation  
|          |                |                       | - Section 6.64 and 6.65: formatting updated  
|          |                |                       | - Section 6.65: removed timeout error code and description  
|          |                |                       | - Section 5.9.5: Reported by Mostafa ADILI: C_EncryptMessageNext should be C_MessageEncryptFinal in the function declaration |
| WD06     | 14 October 2021| Dieter Bong & Tony Cox| - Added clarifying text to 6.64.6  
|          |                |                       | - Clarified deprecation statement for CKM_ECDH_AES_KEY_WRAP  
|          |                |                       | - Updated [PKCS11 Prof] Reference  
|          |                |                       | - Clarified encodings in sections 6.3.5, 6.3.6, 6.3.7, 6.3.8, 6.3.16 & 6.3.17 |
| WD07     | 23 November 2021| Dieter Bong & Tony Cox| - Further clarification for CKM_ECDH_AES_KEY_WRAP deprecation notice  
|          |                |                       | - Clarified multiple EC key references in §6 (insertion of short Weierstrass descriptor)  
|          |                |                       | - Correction to description of CK_RSA_PKCS_MGF_TYPE |
| WD08     | 9 December 2021 | Dieter Bong            | - Clarified a few more EC key references in §6 (insertion of short Weierstrass descriptor and/or key type CKK_EC) |
| WD09     | 14 December 2021| Dieter Bong            | - Updated deprecation notice for CKM_ECDH_AES_KEY_WRAP in section 6.3.20 |
| 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 |
Appendix D. Notices

Copyright © OASIS Open 2022. All Rights Reserved.

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

[https://www.oasis-open.org/policies-guidelines/ipr/].

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

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

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. OASIS AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS DOCUMENT OR ANY PART THEREOF.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

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

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

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

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