PKCS #11 v2.20: Cryptographic Token Interface Standard

RSA Laboratories

28 June 2004

Table of Contents

1  INTRODUCTION ................................................................................................................... ......... 1
2  SCOPE ............................................................................................................................................... 2
3  REFERENCES..................................................................................................................... ............. 3
4  DEFINITIONS.................................................................................................................... .............. 7
5  SYMBOLS AND ABBREVIATIONS........................................................................................... 10
6  GENERAL OVERVIEW ............................................................................................................... 12
   6.1  INTRODUCTION......................................................................................................................... 12
   6.2  DESIGN GOALS ......................................................................................................................... 13
   6.3  GENERAL MODEL ..................................................................................................................... 13
   6.4  LOGICAL VIEW OF A TOKEN .................................................................................................... 15
   6.5  USERS ....................................................................................................................................... 16
   6.6  APPLICATIONS AND THEIR USE OF CRYPTOoki ............................................................... 17
       6.6.1  Applications and processes ........................................................................................... 17
       6.6.2  Applications and threads ............................................................................................... 18
   6.7  SESSIONS ................................................................................................................................... 19
       6.7.1  Read-only session states .................................................................................................. 19
       6.7.2  Read/write session states ............................................................................................... 20
       6.7.3  Permitted object accesses by sessions ........................................................................... 21
       6.7.4  Session events ............................................................................................................... 22
       6.7.5  Session handles and object handles .............................................................................. 23
       6.7.6  Capabilities of sessions .................................................................................................. 23
       6.7.7  Example of use of sessions ........................................................................................... 24
   6.8  SECONDARY AUTHENTICATION (DEPRECATED)........................................................................ 26
   6.9  FUNCTION OVERVIEW............................................................................................................... 27
7  SECURITY CONSIDERATIONS ................................................................................................ 30
8  PLATFORM- AND COMPILER-DEPENDENT DIRECTIVES FOR C OR C++ ....................... 31
   8.1  STRUCTURE PACKING............................................................................................................. 31
   8.2  POINTER-RELATED MACROS .................................................................................................. 32
       ♦  CK_PTR ......................................................................................................................... ......... 32
       ♦  CK_DEFINE_FUNCTION...................................................................................................... 32
       ♦  CK_DECLARE_FUNCTION .................................................................................................. 32
       ♦  CK_DECLARE_FUNCTION_POINTER................................................................................ 32

Copyright © 1994-2004 RSA Security Inc. License to copy this document is granted provided that it is identified as “RSA Security Inc. Public-Key Cryptography Standards (PKCS)” in all material mentioning or referencing this document.
10.3.2 Overview ........................................................................................................... 67
10.3.3 Clock .................................................................................................................. 67
10.3.4 Monotonic Counter Objects ................................................................................. 68
10.3.5 User Interface Objects ......................................................................................... 69
10.4 STORAGE OBJECTS ................................................................................................. 71
10.5 DATA OBJECTS ....................................................................................................... 72
  10.5.1 Definitions ........................................................................................................... 72
  10.5.2 Overview ............................................................................................................. 72
10.6 CERTIFICATE OBJECTS .......................................................................................... 73
  10.6.1 Definitions ........................................................................................................... 73
  10.6.2 Overview ............................................................................................................. 73
  10.6.3 X.509 public key certificate objects ...................................................................... 74
  10.6.4 WTLS public key certificate objects .................................................................... 76
  10.6.5 X.509 attribute certificate objects ...................................................................... 78
10.7 KEY OBJECTS ........................................................................................................... 79
  10.7.1 Definitions ........................................................................................................... 79
  10.7.2 Overview ............................................................................................................. 79
10.8 PUBLIC KEY OBJECTS ............................................................................................. 81
10.9 PRIVATE KEY OBJECTS ........................................................................................... 82
10.10 SECRET KEY OBJECTS .......................................................................................... 84
10.11 DOMAIN PARAMETER OBJECTS ............................................................................ 87
  10.11.1 Definitions ........................................................................................................... 87
  10.11.2 Overview ............................................................................................................. 87
10.12 MECHANISM OBJECTS ............................................................................................ 88
  10.12.1 Definitions ........................................................................................................... 88
  10.12.2 Overview ............................................................................................................. 88
11 FUNCTIONS ..................................................................................................................... 89
  11.1 FUNCTION RETURN VALUES .................................................................................. 90
    11.1.1 Universal Cryptoki function return values ............................................................ 90
    11.1.2 Cryptoki function return values for functions that use a session handle .................. 91
    11.1.3 Cryptoki function return values for functions that use a token ................................. 92
    11.1.4 Special return value for application-supplied callbacks .......................................... 92
    11.1.5 Special return values for mutex-handling functions ............................................... 93
    11.1.6 All other Cryptoki function return values ............................................................... 93
    11.1.7 More on relative priorities of Cryptoki errors ........................................................ 100
    11.1.8 Error code “gotchas” ............................................................................................ 101
  11.2 CONVENTIONS FOR FUNCTIONS RETURNING OUTPUT IN A VARIABLE-LENGTH BUFFER .......... 101
  11.3 DISCLAIMER CONCERNING SAMPLE CODE .......................................................... 102
  11.4 GENERAL-PURPOSE FUNCTIONS ........................................................................... 102
    ◆ C_Initialize .................................................................................................................. 102
    ◆ C_Finalize .................................................................................................................... 104
    ◆ C_GetInfo .................................................................................................................... 105
    ◆ C_GetFunctionList ...................................................................................................... 106
  11.5 SLOT AND TOKEN MANAGEMENT FUNCTIONS ...................................................... 106
    ◆ C_GetSlotList ............................................................................................................. 106
    ◆ C_GetSlotInfo ............................................................................................................. 108
    ◆ C_GetTokenInfo .......................................................................................................... 109
    ◆ C_WaitForSlotEvent .................................................................................................... 110
    ◆ C_GetMechanismList .................................................................................................. 111
    ◆ C_GetMechanismInfo .................................................................................................. 112
    ◆ C_InitToken ................................................................................................................ 113
    ◆ C_InitPIN .................................................................................................................... 115
    ◆ C_SetPIN ..................................................................................................................... 116
# PKCS #11 V2.20: CRYPTOGRAPHIC TOKEN INTERFACE STANDARD

## 11.6 SESSION MANAGEMENT FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_OpenSession</td>
<td>117</td>
</tr>
<tr>
<td>C_CloseSession</td>
<td>118</td>
</tr>
<tr>
<td>C_CloseAllSessions</td>
<td>120</td>
</tr>
<tr>
<td>C_GetSessionInfo</td>
<td>120</td>
</tr>
<tr>
<td>C_GetOperationInfo</td>
<td>121</td>
</tr>
<tr>
<td>C_SetOperationState</td>
<td>123</td>
</tr>
<tr>
<td>C_Login</td>
<td>125</td>
</tr>
<tr>
<td>C_Logout</td>
<td>127</td>
</tr>
</tbody>
</table>

## 11.7 OBJECT MANAGEMENT FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CreateObject</td>
<td>128</td>
</tr>
<tr>
<td>C_CopyObject</td>
<td>130</td>
</tr>
<tr>
<td>C_DestroyObject</td>
<td>131</td>
</tr>
<tr>
<td>C_GetObjectSize</td>
<td>132</td>
</tr>
<tr>
<td>C_GetAttributeValue</td>
<td>133</td>
</tr>
<tr>
<td>C_SetAttributeValue</td>
<td>135</td>
</tr>
<tr>
<td>C_FindObjectsInit</td>
<td>136</td>
</tr>
<tr>
<td>C_FindObjects</td>
<td>137</td>
</tr>
<tr>
<td>C_FindObjectsFinal</td>
<td>138</td>
</tr>
</tbody>
</table>

## 11.8 ENCRYPTION FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_EncryptInit</td>
<td>139</td>
</tr>
<tr>
<td>C_Encrypt</td>
<td>139</td>
</tr>
<tr>
<td>C_EncryptUpdate</td>
<td>140</td>
</tr>
<tr>
<td>C_EncryptFinal</td>
<td>141</td>
</tr>
</tbody>
</table>

## 11.9 DECRYPTION FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_DecryptInit</td>
<td>144</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>144</td>
</tr>
<tr>
<td>C_DecryptUpdate</td>
<td>145</td>
</tr>
<tr>
<td>C_DecryptFinal</td>
<td>146</td>
</tr>
</tbody>
</table>

## 11.10 MESSAGE DIGESTING FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_DigestInit</td>
<td>148</td>
</tr>
<tr>
<td>C_Digest</td>
<td>149</td>
</tr>
<tr>
<td>C_DigestUpdate</td>
<td>150</td>
</tr>
<tr>
<td>C_DigestKey</td>
<td>150</td>
</tr>
<tr>
<td>C_DigestFinal</td>
<td>151</td>
</tr>
</tbody>
</table>

## 11.11 SIGNING AND MACING FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_SignInit</td>
<td>152</td>
</tr>
<tr>
<td>C_Sign</td>
<td>152</td>
</tr>
<tr>
<td>C_SignUpdate</td>
<td>153</td>
</tr>
<tr>
<td>C_SignFinal</td>
<td>154</td>
</tr>
<tr>
<td>C_SignRecoverInit</td>
<td>155</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>156</td>
</tr>
</tbody>
</table>

## 11.12 FUNCTIONS FOR VERIFYING SIGNATURES AND MACs

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_VerifyInit</td>
<td>157</td>
</tr>
<tr>
<td>C_Verify</td>
<td>158</td>
</tr>
<tr>
<td>C_VerifyUpdate</td>
<td>159</td>
</tr>
<tr>
<td>C_VerifyFinal</td>
<td>159</td>
</tr>
<tr>
<td>C_VerifyRecoverInit</td>
<td>161</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>161</td>
</tr>
</tbody>
</table>

## 11.13 DUAL-FUNCTION CRYPTOGRAPHIC FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_DigestEncryptUpdate</td>
<td>163</td>
</tr>
<tr>
<td>C_DecryptDigestUpdate</td>
<td>165</td>
</tr>
</tbody>
</table>
12 MECHANISMS ................................................................. 188

12.1 RSA ............................................................................. 193
  12.1.1 Definitions............................................................. 193
  12.1.2 RSA public key objects ........................................ 193
  12.1.3 RSA private key objects ........................................ 194
  12.1.4 PKCS #1 RSA key pair generation ...................... 196
  12.1.5 X9.31 RSA key pair generation ........................... 197
  12.1.6 PKCS #1 v1.5 RSA ............................................... 197
  12.1.7 PKCS #1 RSA OAEP mechanism parameters .......... 198
  ♦ CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR .............................. 198
  ♦ CK_RSA_PKCS_OAEP_SOURCE_TYPE; CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR...... 199
  ♦ CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR ...................... 200
  12.1.8 PKCS #1 RSA OAEP ............................................... 200
  12.1.9 PKCS #1 RSA PSS mechanism parameters ............. 201
  ♦ CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR ..................... 201
  12.1.10 PKCS #1 RSA PSS ............................................... 202
  12.1.11 ISO/IEC 9796 RSA .............................................. 203
  12.1.12 X.509 (raw) RSA ................................................ 203
  12.1.13 ANSI X9.31 RSA ................................................ 205
  12.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160 ................................................................. 206
  12.1.15 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512 ... 207
  12.1.16 ANSI X9.31 RSA signature with SHA-1 .......................... 208

12.2 DSA ............................................................................. 209
  12.2.1 Definitions............................................................. 209
  12.2.2 DSA public key objects ........................................ 209
  12.2.3 DSA private key objects ........................................ 210
  12.2.4 DSA domain parameter objects ........................... 211
  12.2.5 DSA key pair generation ...................................... 212
  12.2.6 DSA domain parameter generation ...................... 212
  12.2.7 DSA without hashing ........................................... 213
  12.2.8 DSA with SHA-1 .................................................. 213
  12.2.9 FORTEZZA timestamp ......................................... 214

12.3 ELLIPTIC CURVE ......................................................... 214
  12.3.1 EC Signatures ....................................................... 216
  12.3.2 Definitions .......................................................... 216
12.3.3 ECDSA public key objects ......................................................................................... 217
12.3.4 Elliptic curve private key objects ............................................................................. 218
12.3.5 Elliptic curve key pair generation ............................................................................. 219
12.3.6 ECDSA without hashing .......................................................................................... 220
12.3.7 ECDSA with SHA-1 ................................................................................................. 220
12.3.8 EC mechanism parameters ..................................................................................... 221
12.3.9 Elliptic curve Diffie-Hellman key derivation .............................................................. 224
12.3.10 Elliptic curve Diffie-Hellman with cofactor key derivation ...................................... 224
12.3.11 Elliptic curve Menezes-Qu-Vanstone key derivation ............................................... 225
12.4 DIFFIE-HELLMAN .................................................................................................... 226
12.4.1 Definitions .................................................................................................................. 226
12.4.2 Diffie-Hellman public key objects ........................................................................... 227
12.4.3 X9.42 Diffie-Hellman public key objects .................................................................. 228
12.4.4 Diffie-Hellman private key objects .......................................................................... 229
12.4.5 X9.42 Diffie-Hellman private key objects ............................................................... 230
12.4.6 Diffie-Hellman domain parameter objects .............................................................. 231
12.4.7 X9.42 Diffie-Hellman domain parameters objects ................................................... 232
12.4.8 PKCS #3 Diffie-Hellman key pair generation ............................................................ 233
12.4.9 PKCS #3 Diffie-Hellman domain parameter generation ........................................... 233
12.4.10 PKCS #3 Diffie-Hellman key derivation .................................................................. 234
12.4.11 X9.42 Diffie-Hellman mechanism parameters ....................................................... 235
  ♦ CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR ........ 235
  ♦ CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR ........ 236
  ♦ CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR .... 238
12.4.12 X9.42 Diffie-Hellman key pair generation .............................................................. 239
12.4.13 X9.42 Diffie-Hellman domain parameter generation ............................................. 239
12.4.14 X9.42 Diffie-Hellman key derivation .................................................................... 240
12.4.15 X9.42 Diffie-Hellman hybrid key derivation .......................................................... 241
12.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation .................................. 242
12.5 KEA ............................................................................................................................... 243
12.5.1 Definitions ................................................................................................................ 243
12.5.2 KEA mechanism parameters ................................................................................... 243
  ♦ CK KEA DERIVE_PARAMS, CK KEA DERIVE_PARAMS_PTR ...................... 243
12.5.3 KEA public key objects ............................................................................................. 244
12.5.4 KEA private key objects .......................................................................................... 244
12.5.5 KEA key pair generation ......................................................................................... 246
12.5.6 KEA key derivation .................................................................................................. 246
12.6 WRAPPING/UNWRAPPING PRIVATE KEYS ................................................................ 248
12.7 GENERIC SECRET KEY .......................................................................................... 251
12.7.1 Definitions .............................................................................................................. 251
12.7.2 Generic secret key objects ...................................................................................... 251
12.7.3 Generic secret key generation ................................................................................ 252
12.8 HMAC MECHANISMS ............................................................................................... 252
12.9 RC2 ............................................................................................................................... 253
12.9.1 Definitions .............................................................................................................. 253
12.9.2 RC2 secret key objects ............................................................................................ 253
12.9.3 RC2 mechanism parameters .................................................................................... 254
  ♦ CK_RC2_PARAMS, CK_RC2_PARAMS_PTR ...................................................... 254
  ♦ CK_RC2_CBC_PARAMS, CK_RC2_CBC_PARAMS_PTR ...................................... 254
  ♦ CK_RC2_MAC_GENERAL_PARAMS, CK_RC2_MAC_GENERAL_PARAMS_PTR .... 255
12.9.4 RC2 key generation ................................................................................................. 255
12.9.5 RC2-ECB ................................................................................................................ 255
12.9.6 RC2-CBC ............................................................................................................... 256
12.9.7 RC2-CBC with PKCS padding ................................................................................ 257
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.9.8 General-length RC2-MAC</td>
<td>258</td>
</tr>
<tr>
<td>12.9.9 RC2-MAC</td>
<td>259</td>
</tr>
<tr>
<td>12.10 RC4</td>
<td>259</td>
</tr>
<tr>
<td>12.10.1 Definitions</td>
<td>259</td>
</tr>
<tr>
<td>12.10.2 RC4 secret key objects</td>
<td>260</td>
</tr>
<tr>
<td>12.10.3 RC4 key generation</td>
<td>260</td>
</tr>
<tr>
<td>12.10.4 RC4 mechanism</td>
<td>261</td>
</tr>
<tr>
<td>12.11 RC5</td>
<td>261</td>
</tr>
<tr>
<td>12.11.1 Definitions</td>
<td>261</td>
</tr>
<tr>
<td>12.11.2 RC5 secret key objects</td>
<td>262</td>
</tr>
<tr>
<td>12.11.3 RC5 mechanism parameters</td>
<td>262</td>
</tr>
<tr>
<td>CK_RC5_PARAMS; CK_RC5_PARAMS_PTR</td>
<td>262</td>
</tr>
<tr>
<td>CK_RC5_CBC_PARAMS; CK_RC5_CBC_PARAMS_PTR</td>
<td>263</td>
</tr>
<tr>
<td>CK_RC5_MACGENERAL_PARAMS; CK_RC5_MACGENERAL_PARAMS_PTR</td>
<td>263</td>
</tr>
<tr>
<td>12.11.4 RC5 key generation</td>
<td>264</td>
</tr>
<tr>
<td>12.11.5 RC5-ECB</td>
<td>264</td>
</tr>
<tr>
<td>12.11.6 RC5-CBC</td>
<td>265</td>
</tr>
<tr>
<td>12.11.7 RC5-CBC with PKCS padding</td>
<td>266</td>
</tr>
<tr>
<td>12.11.8 General-length RC5-MAC</td>
<td>267</td>
</tr>
<tr>
<td>12.11.9 RC5-MAC</td>
<td>268</td>
</tr>
<tr>
<td>12.12 AES</td>
<td>268</td>
</tr>
<tr>
<td>12.12.1 Definitions</td>
<td>268</td>
</tr>
<tr>
<td>12.12.2 AES secret key objects</td>
<td>268</td>
</tr>
<tr>
<td>12.12.3 AES key generation</td>
<td>269</td>
</tr>
<tr>
<td>12.12.4 AES-ECB</td>
<td>270</td>
</tr>
<tr>
<td>12.12.5 AES-CBC</td>
<td>271</td>
</tr>
<tr>
<td>12.12.6 AES-CBC with PKCS padding</td>
<td>272</td>
</tr>
<tr>
<td>12.12.7 General-length AES-MAC</td>
<td>272</td>
</tr>
<tr>
<td>12.12.8 AES-MAC</td>
<td>273</td>
</tr>
<tr>
<td>12.13 GENERAL BLOCK CIPHER</td>
<td>274</td>
</tr>
<tr>
<td>12.13.1 Definitions</td>
<td>274</td>
</tr>
<tr>
<td>12.13.2 DES secret key objects</td>
<td>275</td>
</tr>
<tr>
<td>12.13.3 CAST secret key objects</td>
<td>276</td>
</tr>
<tr>
<td>12.13.4 CAST3 secret key objects</td>
<td>277</td>
</tr>
<tr>
<td>12.13.5 CAST128 (CAST5) secret key objects</td>
<td>277</td>
</tr>
<tr>
<td>12.13.6 IDEA secret key objects</td>
<td>278</td>
</tr>
<tr>
<td>12.13.7 CDMF secret key objects</td>
<td>278</td>
</tr>
<tr>
<td>12.13.8 General block cipher mechanism parameters</td>
<td>279</td>
</tr>
<tr>
<td>CK_MACGENERAL_PARAMS; CK_MACGENERAL_PARAMS_PTR</td>
<td>279</td>
</tr>
<tr>
<td>12.13.9 General block cipher key generation</td>
<td>280</td>
</tr>
<tr>
<td>12.13.10 General block cipher ECB</td>
<td>280</td>
</tr>
<tr>
<td>12.13.11 General block cipher CBC</td>
<td>281</td>
</tr>
<tr>
<td>12.13.12 General block cipher CBC with PKCS padding</td>
<td>282</td>
</tr>
<tr>
<td>12.13.13 General-length general block cipher MAC</td>
<td>283</td>
</tr>
<tr>
<td>12.13.14 General block cipher MAC</td>
<td>284</td>
</tr>
<tr>
<td>12.14 KEY DERIVATION BY DATA ENCRYPTION – DES &amp; AES</td>
<td>284</td>
</tr>
<tr>
<td>12.14.1 Definitions</td>
<td>285</td>
</tr>
<tr>
<td>12.14.2 Mechanism Parameters</td>
<td>285</td>
</tr>
<tr>
<td>12.14.3 Mechanism Description</td>
<td>286</td>
</tr>
<tr>
<td>12.15 DOUBLE AND TRIPLE-LENGTH DES</td>
<td>286</td>
</tr>
<tr>
<td>12.15.1 Definitions</td>
<td>286</td>
</tr>
<tr>
<td>12.15.2 DES2 secret key objects</td>
<td>287</td>
</tr>
<tr>
<td>12.15.3 DES3 secret key objects</td>
<td>288</td>
</tr>
<tr>
<td>12.15.4 Double-length DES key generation</td>
<td>288</td>
</tr>
<tr>
<td>12.15.5 Triple-length DES Order of Operations</td>
<td>289</td>
</tr>
</tbody>
</table>
12.16  SKIPJACK
12.16.1  Definitions ................................................................. 291
12.16.2  SKIPJACK secret key objects ....................................... 292
12.16.3  SKIPJACK Mechanism parameters ......................... 293
  * CK_SKIPJACK_PRIVATE_WRAP_PARAMS;
  CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR ........................................ 293
  * CK_SKIPJACK_RELAYX_PARAMS; CK_SKIPJACK_RELAYX_PARAMS_PTR .... 294
12.16.4  SKIPJACK key generation ........................................ 295
12.16.5  SKIPJACK-ECB64 ....................................................... 295
12.16.6  SKIPJACK-CBC64 ...................................................... 296
12.16.7  SKIPJACK-OFB64 ...................................................... 296
12.16.8  SKIPJACK-CFB64 ...................................................... 297
12.16.9  SKIPJACK-CFB32 ...................................................... 297
12.16.10 SKIPJACK-CFB16 ..................................................... 298
12.16.11 SKIPJACK-CFB8 ...................................................... 298
12.16.12 SKIPJACK-WRAP .................................................... 299
12.16.13 SKIPJACK-PRIVATE-WRAP .. .................................... 299
12.16.14 SKIPJACK-RELAYX .................................................. 299

12.17  BATON
12.17.1  Definitions ................................................................. 299
12.17.2  BATON secret key objects ......................................... 300
12.17.3  BATON key generation ............................................. 301
12.17.4  BATON-ECB128 ....................................................... 301
12.17.5  BATON-ECB96 ......................................................... 302
12.17.6  BATON-CBC128 ...................................................... 302
12.17.7  BATON-COUNTER .................................................. 303
12.17.8  BATON-SHUFFLE ..................................................... 303
12.17.9  BATON WRAP ......................................................... 304

12.18  JUNIPER ................................................................. 304
12.18.1  Definitions ................................................................. 304
12.18.2  JUNIPER secret key objects ....................................... 304
12.18.3  JUNIPER key generation ........................................... 305
12.18.4  JUNIPER-ECB128 ................................................... 305
12.18.5  JUNIPER-CBC128 ................................................... 306
12.18.6  JUNIPER-COUNTER ................................................ 306
12.18.7  JUNIPER-SHUFFLE ................................................ 307
12.18.8  JUNIPER WRAP ..................................................... 307

12.19  MD2 ................................................................. 308
12.19.1  Definitions ................................................................. 308
12.19.2  MD2 digest ............................................................... 308
12.19.3  General-length MD2-HMAC .................................... 308
12.19.4  MD2-HMAC ............................................................ 309
12.19.5  MD2 key derivation .................................................. 309

12.20  MD5 ................................................................. 310
12.20.1  Definitions ................................................................. 310
12.20.2  MD5 digest ............................................................... 310
12.20.3  General-length MD5-HMAC .................................... 311
12.20.4  MD5-HMAC ............................................................ 311
12.20.5  MD5 key derivation .................................................. 311

12.21  SHA-1 ................................................................. 312
12.21.1  Definitions ................................................................. 312
12.21.2  SHA-1 digest ............................................................ 313
12.21.3 General-length SHA-1-HMAC
12.21.4 SHA-1-HMAC
12.21.5 SHA-1 key derivation
12.22 SHA-256
12.22.1 Definitions
12.22.2 SHA-256 digest
12.22.3 General-length SHA-256-HMAC
12.22.4 SHA-256-HMAC
12.22.5 SHA-256 key derivation
12.23 SHA-384
12.23.1 Definitions
12.23.2 SHA-384 digest
12.23.3 General-length SHA-384-HMAC
12.23.4 SHA-384-HMAC
12.23.5 SHA-384 key derivation
12.24 SHA-512
12.24.1 Definitions
12.24.2 SHA-512 digest
12.24.3 General-length SHA-512-HMAC
12.24.4 SHA-512-HMAC
12.24.5 SHA-512 key derivation
12.25 FASTHASH
12.25.1 Definitions
12.25.2 FASTHASH digest
12.26 PKCS #5 AND PKCS #5-STYLE PASSWORD-BASED ENCRYPTION (PBE)
12.26.1 Definitions
12.26.2 Password-based encryption/authentication mechanism parameters
  ✔ CK_PBE_PARAMS; CK_PBE_PARAMS_PTR
12.26.3 MD2-PBE for DES-CBC
12.26.4 MD5-PBE for DES-CBC
12.26.5 MD5-PBE for CAST-CBC
12.26.6 MD5-PBE for CAST3-CBC
12.26.7 MD5-PBE for CAST128-CBC (CAST5-CBC)
12.26.8 SHA-1-PBE for CAST128-CBC (CAST5-CBC)
12.26.9 PKCS #5 PBKDF2 key generation mechanism parameters
  ✔ CK_PKCS5_PBKDF2_PSEUDO_RANDOM_FUNCTION_TYPE;
  CK_PKCS5_PBKDF2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR
  ✔ CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;
  CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR
  ✔ CK_PKCS5_PBKDF2_PARAMS; CK_PKCS5_PBKDF2_PARAMS_PTR
12.26.10 PKCS #5 PBKDF2 key generation
12.27 PKCS #12 PASSWORD-BASED ENCRYPTION/AUTHENTICATION MECHANISMS
12.27.1 SHA-1-PBE for 128-bit RC4
12.27.2 SHA-1-PBE for 40-bit RC4
12.27.3 SHA-1-PBE for 3-key triple-DES-CBC
12.27.4 SHA-1-PBE for 2-key triple-DES-CBC
12.27.5 SHA-1-PBE for 128-bit RC2-CBC
12.27.6 SHA-1-PBE for 40-bit RC2-CBC
12.27.7 SHA-1-PBA for SHA-1-HMAC
12.28 RIPE-MD
12.28.1 Definitions
12.28.2 RIPE-MD 128 digest
12.28.3 General-length RIPE-MD 128-HMAC
12.28.4 RIPE-MD 128-HMAC
12.28.5 RIPE-MD 160
12.28.6 General-length RIPE-MD 160-HMAC ................................................................. 330
12.28.7 RIPE-MD 160-HMAC ....................................................................................... 331
12.29 SET ..................................................................................................................... 331
12.29.1 Definitions ........................................................................................................ 331
12.29.2 SET mechanism parameters .......................................................................... 331
   • CK_KEY_WRAP_SET_OAEP_PARAMS; CK_KEY_WRAP_SET_OAEP_PARAMS_PTR 331
12.29.3 OAEP key wrapping for SET ........................................................................... 332
12.30 LYNKS ................................................................................................................ 332
12.30.1 Definitions ........................................................................................................ 332
12.30.2 LYNKS key wrapping ....................................................................................... 333
12.31 SSL ...................................................................................................................... 333
12.31.1 Definitions ........................................................................................................ 333
12.31.2 SSL mechanism parameters .......................................................................... 334
   • CK_SSL3_RANDOM_DATA......................................................................................... 334
   • CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
     CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR .................................................. 334
   • CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR ...................................... 335
   • CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR ..................... 335
12.31.3 Pre_master key generation .............................................................................. 336
12.31.4 Master key derivation ...................................................................................... 337
12.31.5 Master key derivation for Diffie-Hellman ...................................................... 338
12.31.6 Key and MAC derivation ................................................................................ 339
12.31.7 MD5 MACing in SSL 3.0 ................................................................................ 340
12.31.8 SHA-1 MACing in SSL 3.0 ............................................................................. 341
12.32 TLS ....................................................................................................................... 341
12.32.1 Definitions ........................................................................................................ 341
12.32.2 TLS mechanism parameters .......................................................................... 342
   • CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR ............................................... 342
12.32.3 TLS PRF (pseudorandom function) ................................................................ 342
12.32.4 Pre_master key generation .............................................................................. 343
12.32.5 Master key derivation ...................................................................................... 343
12.32.6 Master key derivation for Diffie-Hellman ...................................................... 344
12.32.7 Key and MAC derivation ................................................................................ 345
12.33 WTLS .................................................................................................................. 347
12.33.1 Definitions ........................................................................................................ 347
12.33.2 WTLS mechanism parameters ....................................................................... 347
   • CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR .......................... 347
   • CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
     CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR ............................................ 348
   • CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR .............. 349
12.33.3 Pre master secret key generation for RSA key exchange suite ...................... 351
12.33.4 Master secret key derivation .......................................................................... 351
12.33.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography ..................................................................................................................................................... 352
12.33.6 WTLS PRF (pseudorandom function) .............................................................. 353
12.33.7 Server Key and MAC derivation ..................................................................... 354
12.33.8 Client key and MAC derivation ..................................................................... 355
12.34 MISCELLANEOUS SIMPLE KEY DERIVATION MECHANISMS ....................... 356
12.34.1 Definitions ........................................................................................................ 356
12.34.2 Parameters for miscellaneous simple key derivation mechanisms ............... 357
   • CK_KEY Derivation STRING_DATA; CK_KEY Derivation STRING_DATA_PTR 357
   • CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR ...................................... 357
12.34.3 Concatenation of a base key and another key .............................................. 357

Copyright © 2004 RSA Security Inc. June 2004
12.34.4 Concatenation of a base key and data ................................................................. 359
12.34.5 Concatenation of data and a base key ................................................................. 360
12.34.6 XORing of a key and data .................................................................................. 361
12.34.7 Extraction of one key from another key ............................................................ 362
12.35 CMS ....................................................................................................................... 364
  12.35.1 Definitions ....................................................................................................... 364
  12.35.2 CMS Signature Mechanism Objects ............................................................... 364
  12.35.3 CMS mechanism parameters ......................................................................... 365
• CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR ............................................... 365
  12.35.4 CMS signatures .............................................................................................. 366
12.36 BLOWFISH ............................................................................................................ 367
  12.36.1 Definitions ....................................................................................................... 368
  12.36.2 BLOWFISH secret key objects ....................................................................... 368
  12.36.3 Blowfish key generation ................................................................................ 369
  12.36.4 Blowfish -CBC ............................................................................................. 369
12.37 TWOFISH ............................................................................................................. 369
  12.37.1 Definitions ....................................................................................................... 370
  12.37.2 Twofish secret key objects ............................................................................. 370
  12.37.3 Twofish key generation ................................................................................ 370
  12.37.4 Twofish -CBC ............................................................................................. 371
13 CRYPTOKI TIPS AND REMINDERS ......................................................................... 371
  13.1 OPERATIONS, SESSIONS, AND THREADS ......................................................... 371
  13.2 MULTIPLE APPLICATION ACCESS BEHAVIOR ............................................... 372
  13.3 OBJECTS, ATTRIBUTES, AND TEMPLATES .................................................. 372
  13.4 SIGNING WITH RECOVERY ............................................................................... 373
A MANIFEST CONSTANTS ................................................................................................. 375
B TOKEN PROFILES ......................................................................................................... 382
  B.1 GOVERNMENT AUTHENTICATION-ONLY ......................................................... 383
  B.2 CELLULAR DIGITAL PACKET DATA ................................................................. 383
  B.3 OTHER PROFILES ............................................................................................. 384
C COMPARISON OF CRYPTOKI AND OTHER APIS ................................................ 385
  C.1 FORTEZZA CIPG, REV. 1.52 ............................................................................. 385
  C.2 GCS-API ............................................................................................................ 387
D INTELLECTUAL PROPERTY CONSIDERATIONS ......................................................... 389
E METHOD FOR EXPOSING MULTIPLE-PINS ON A TOKEN THROUGH CRYPTOKI (DEPRECATED) ................................................................................................. 390
F REVISION HISTORY .................................................................................................... 391

List of Figures

FIGURE 1, GENERAL CRYPTOKI MODEL ...................................................................... 14
FIGURE 2, OBJECT HIERARCHY ................................................................................... 15
FIGURE 3, READ-ONLY SESSION STATES ................................................................. 20
FIGURE 4, READ/WRITE SESSION STATES ............................................................... 21
FIGURE 5, OBJECT ATTRIBUTE HIERARCHY ............................................................ 62

List of Tables
TABLE 1, SYMBOLS .............................................................................................................10
TABLE 2, PREFIXES .............................................................................................................10
TABLE 3, CHARACTER SET .................................................................................................12
TABLE 4, READ-ONLY SESSION STATES .............................................................................20
TABLE 5, READ/WRITE SESSION STATES ............................................................................21
TABLE 6, ACCESS TO DIFFERENT TYPES OBJECTS BY DIFFERENT TYPES OF SESSIONS .....22
TABLE 7, SESSION EVENTS .................................................................................................22
TABLE 8, SUMMARY OF CRYPTOKI FUNCTIONS ..................................................................27
TABLE 9, MAJOR AND MINOR VERSION VALUES FOR PUBLISHED CRYPTOKI SPECIFICATIONS37
TABLE 10, SLOT INFORMATION FLAGS ...............................................................................39
TABLE 11, TOKEN INFORMATION FLAGS............................................................................42
TABLE 12, SESSION INFORMATION FLAGS ........................................................................48
TABLE 13, MECHANISM INFORMATION FLAGS ...................................................................54
TABLE 14, C_INITIALIZE PARAMETER FLAGS .....................................................................61
TABLE 15, COMMON FOOTNOTES FOR OBJECT ATTRIBUTE TABLES .....................................66
TABLE 16, COMMON OBJECT ATTRIBUTES .........................................................................67
TABLE 17, HARDWARE FEATURE COMMON ATTRIBUTES ...................................................67
TABLE 18, CLOCK OBJECT ATTRIBUTES .............................................................................68
TABLE 19, MONOTONIC COUNTER ATTRIBUTES ...................................................................69
TABLE 20, USER INTERFACE OBJECT ATTRIBUTES ............................................................70
TABLE 21, COMMON STORAGE OBJECT ATTRIBUTES .........................................................71
TABLE 22, DATA OBJECT ATTRIBUTES ..............................................................................72
TABLE 23, COMMON CERTIFICATE OBJECT ATTRIBUTES ..................................................73
TABLE 24, X.509 CERTIFICATE OBJECT ATTRIBUTES ......................................................75
TABLE 25, WTLS CERTIFICATE OBJECT ATTRIBUTES .......................................................77
TABLE 26, X.509 ATTRIBUTE CERTIFICATE OBJECT ATTRIBUTES ....................................78
TABLE 27, COMMON KEY ATTRIBUTES ..............................................................................79
TABLE 28, COMMON PUBLIC KEY ATTRIBUTES ...............................................................81
TABLE 29, TABLE 29, MAPPING OF X.509 KEY USAGE FLAGS TO CRYPTOKI ATTRIBUTES FOR PUBLIC KEYS .........................................................................................82
TABLE 30, COMMON PRIVATE KEY ATTRIBUTES ...............................................................82
TABLE 31, COMMON SECRET KEY ATTRIBUTES .................................................................85
TABLE 32, COMMON DOMAIN PARAMETER ATTRIBUTES ...................................................88
TABLE 33, COMMON MECHANISM ATTRIBUTES .................................................................88
TABLE 34, MECHANISMS VS. FUNCTIONS .............................................................................188
TABLE 35, RSA PUBLIC KEY OBJECT ATTRIBUTES ..........................................................193
TABLE 36, RSA PRIVATE KEY OBJECT ATTRIBUTES ........................................................194
TABLE 37, PKCS #1 v1.5 RSA: KEY AND DATA LENGTH ..................................................198
TABLE 38, PKCS #1 MASK GENERATION FUNCTIONS .......................................................199
TABLE 39, PKCS #1 RSA OAEP: ENCODING PARAMETER SOURCES .................................199
TABLE 40, PKCS #1 RSA OAEP: KEY AND DATA LENGTH ..............................................201
TABLE 41, PKCS #1 RSA PSS: KEY AND DATA LENGTH ..................................................201
TABLE 42, ISO/IEC 9796 RSA: KEY AND DATA LENGTH ................................................203
TABLE 43, X.509 (RAW) RSA: KEY AND DATA LENGTH ................................................205
TABLE 44, ANSI X9.31 RSA: KEY AND DATA LENGTH .....................................................206
TABLE 45, PKCS #1 v1.5 RSA SIGNATURES WITH VARIOUS HASH FUNCTIONS: KEY AND DATA LENGTH............................................................................................................207
TABLE 46, PKCS #1 RSA PSS SIGNATURES WITH VARIOUS HASH FUNCTIONS: KEY AND DATA LENGTH............................................................................................................208
TABLE 47, ANSI X9.31 RSA SIGNATURES WITH SHA-1: KEY AND DATA LENGTH .................................................................................................................................208
TABLE 48, DSA Public Key Object Attributes .................................................................................................................................209
TABLE 49, DSA Private Key Object Attributes .................................................................................................................................210
TABLE 50, DSA Domain Parameter Object Attributes .................................................................................................................................211
TABLE 51, DSA: KEY AND DATA LENGTH .................................................................................................................................................................213
TABLE 52, DSA with SHA-1: KEY AND DATA LENGTH .................................................................................................................................................................214
TABLE 53, FORTEZZA Timestamp: KEY AND DATA LENGTH .................................................................................................................................................................214
TABLE 54, Mechanism Information Flags .................................................................................................................................................................214
TABLE 55, Elliptic Curve Public Key Object Attributes .................................................................................................................................217
TABLE 56, Elliptic Curve Private Key Object Attributes .................................................................................................................................218
TABLE 57, ECDSA: KEY AND DATA LENGTH .................................................................................................................................................................220
TABLE 58, ECDSA with SHA-1: KEY AND DATA LENGTH .................................................................................................................................................................221
TABLE 59, EC: Key Derivation Functions .................................................................................................................................................................221
TABLE 60, Diffie-Hellman Public Key Object Attributes .................................................................................................................................................................227
TABLE 61, X9.42 Diffie-Hellman Public Key Object Attributes .................................................................................................................................................................228
TABLE 62, Diffie-Hellman Private Key Object Attributes .................................................................................................................................................................229
TABLE 63, X9.42 Diffie-Hellman Private Key Object Attributes .................................................................................................................................................................230
TABLE 64, Diffie-Hellman Domain Parameter Object Attributes .................................................................................................................................................................231
TABLE 65, X9.42 Diffie-Hellman Domain Parameters Object Attributes .................................................................................................................................................................232
TABLE 66, X9.42 Diffie-Hellman Key Derivation Functions .................................................................................................................................................................235
TABLE 67, KEA Public Key Object Attributes .................................................................................................................................................................244
TABLE 68, KEA Private Key Object Attributes .................................................................................................................................................................245
TABLE 69, KEA Parameter Values and Operations .................................................................................................................................................................247
TABLE 70, Generic Secret Key Object Attributes .................................................................................................................................................................251
TABLE 71, RC2 Secret Key Object Attributes .................................................................................................................................................................253
TABLE 72, RC2-ECB: KEY AND DATA LENGTH .................................................................................................................................................................256
TABLE 73, RC2-CBC: KEY AND DATA LENGTH .................................................................................................................................................................257
TABLE 74, RC2-CBC with PKCS Padding: KEY AND DATA LENGTH .................................................................................................................................................................258
TABLE 75, General-Length RC2-MAC: KEY AND DATA LENGTH .................................................................................................................................................................259
TABLE 76, RC2-MAC: KEY AND DATA LENGTH .................................................................................................................................................................259
TABLE 77, RC4 Secret Key Object .................................................................................................................................................................260
TABLE 78, RC4: KEY AND DATA LENGTH .................................................................................................................................................................261
TABLE 79, RC5 Secret Key Object .................................................................................................................................................................262
TABLE 80, RC5-ECB: KEY AND DATA LENGTH .................................................................................................................................................................265
TABLE 81, RC5-CBC: KEY AND DATA LENGTH .................................................................................................................................................................266
TABLE 82, RC5-CBC with PKCS Padding: KEY AND DATA LENGTH .................................................................................................................................................................267
TABLE 83, General-Length RC2-MAC: KEY AND DATA LENGTH .................................................................................................................................................................267
TABLE 84, RC5-MAC: KEY AND DATA LENGTH .................................................................................................................................................................268
TABLE 85, AES Secret Key Object Attributes .................................................................................................................................................................269
TABLE 86, AES-ECB: KEY AND DATA LENGTH .................................................................................................................................................................270
TABLE 87, AES-CBC: KEY AND DATA LENGTH .................................................................................................................................................................271
TABLE 88, AES-CBC with PKCS Padding: KEY AND DATA LENGTH .................................................................................................................................................................272
<table>
<thead>
<tr>
<th>Table Numbers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>General-length AES-MAC: Key and Data Length</td>
</tr>
<tr>
<td>90</td>
<td>AES-MAC: Key and Data Length</td>
</tr>
<tr>
<td>91</td>
<td>DES secret key object</td>
</tr>
<tr>
<td>92</td>
<td>CAST secret key object attributes</td>
</tr>
<tr>
<td>93</td>
<td>CAST3 secret key object attributes</td>
</tr>
<tr>
<td>94</td>
<td>CAST128 (CAST5) secret key object attributes</td>
</tr>
<tr>
<td>95</td>
<td>IDEA secret key object</td>
</tr>
<tr>
<td>96</td>
<td>CDMF secret key object</td>
</tr>
<tr>
<td>97</td>
<td>General block cipher ECB: Key and Data Length</td>
</tr>
<tr>
<td>98</td>
<td>General block cipher CBC: Key and Data Length</td>
</tr>
<tr>
<td>99</td>
<td>General block cipher CBC with PKCS padding: Key and Data Length</td>
</tr>
<tr>
<td>100</td>
<td>General-length general block cipher MAC: Key and Data Length</td>
</tr>
<tr>
<td>101</td>
<td>General block cipher MAC: Key and Data Length</td>
</tr>
<tr>
<td>102</td>
<td>Mechanism parameters</td>
</tr>
<tr>
<td>103</td>
<td>DES2 secret key object attributes</td>
</tr>
<tr>
<td>104</td>
<td>DES3 secret key object attributes</td>
</tr>
<tr>
<td>105</td>
<td>OFB: Key and Data Length</td>
</tr>
<tr>
<td>106</td>
<td>CFB: Key and Data Length</td>
</tr>
<tr>
<td>107</td>
<td>Skipjack secret key object</td>
</tr>
<tr>
<td>108</td>
<td>Skipjack-ECB64: Data and Length</td>
</tr>
<tr>
<td>109</td>
<td>Skipjack-CBC64: Data and Length</td>
</tr>
<tr>
<td>110</td>
<td>Skipjack-OFB64: Data and Length</td>
</tr>
<tr>
<td>111</td>
<td>Skipjack-CFB64: Data and Length</td>
</tr>
<tr>
<td>112</td>
<td>Skipjack-CFB32: Data and Length</td>
</tr>
<tr>
<td>113</td>
<td>Skipjack-CFB16: Data and Length</td>
</tr>
<tr>
<td>114</td>
<td>Skipjack-CFB8: Data and Length</td>
</tr>
<tr>
<td>115</td>
<td>Baton secret key object</td>
</tr>
<tr>
<td>116</td>
<td>Baton-ECB128: Data and Length</td>
</tr>
<tr>
<td>117</td>
<td>Baton-ECB96: Data and Length</td>
</tr>
<tr>
<td>118</td>
<td>Baton-CBC128: Data and Length</td>
</tr>
<tr>
<td>119</td>
<td>Baton-counter: Data and Length</td>
</tr>
<tr>
<td>120</td>
<td>Baton-shuffle: Data and Length</td>
</tr>
<tr>
<td>121</td>
<td>Juniper secret key object</td>
</tr>
<tr>
<td>122</td>
<td>Juniper-ECB128: Data and Length</td>
</tr>
<tr>
<td>123</td>
<td>Juniper-CBC128: Data and Length</td>
</tr>
<tr>
<td>124</td>
<td>Juniper-counter: Data and Length</td>
</tr>
<tr>
<td>125</td>
<td>Juniper-shuffle: Data and Length</td>
</tr>
<tr>
<td>126</td>
<td>MD2: Data Length</td>
</tr>
<tr>
<td>127</td>
<td>General-length MD2-HMAC: Key and Data Length</td>
</tr>
<tr>
<td>128</td>
<td>MD5: Data Length</td>
</tr>
<tr>
<td>129</td>
<td>General-length MD5-HMAC: Key and Data Length</td>
</tr>
<tr>
<td>130</td>
<td>SHA-1: Data Length</td>
</tr>
<tr>
<td>131</td>
<td>General-length SHA-1-HMAC: Key and Data Length</td>
</tr>
<tr>
<td>132</td>
<td>SHA-256: Data Length</td>
</tr>
<tr>
<td>133</td>
<td>General-length SHA-256-HMAC: Key and Data Length</td>
</tr>
<tr>
<td>134</td>
<td>SHA-384: Data Length</td>
</tr>
</tbody>
</table>
TABLE 135, SHA-512: DATA LENGTH ................................................................. 318
TABLE 136, FASTHASH: DATA LENGTH ......................................................... 319
TABLE 137, PKCS #5 PBKDF2 KEY GENERATION: PSEUDO-RANDOM FUNCTIONS 323
TABLE 138, PKCS #5 PBKDF2 KEY GENERATION: SALT SOURCES .............. 323
TABLE 139, RIPE-MD 128: DATA LENGTH ...................................................... 329
TABLE 140, GENERAL-LENGTH RIPE-MD 128-HMAC: ................................. 330
TABLE 141, RIPE-MD 160: DATA LENGTH ...................................................... 330
TABLE 142, GENERAL-LENGTH RIPE-MD 160-HMAC: ................................. 331
TABLE 143, MD5 MACING IN SSL 3.0: KEY AND DATA LENGTH .................. 340
TABLE 144, SHA-1 MACING IN SSL 3.0: KEY AND DATA LENGTH .............. 341
TABLE 145, CMS SIGNATURE MECHANISM OBJECT ATTRIBUTES ............ 364
TABLE 146, BLOWFISH SECRET KEY OBJECT ............................................ 368
TABLE 147, TWOFISH SECRET KEY OBJECT .............................................. 370
1 Introduction

As cryptography begins to see wide application and acceptance, one thing is increasingly clear: if it is going to be as effective as the underlying technology allows it to be, there must be interoperable standards. Even though vendors may agree on the basic cryptographic techniques, compatibility between implementations is by no means guaranteed. Interoperability requires strict adherence to agreed-upon standards.

Towards that goal, RSA Laboratories has developed, in cooperation with representatives of industry, academia and government, a family of standards called Public-Key Cryptography Standards, or PKCS for short.

PKCS is offered by RSA Laboratories to developers of computer systems employing public-key and related technology. It is RSA Laboratories' intention to improve and refine the standards in conjunction with computer system developers, with the goal of producing standards that most if not all developers adopt.

The role of RSA Laboratories in the standards-making process is four-fold:

1. Publish carefully written documents describing the standards.
2. Solicit opinions and advice from developers and users on useful or necessary changes and extensions.
3. Publish revised standards when appropriate.
4. Provide implementation guides and/or reference implementations.

During the process of PKCS development, RSA Laboratories retains final authority on each document, though input from reviewers is clearly influential. However, RSA Laboratories' goal is to accelerate the development of formal standards, not to compete with such work. Thus, when a PKCS document is accepted as a base document for a formal standard, RSA Laboratories relinquishes its “ownership” of the document, giving way to the open standards development process. RSA Laboratories may continue to develop related documents, of course, under the terms described above.

PKCS documents and information are available online at http://www.rsasecurity.com/rsalabs/PKCS/. There is an electronic mailing list, “cryptoki”, at rsasecurity.com, specifically for discussion and development of PKCS #11. To subscribe to this list, send e-mail to majordomo@rsasecurity.com with the line “subscribe cryptoki” in the message body. To unsubscribe, send e-mail to majordomo@rsasecurity.com with the line “unsubscribe cryptoki” in the message body.

Comments on the PKCS documents, requests to register extensions to the standards, and suggestions for additional standards are welcomed. Address correspondence to:
It would be difficult to enumerate all the people and organizations who helped to produce PKCS #11. RSA Laboratories is grateful to each and every one of them. Special thanks go to Bruno Couillard of Chrysalis-ITS and John Centafont of NSA for the many hours they spent writing up parts of this document. Thanks also for the many other technical descriptions provided by many industry specialists. The reviewers of the document, without whose help the quality of the content would not be as great, must also be acknowledged and thanked. The review effort cannot be underestimated especially for a document so large.

For Version 1.0, PKCS #11’s document editor was Aram Pérez of International Computer Services, under contract to RSA Laboratories; the project coordinator was Burt Kaliski of RSA Laboratories. For Version 2.01, Ray Sidney served as document editor and project coordinator. Matthew Wood of Intel was document editor and project coordinator for Version 2.10 and Version 2.11. Simon McMahon from Eracom was editor for Version 2.20 while Magnus Nystrom of RSA coordinated the project.

2 Scope

This standard specifies an application programming interface (API), called “Cryptoki,” to devices which hold cryptographic information and perform cryptographic functions. Cryptoki, pronounced “crypto-key” and short for “cryptographic token interface,” follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

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

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

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

A number of cryptographic mechanisms (algorithms) are supported in this version. In addition, new mechanisms can be added later without changing the general interface. It is possible that additional mechanisms will be published from time to time in separate documents; it is also possible for token vendors to define their own mechanisms (although, for the sake of interoperability, registration through the PKCS process is preferable).

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

3 References

3.1  ANSI C


3.2  ANSI X9.31


3.3  ANSI X9.42


3.4  ANSI X9.62


3.5  ANSI X9.63


3.6  CC/PP


3.7  CDPD


3.8  FIPS PUB 46–3

<table>
<thead>
<tr>
<th>Standard</th>
<th>Title</th>
<th>Date</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCS-API</td>
<td>X/Open Company Ltd. <em>Generic Cryptographic Service API (GCS-API), Base - Draft 2.</em></td>
<td>February 14, 1995</td>
<td></td>
</tr>
<tr>
<td>ISO/IEC 7816-4</td>
<td>ISO. <em>Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 4: Interindustry Commands for Interchange.</em></td>
<td>1995</td>
<td></td>
</tr>
<tr>
<td>ISO/IEC 8825-1</td>
<td>ISO. <em>Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).</em></td>
<td>2002</td>
<td></td>
</tr>
</tbody>
</table>
3. References

PCMCIA

PKCS #1

PKCS #3

PKCS #5

PKCS #7

PKCS #8

PKCS #11-C

PKCS #11-P

PKCS #12

RFC 1319

RFC 1321

RFC 1421

RFC 2045

RFC 2246

RFC 2279


4 Definitions

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

API  Application programming interface.
Application Any computer program that calls the Cryptoki interface.
ASN.1  Abstract Syntax Notation One, as defined in X.680.
Attribute A characteristic of an object.
BATON  MISSI’s BATON block cipher.
BER  Basic Encoding Rules, as defined in X.690.
CAST  Entrust Technologies’ proprietary symmetric block cipher.
CAST3  Entrust Technologies’ proprietary symmetric block cipher.
CAST5  Another name for Entrust Technologies’ symmetric block cipher CAST128. CAST128 is the preferred name.
CAST128  Entrust Technologies’ symmetric block cipher.
CBC  Cipher-Block Chaining mode, as defined in FIPS PUB 81.
CDMF  Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
Certificate A signed message binding a subject name and a public key, or a subject name and a set of attributes.
CMS  Cryptographic Message Syntax (see RFC 2630)
Cryptographic Device  A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.
Cryptoki  The Cryptographic Token Interface defined in this standard.
Cryptoki library  A library that implements the functions specified in this standard.
DER  Distinguished Encoding Rules, as defined in X.690.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>Data Encryption Standard, as defined in FIPS PUB 46-3.</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm, as defined in FIPS PUB 186-2.</td>
</tr>
<tr>
<td>EC</td>
<td>Elliptic Curve</td>
</tr>
<tr>
<td>ECB</td>
<td>Electronic Codebook mode, as defined in FIPS PUB 81.</td>
</tr>
<tr>
<td>ECDH</td>
<td>Elliptic Curve Diffie-Hellman.</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curve DSA, as in ANSI X9.62.</td>
</tr>
<tr>
<td>ECMQV</td>
<td>Elliptic Curve Menezes-Qu-Vanstone</td>
</tr>
<tr>
<td>FASTHASH</td>
<td>MISSI’s FASTHASH message-digesting algorithm.</td>
</tr>
<tr>
<td>IDEA</td>
<td>Ascom Systec’s symmetric block cipher.</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization Vector.</td>
</tr>
<tr>
<td>JUNIPER</td>
<td>MISSI’s JUNIPER block cipher.</td>
</tr>
<tr>
<td>KEA</td>
<td>MISSI’s Key Exchange Algorithm.</td>
</tr>
<tr>
<td>LYNKS</td>
<td>A smart card manufactured by SPYRUS.</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code.</td>
</tr>
<tr>
<td>MD2</td>
<td>RSA Security's MD2 message-digest algorithm, as defined in RFC 1319.</td>
</tr>
<tr>
<td>MD5</td>
<td>RSA Security's MD5 message-digest algorithm, as defined in RFC 1321.</td>
</tr>
<tr>
<td>Mechanism</td>
<td>A process for implementing a cryptographic operation.</td>
</tr>
<tr>
<td>MQV</td>
<td>Menezes-Qu-Vanstone</td>
</tr>
<tr>
<td>OAEP</td>
<td>Optimal Asymmetric Encryption Padding for RSA.</td>
</tr>
<tr>
<td>Object</td>
<td>An item that is stored on a token. May be data, a certificate, or a key.</td>
</tr>
<tr>
<td>PIN</td>
<td>Personal Identification Number.</td>
</tr>
<tr>
<td>PKCS</td>
<td>Public-Key Cryptography Standards.</td>
</tr>
<tr>
<td>PRF</td>
<td>Pseudo random function.</td>
</tr>
<tr>
<td>PTD</td>
<td>Personal Trusted Device, as defined in MeT-PTD</td>
</tr>
<tr>
<td>RSA</td>
<td>The RSA public-key cryptosystem.</td>
</tr>
<tr>
<td>RC2</td>
<td>RSA Security’s RC2 symmetric block cipher.</td>
</tr>
<tr>
<td>RC4</td>
<td>RSA Security’s proprietary RC4 symmetric stream cipher.</td>
</tr>
</tbody>
</table>
4. Definitions

- **RC5**: RSA Security’s RC5 symmetric block cipher.
- **Reader**: The means by which information is exchanged with a device.
- **Session**: A logical connection between an application and a token.
- **SET**: The Secure Electronic Transaction protocol.
- **SHA-1**: The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.
- **SHA-256**: 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.
- **SKIPJACK**: MISSI’s SKIPJACK block cipher.
- **SSL**: The Secure Sockets Layer 3.0 protocol.
- **Subject Name**: The X.500 distinguished name of the entity to which a key is assigned.
- **SO**: A Security Officer user.
- **TLS**: Transport Layer Security.
- **Token**: The logical view of a cryptographic device defined by Cryptoki.
- **User**: The person using an application that interfaces to Cryptoki.
- **UTF-8**: Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.
- **WIM**: Wireless Identification Module.
5 Symbols and abbreviations

The following symbols are used in this standard:

Table 1, Symbols

<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 2, Prefixes

<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>
<tr>
<td>CKP_</td>
<td>Pseudo-random function</td>
</tr>
<tr>
<td>CKS_</td>
<td>Session state</td>
</tr>
<tr>
<td>CKR_</td>
<td>Return value</td>
</tr>
<tr>
<td>CKU_</td>
<td>User type</td>
</tr>
<tr>
<td>CKZ_</td>
<td>Salt/Encoding parameter source</td>
</tr>
<tr>
<td>h</td>
<td>a handle</td>
</tr>
<tr>
<td>ul</td>
<td>a CK_ULONG</td>
</tr>
<tr>
<td>p</td>
<td>a pointer</td>
</tr>
<tr>
<td>pb</td>
<td>a pointer to a CK_BYTE</td>
</tr>
<tr>
<td>ph</td>
<td>a pointer to a handle</td>
</tr>
<tr>
<td>pul</td>
<td>a pointer to a CK_ULONG</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>! “ # % &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 defines TRUE and FALSE as (**CK_DISABLE_TRUE_FALSE** may be set by the application vendor):

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

### 6 General overview

#### 6.1 Introduction

Portable computing devices such as smart cards, PCMCIA cards, and smart diskettes are ideal tools for implementing public-key cryptography, as they provide a way to store the
private-key component of a public-key/private-key pair securely, under the control of a single user. With such a device, a cryptographic application, rather than performing cryptographic operations itself, utilizes the device to perform the operations, with sensitive information such as private keys never being revealed. As more applications are developed for public-key cryptography, a standard programming interface for these devices becomes increasingly valuable. This standard addresses this need.

6.2 Design goals

Cryptoki was intended from the beginning to be an interface between applications and all kinds of portable cryptographic devices, such as those based on smart cards, PCMCIA cards, and smart diskettes. There are already standards (de facto or official) for interfacing to these devices at some level. For instance, the mechanical characteristics and electrical connections are well-defined, as are the methods for supplying commands and receiving results. (See, for example, ISO 7816, or the PCMCIA specifications.)

What remained to be defined were particular commands for performing cryptography. It would not be enough simply to define command sets for each kind of device, as that would not solve the general problem of an application interface independent of the device. To do so is still a long-term goal, and would certainly contribute to interoperability. The primary goal of Cryptoki was a lower-level programming interface that abstracts the details of the devices, and presents to the application a common model of the cryptographic device, called a “cryptographic token” (or simply “token”).

A secondary goal was resource-sharing. As desktop multi-tasking operating systems become more popular, a single device should be shared between more than one application. In addition, an application should be able to interface to more than one device at a given time.

It is not the goal of Cryptoki to be a generic interface to cryptographic operations or security services, although one certainly could build such operations and services with the functions that Cryptoki provides. Cryptoki is intended to complement, not compete with, such emerging and evolving interfaces as “Generic Security Services Application Programming Interface” (RFC 2743 and RFC 2744) and “Generic Cryptographic Service API” (GCS-API) from X/Open.

6.3 General model

Cryptoki's general model is illustrated in the following figure. The model begins with one or more applications that need to perform certain cryptographic operations, and ends with one or more cryptographic devices, on which some or all of the operations are actually performed. A user may or may not be associated with an application.
Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of “slots”. Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is typically “present in the slot” when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots, and applications can connect to tokens in any or all of those slots.

A cryptographic device can perform some cryptographic operations, following a certain command set; these commands are typically passed through standard device drivers, for instance PCMCIA card services or socket services. Cryptoki makes each cryptographic device look logically like every other device, regardless of the implementation technology. Thus the application need not interface directly to the device drivers (or even know which ones are involved); Cryptoki hides these details. Indeed, the underlying “device” may be implemented entirely in software (for instance, as a process running on a server)—no special hardware is necessary.

Cryptoki is likely to be implemented as a library supporting the functions in the interface, and applications will be linked to the library. An application may be linked to Cryptoki directly; alternatively, Cryptoki can be a so-called “shared” library (or dynamic link
library), in which case the application would link the library dynamically. Shared libraries are fairly straightforward to produce in operating systems such as Microsoft Windows and OS/2, and can be achieved without too much difficulty in UNIX and DOS systems.

The dynamic approach certainly has advantages as new libraries are made available, but from a security perspective, there are some drawbacks. In particular, if a library is easily replaced, then there is the possibility that an attacker can substitute a rogue library that intercepts a user’s PIN. From a security perspective, therefore, direct linking is generally preferable, although code-signing techniques can prevent many of the security risks of dynamic linking. In any case, whether the linking is direct or dynamic, the programming interface between the application and a Cryptoki library remains the same.

The kinds of devices and capabilities supported will depend on the particular Cryptoki library. This standard specifies only the interface to the library, not its features. In particular, not all libraries will support all the mechanisms (algorithms) defined in this interface (since not all tokens are expected to support all the mechanisms), and libraries will likely support only a subset of all the kinds of cryptographic devices that are available. (The more kinds, the better, of course, and it is anticipated that libraries will be developed supporting multiple kinds of token, rather than just those from a single vendor.) It is expected that as applications are developed that interface to Cryptoki, standard library and token “profiles” will emerge.

6.4 Logical view of a token

Cryptoki’s logical view of a token is a device that stores objects and can perform cryptographic functions. Cryptoki defines three classes of object: data, certificates, and keys. A data object is defined by an application. A certificate object stores a certificate. A key object stores a cryptographic key. The key may be a public key, a private key, or a secret key; each of these types of keys has subtypes for use in specific mechanisms. This view is illustrated in the following figure:

![Figure 2, Object Hierarchy](image-url)
Objects are also classified according to their lifetime and visibility. “Token objects” are visible to all applications connected to the token that have sufficient permission, and remain on the token even after the “sessions” (connections between an application and the token) are closed and the token is removed from its slot. “Session objects” are more temporary: whenever a session is closed by any means, all session objects created by that session are automatically destroyed. In addition, session objects are only visible to the application which created them.

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

See Table 6 on page 22 for further clarification on access to objects.

A token can create and destroy objects, manipulate them, and search for them. It can also perform cryptographic functions with objects. A token may have an internal random number generator.

It is important to distinguish between the logical view of a token and the actual implementation, because not all cryptographic devices will have this concept of “objects,” or be able to perform every kind of cryptographic function. Many devices will simply have fixed storage places for keys of a fixed algorithm, and be able to do a limited set of operations. Cryptoki's role is to translate this into the logical view, mapping attributes to fixed storage elements and so on. Not all Cryptoki libraries and tokens need to support every object type. It is expected that standard “profiles” will be developed, specifying sets of algorithms to be supported.

“Attributes” are characteristics that distinguish an instance of an object. In Cryptoki, there are general attributes, such as whether the object is private or public. There are also attributes that are specific to a particular type of object, such as a modulus or exponent for RSA keys.

### 6.5 Users

This version of Cryptoki recognizes two token user types. One type is a Security Officer (SO). The other type is the normal user. Only the normal user is allowed access to private objects on the token, and that access is granted only after the normal user has been authenticated. Some tokens may also require that a user be authenticated before any cryptographic function can be performed on the token, whether or not it involves private objects. The role of the SO is to initialize a token and to set the normal user’s PIN (or otherwise define, by some method outside the scope of this version of Cryptoki, how the normal user may be authenticated), and possibly to manipulate some public objects. The normal user cannot log in until the SO has set the normal user’s PIN.
Other than the support for two types of user, Cryptoki does not address the relationship between the SO and a community of users. In particular, the SO and the normal user may be the same person or may be different, but such matters are outside the scope of this standard.

With respect to PINs that are entered through an application, Cryptoki assumes only that they are variable-length strings of characters from the set in Table 3. Any translation to the device’s requirements is left to the Cryptoki library. The following issues are beyond the scope of Cryptoki:

- Any padding of PINs.
- How the PINs are generated (by the user, by the application, or by some other means).

PINs that are supplied by some means other than through an application (e.g., PINs entered via a PINpad on the token) are even more abstract. Cryptoki knows how to wait (if need be) for such a PIN to be supplied and used, and little more.

6.6 Applications and their use of Cryptoki

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

6.6.1 Applications and processes

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

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

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

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

6.6.2 Applications and threads

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

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

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

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

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

The 3\textsuperscript{rd} and 4\textsuperscript{th} types of behavior listed above are appropriate for multi-threaded applications which are not using the native operating system thread model. The application-supplied synchronization primitives consist of four functions for handling mutex (mutual exclusion) objects in the application’s threading model. Mutex objects are simple objects which can be in either of two states at any given time: unlocked or locked. If a call is made by a thread to lock a mutex which is already locked, that thread blocks (waits) until the mutex is unlocked; then it locks it and the call returns. If more than one thread is blocking on a particular mutex, and that mutex becomes unlocked, then exactly one of those threads will get the lock on the mutex and return control to the caller (the other blocking threads will continue to block and wait for their turn).
See Section 9.7 for more information on Cryptoki’s view of mutex objects.

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

6.7 Sessions

Cryptoki requires that an application open one or more sessions with a token to gain access to the token’s objects and functions. A session provides a logical connection between the application and the token. A session can be a read/write (R/W) session or a read-only (R/O) session. Read/write and read-only refer to the access to token objects, not to session objects. In both session types, an application can create, read, write and destroy session objects, and read token objects. However, only in a read/write session can an application create, modify, and destroy token objects.

After it opens a session, an application has access to the token’s public objects. All threads of a given application have access to exactly the same sessions and the same session objects. To gain access to the token’s private objects, the normal user must log in and be authenticated.

When a session is closed, any session objects which were created in that session are destroyed. This holds even for session objects which are “being used” by other sessions. That is, if a single application has multiple sessions open with a token, and it uses one of them to create a session object, then that session object is visible through any of that application’s sessions. However, as soon as the session that was used to create the object is closed, that object is destroyed.

Cryptoki supports multiple sessions on multiple tokens. An application may have one or more sessions with one or more tokens. In general, a token may have multiple sessions with one or more applications. A particular token may allow an application to have only a limited number of sessions—or only a limited number of read/write sessions—however.

An open session can be in one of several states. The session state determines allowable access to objects and functions that can be performed on them. The session states are described in Section 6.7.1 and Section 6.7.2.

6.7.1 Read-only session states

A read-only session can be in one of two states, as illustrated in the following figure. When the session is initially opened, it is in either the “R/O Public Session” state (if the application has no previously open sessions that are logged in) or the “R/O User Functions” state (if the application already has an open session that is logged in). Note that read-only SO sessions do not exist.
The following table describes the session states:

Table 4, Read-Only Session States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/O Public Session</td>
<td>The application has opened a read-only session. The application has read-only access to public token objects and read/write access to public session objects.</td>
</tr>
<tr>
<td>R/O User Functions</td>
<td>The normal user has been authenticated to the token. The application has read-only access to all token objects (public or private) and read/write access to all session objects (public or private).</td>
</tr>
</tbody>
</table>

6.7.2 Read/write session states

A read/write session can be in one of three states, as illustrated in the following figure. When the session is opened, it is in either the “R/W Public Session” state (if the application has no previously open sessions that are logged in), the “R/W User Functions” state (if the application already has an open session that the normal user is logged into), or the “R/W SO Functions” state (if the application already has an open session that the SO is logged into).
6. GENERAL OVERVIEW

Figure 4, Read/Write Session States

The following table describes the session states:

Table 5, Read/Write Session States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W Public Session</td>
<td>The application has opened a read/write session. The application has read/write access to all public objects.</td>
</tr>
<tr>
<td>R/W SO Functions</td>
<td>The Security Officer has been authenticated to the token. The application has read/write access only to public objects on the token, not to private objects. The SO can set the normal user’s PIN.</td>
</tr>
<tr>
<td>R/W User Functions</td>
<td>The normal user has been authenticated to the token. The application has read/write access to all objects.</td>
</tr>
</tbody>
</table>

6.7.3 Permitted object accesses by sessions

The following table summarizes the kind of access each type of session has to each type of object. A given type of session has either read-only access, read/write access, or no access whatsoever to a given type of object.

Note that creating or deleting an object requires read/write access to it, e.g., a “R/O User Functions” session cannot create or delete a token object.
Table 6, Access to Different Types Objects by Different Types of Sessions

<table>
<thead>
<tr>
<th>Type of object</th>
<th>R/O Public</th>
<th>R/W Public</th>
<th>R/O User</th>
<th>R/W User</th>
<th>R/W SO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public session object</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Private session object</td>
<td></td>
<td></td>
<td>R/W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public token object</td>
<td>R/O</td>
<td>R/W</td>
<td>R/O</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Private token object</td>
<td></td>
<td></td>
<td>R/O</td>
<td></td>
<td>R/W</td>
</tr>
</tbody>
</table>

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

6.7.4 Session events

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

Table 7, Session Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurs when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log In SO</td>
<td>the SO is authenticated to the token.</td>
</tr>
<tr>
<td>Log In User</td>
<td>the normal user is authenticated to the token.</td>
</tr>
<tr>
<td>Log Out</td>
<td>the application logs out the current user (SO or normal user).</td>
</tr>
<tr>
<td>Close Session</td>
<td>the application closes the session or closes all sessions.</td>
</tr>
<tr>
<td>Device Removed</td>
<td>the device underlying the token has been removed from its slot.</td>
</tr>
</tbody>
</table>

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

In Cryptoki, all sessions that an application has with a token must have the same login/logout status (i.e., for a given application and token, one of the following holds: all sessions are public sessions; all sessions are SO sessions; or all sessions are user sessions). When an application’s session logs into a token, all of that application’s sessions with that token become logged in, and when an application’s session logs out of a token, all of that application’s sessions with that token become logged out. Similarly, for example, if an application already has a R/O user session open with a token, and then opens a R/W session with that token, the R/W session is automatically logged in.
This implies that a given application may not simultaneously have SO sessions and user sessions open with a given token. It also implies that if an application has a R/W SO session with a token, then it may not open a R/O session with that token, since R/O SO sessions do not exist. For the same reason, if an application has a R/O session open, then it may not log any other session into the token as the SO.

### 6.7.5 Session handles and object handles

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

Cryptoki also has object handles, which are identifiers used to manipulate Cryptoki objects. Object handles are similar to session handles in the sense that visibility of a given object through an object handle is the same among all threads of a given application. R/O sessions, of course, only have read-only access to token objects, whereas R/W sessions have read/write access to token objects.

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

\[
\text{CK_INVALID_HANDLE}
\]

### 6.7.6 Capabilities of sessions

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

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

6.7.7 Example of use of sessions

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

We caution that our example is decidedly not meant to indicate how multiple applications should use Cryptoki simultaneously; rather, it is meant to clarify what uses of Cryptoki’s sessions and objects and handles are permissible. In other words, instead of demonstrating good technique here, we demonstrate “pushing the envelope”.

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

1. A1 and B1 each initialize the Cryptoki library by calling C_Initialize (the specifics of Cryptoki functions will be explained in Section 10.12). Note that exactly one call to C_Initialize should be made for each application (as opposed to one call for every thread, for example).

2. A1 opens a R/W session and receives the session handle 7 for the session. Since this is the first session to be opened for A, it is a public session.

3. A2 opens a R/O session and receives the session handle 4. Since all of A’s existing sessions are public sessions, session 4 is also a public session.

4. A1 attempts to log the SO into session 7. The attempt fails, because if session 7 becomes an SO session, then session 4 does, as well, and R/O SO sessions do not exist. A1 receives an error code indicating that the existence of a R/O session has blocked this attempt to log in (CKR_SESSION_READ_ONLY_EXISTS).

5. A2 logs the normal user into session 7. This turns session 7 into a R/W user session, and turns session 4 into a R/O user session. Note that because A1 and A2 belong to
the same application, they have equal access to all sessions, and therefore, A2 is able to perform this action.

6. A2 opens a R/W session and receives the session handle 9. Since all of A’s existing sessions are user sessions, session 9 is also a user session.


8. B1 attempts to log out session 4. The attempt fails, because A and B have no access rights to each other’s sessions or objects. B1 receives an error message which indicates that there is no such session handle (CKR_SESSION_HANDLE_INVALID).

9. B2 attempts to close session 4. The attempt fails in precisely the same way as B1’s attempt to log out session 4 failed (i.e., B2 receives a CKR_SESSION_HANDLE_INVALID error code).

10. B1 opens a R/W session and receives the session handle 7. Note that, as far as B is concerned, this is the first occurrence of session handle 7. A’s session 7 and B’s session 7 are completely different sessions.

11. B1 logs the SO into [B’s] session 7. This turns B’s session 7 into a R/W SO session, and has no effect on either of A’s sessions.

12. B2 attempts to open a R/O session. The attempt fails, since B already has an SO session open, and R/O SO sessions do not exist. B1 receives an error message indicating that the existence of an SO session has blocked this attempt to open a R/O session (CKR_SESSION_READ_WRITE_SO_EXISTS).

13. A1 uses [A’s] session 7 to create a session object O1 of some sort and receives the object handle 7. Note that a Cryptoki implementation may or may not support separate spaces of handles for sessions and objects.

14. B1 uses [B’s] session 7 to create a token object O2 of some sort and receives the object handle 7. As with session handles, different applications have no access rights to each other’s object handles, and so B’s object handle 7 is entirely different from A’s object handle 7. Of course, since B1 is an SO session, it cannot create private objects, and so O2 must be a public object (if B1 attempted to create a private object, the attempt would fail with error code CKR_USER_NOT_LOGGED_IN or CKR_TEMPLATE_INCONSISTENT).

15. B2 uses [B’s] session 7 to perform some operation to modify the object associated with [B’s] object handle 7. This modifies O2.

16. A1 uses [A’s] session 4 to perform an object search operation to get a handle for O2. The search returns object handle 1. Note that A’s object handle 1 and B’s object handle 7 now point to the same object.
17. A1 attempts to use [A’s] session 4 to modify the object associated with [A’s] object handle 1. The attempt fails, because A’s session 4 is a R/O session, and is therefore incapable of modifying O2, which is a token object. A1 receives an error message indicating that the session is a R/O session (CKR_SESSION_READ_ONLY).

18. A1 uses [A’s] session 7 to modify the object associated with [A’s] object handle 1. This time, since A’s session 7 is a R/W session, the attempt succeeds in modifying O2.

19. B1 uses [B’s] session 7 to perform an object search operation to find O1. Since O1 is a session object belonging to A, however, the search does not succeed.

20. A2 uses [A’s] session 4 to perform some operation to modify the object associated with [A’s] object handle 7. This operation modifies O1.

21. A2 uses [A’s] session 7 to destroy the object associated with [A’s] object handle 1. This destroys O2.

22. B1 attempts to perform some operation with the object associated with [B’s] object handle 7. The attempt fails, since there is no longer any such object. B1 receives an error message indicating that its object handle is invalid (CKR_OBJECT_HANDLE_INVALID).

23. A1 logs out [A’s] session 4. This turns A’s session 4 into a R/O public session, and turns A’s session 7 into a R/W public session.

24. A1 closes [A’s] session 7. This destroys the session object O1, which was created by A’s session 7.

25. A2 attempt to use [A’s] session 4 to perform some operation with the object associated with [A’s] object handle 7. The attempt fails, since there is no longer any such object. It returns a CKR_OBJECT_HANDLE_INVALID.

26. A2 executes a call to C_CloseAllSessions. This closes [A’s] session 4. At this point, if A were to open a new session, the session would not be logged in (i.e., it would be a public session).

27. B2 closes [B’s] session 7. At this point, if B were to open a new session, the session would not be logged in.

28. A and B each call C_Finalize to indicate that they are done with the Cryptoki library.

6.8 Secondary authentication (Deprecated)

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

The Cryptoki API consists of a number of functions, spanning slot and token management and object management, as well as cryptographic functions. These functions are presented in the following table:

Table 8, Summary of Cryptoki Functions

<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>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></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_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_Logout</td>
<td>logs out from a token</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Object management</td>
<td>C_CreateObject</td>
<td>creates an object</td>
</tr>
<tr>
<td>functions</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>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 digesting</td>
<td>C_DigestInit</td>
<td>initializes a message-digesting operation</td>
</tr>
<tr>
<td>functions</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>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------</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>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>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 functions</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></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>Category</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</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>

7 Security considerations

As an interface to cryptographic devices, Cryptoki provides a basis for security in a computer or communications system. Two of the particular features of the interface that facilitate such security are the following:

1. Access to private objects on the token, and possibly to cryptographic functions and/or certificates on the token as well, requires a PIN. Thus, possessing the cryptographic device that implements the token may not be sufficient to use it; the PIN may also be needed.

2. Additional protection can be given to private keys and secret keys by marking them as “sensitive” or “unextractable”. Sensitive keys cannot be revealed in plaintext off the token, and unextractable keys cannot be revealed off the token even when encrypted (though they can still be used as keys).

It is expected that access to private, sensitive, or unextractable objects by means other than Cryptoki (e.g., other programming interfaces, or reverse engineering of the device) would be difficult.

If a device does not have a tamper-proof environment or protected memory in which to store private and sensitive objects, the device may encrypt the objects with a master key which is perhaps derived from the user’s PIN. The particular mechanism for protecting private objects is left to the device implementation, however.

Based on these features it should be possible to design applications in such a way that the token can provide adequate security for the objects the applications manage.

Of course, cryptography is only one element of security, and the token is only one component in a system. While the token itself may be secure, one must also consider the security of the operating system by which the application interfaces to it, especially since the PIN may be passed through the operating system. This can make it easy for a rogue...
application on the operating system to obtain the PIN; it is also possible that other
devices monitoring communication lines to the cryptographic device can obtain the PIN.
Rogue applications and devices may also change the commands sent to the cryptographic
device to obtain services other than what the application requested.

It is important to be sure that the system is secure against such attack. Cryptoki may well
play a role here; for instance, a token may be involved in the “booting up” of the system.

We note that none of the attacks just described can compromise keys marked “sensitive,”
since a key that is sensitive will always remain sensitive. Similarly, a key that is
unextractable cannot be modified to be extractable.

An application may also want to be sure that the token is “legitimate” in some sense (for
a variety of reasons, including export restrictions and basic security). This is outside the
scope of the present standard, but it can be achieved by distributing the token with a
built-in, certified public/private-key pair, by which the token can prove its identity. The
certificate would be signed by an authority (presumably the one indicating that the token
is “legitimate”) whose public key is known to the application. The application would
verify the certificate and challenge the token to prove its identity by signing a time-
varying message with its built-in private key.

Once a normal user has been authenticated to the token, Cryptoki does not restrict which
cryptographic operations the user may perform; the user may perform any operation
supported by the token. Some tokens may not even require any type of authentication to
make use of its cryptographic functions.

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

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

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

8.1 Structure packing

Cryptoki structures are packed to occupy as little space as is possible. In particular, on
the Win32 and Win16 platforms, Cryptoki structures should be packed with 1-byte
alignment. In a UNIX environment, it may or may not be necessary (or even possible) to
alter the byte-alignment of structures.
8.2 Pointer-related macros

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

♦ **CK_PTR**

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

```
typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

♦ **CK_DEFINE_FUNCTION**

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

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

♦ **CK_DECLARE_FUNCTION**

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

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

♦ **CK_DECLARE_FUNCTION_POINTER**

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

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

or:

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

♦ CK_CALLBACK_FUNCTION

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

CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);

or:

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

♦ NULL_PTR

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

8.3 Sample platform- and compiler-dependent code

8.3.1 Win32

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

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

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

/* Ensures the calling convention for Win32 builds */
#define CK_CALL_SPEC __cdecl
#define CK_PTR *
#define CK_DEFINE_FUNCTION(returnType, name) \
   returnType CK_EXPORT_SPEC CK_CALL_SPEC name
#define CK_DECLARE_FUNCTION(returnType, name) \
   returnType CK_EXPORT_SPEC CK_CALL_SPEC name
#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
   returnType CK_IMPORT_SPEC ( CK_CALL_SPEC CK_PTR name)
#define CK_CALLBACK_FUNCTION(returnType, name) \
   returnType (CK_CALL_SPEC CK_PTR name)

#ifndef NULL_PTR
#define NULL_PTR 0
#endif

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

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

#pragma pack(pop, cryptoki)

### 8.3.2 Win16

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

#pragma pack(1)

#define CK_PTR far *
#define CK_DEFINE_FUNCTION(returnType, name)  
 returnType __export _far _pascal name

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

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

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

#ifndef NULL_PTR
#define NULL_PTR 0
#endif

### 8.3.3 Generic UNIX

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

#define CK_PTR *

#define CK_DEFINE_FUNCTION(returnType, name)  
 returnType name

#define CK_DECLARE_FUNCTION(returnType, name)  
 returnType name

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

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

#ifndef NULL_PTR
#define NULL_PTR 0
#endif
9 General data types

The general Cryptoki data types are described in the following subsections. The data types for holding parameters for various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves, in Section 12.

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

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

9.1 General information

Cryptoki represents general information with the following types:

♦ CK_VERSION; CK_VERSION_PTR

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

```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 9 below lists the major and minor version values for the officially published Cryptoki specifications.
Table 9, Major and minor version values for published Cryptoki specifications

<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>
</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:

```c
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 document; 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:

```c
typedef CK_ULONG CK_NOTIFICATION;
```

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

**CKN_SURRENDER**

The notifications have the following meanings:

| **CKN_SURRENDER** | Cryptoki is surrendering the execution of a function executing in a session so that the application may perform other operations. After performing any desired operations, the application should indicate to Cryptoki whether to continue or cancel the function (see Section 11.17.1). |

9.2 Slot and token types

Cryptoki represents slot and token information with the following types:

♦ **CK_SLOT_ID; CK_SLOT_ID_PTR**

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

```c
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 (‘ ’). Should not be null-terminated.

- **manufacturerID**: ID of the slot manufacturer. Must be padded with the blank character (‘ ’). Should not be null-terminated.

- **flags**: bits flags that provide capabilities of the slot. The flags are defined below.

- **hardwareVersion**: version number of the slot’s hardware

- **firmwareVersion**: version number of the slot’s firmware

The following table defines the **flags** field:

<table>
<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_CHAR model[16];
    CK_CHAR serialNumber[16];
    CK_FLAGS flags;
    CK_ULONG ulMaxSessionCount;
    CK_ULONG ulSessionCount;
    CK_ULONG ulMaxRwSessionCount;
    CK_ULONG ulRwSessionCount;
    CK_ULONG ulMaxPinLen;
    CK_ULONG ulMinPinLen;
    CK_ULONG ulTotalPublicMemory;
    CK_ULONG ulFreePublicMemory;
    CK_ULONG ulTotalPrivateMemory;
    CK_ULONG ulFreePrivateMemory;
    CK_VERSION hardwareVersion;
    CK_VERSION firmwareVersion;
    CK_CHAR utcTime[16];
} CK_TOKEN_INFO;
```

The fields of the structure have the following meanings:

- **label**: application-defined label, assigned during token initialization. Must be padded with the blank character (' '). Should not be null-terminated.

- **manufacturerID**: ID of the device manufacturer. Must be padded with the blank character (' '). Should not be null-terminated.

- **model**: model of the device. Must be padded with the blank character (' '). Should not be null-terminated.

- **serialNumber**: character-string serial number of the device. Must be padded with the blank character (' '). Should not be null-terminated.

- **flags**: bit flags indicating capabilities and status of the device as defined below

- **ulMaxSessionCount**: maximum number of sessions that can be opened with the token at one time by a single application (see note below)
### 9. General Data Types

- **ulSessionCount**: number of sessions that this application currently has open with the token (see note below)
- **ulMaxRwSessionCount**: maximum number of read/write sessions that can be opened with the token at one time by a single application (see note below)
- **ulRwSessionCount**: number of read/write sessions that this application currently has open with the token (see note below)
- **ulMaxPinLen**: maximum length in bytes of the PIN
- **ulMinPinLen**: minimum length in bytes of the PIN
- **ulTotalPublicMemory**: the total amount of memory on the token in bytes in which public objects may be stored (see note below)
- **ulFreePublicMemory**: the amount of free (unused) memory on the token in bytes for public objects (see note below)
- **ulTotalPrivateMemory**: the total amount of memory on the token in bytes in which private objects may be stored (see note below)
- **ulFreePrivateMemory**: the amount of free (unused) memory on the token in bytes for private objects (see note below)
- **hardwareVersion**: version number of hardware
- **firmwareVersion**: version number of firmware
- **utcTime**: current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved ‘0’ characters). The value of this field only makes sense for tokens equipped with a clock, as indicated in the token information flags (see below)
The following table defines the *flags* field:

**Table 11, Token Information Flags**

<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>0x00000010</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>Bit Flag</td>
<td>Mask</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>---------</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 11.13)</td>
</tr>
<tr>
<td>CKF_TOKEN_INITIALIZED</td>
<td>0x00000400</td>
<td>True if the token has been initialized using C_InitializeToken or an equivalent mechanism outside the scope of this standard. Calling C_InitializeToken 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>CKF_USER_PIN_FINAL_TRY</td>
<td>0x00020000</td>
<td>True if supplying an incorrect user PIN will it to become locked.</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_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 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. User 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>
</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_COUNT_LOW**, **CKF_USER_PIN_FINAL_TRY**, and **CKF_SO_PIN_FINAL_TRY** flags may always be set to false if the token does not support the functionality or will not reveal the information because of its security policy.

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

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

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

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

```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.

9.3 Session types

Cryptoki represents session information with the following types:

- CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR

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

typedef CK_ULONG CK_SESSION_HANDLE;

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

CK_INVALID_HANDLE

CK_SESSION_HANDLE_PTR is a pointer to a CK_SESSION_HANDLE.

- CK_USER_TYPE

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

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

- CKU_SO
- CKU_USER
- CKUCONTEXT_SPECIFIC

♦ CK_STATE

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

```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 12, Session Information Flags

<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**.

### 9.4 Object types

Cryptoki represents object information with the following types:

- **CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR**

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

```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.

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.

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.

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

♦ **CK_CERTIFICATE_TYPE**

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

```
typedef CK_ULONG CK_CERTIFICATE_TYPE;
```

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

Vendor defined values for this type may also be specified.

`CKC_VENDOR_DEFINED`

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

♦ **CK_ATTRIBUTE_TYPE**

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

```
typedef CK_ULONG CK_ATTRIBUTE_TYPE;
```

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

Vendor defined values for this type may also be specified.

`CKA_VENDOR_DEFINED`

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

♦ 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).

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 that applications that need to interoperate with these libraries therefore have to be flexible in what they accept as an empty value.

9.5 Data types for mechanisms

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

♦ **CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR**

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

```c
typedef CK_ULONG CK_MECHANISM_TYPE;
```

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

Vendor defined values for this type may also be specified.

**CKM_VENDOR_DEFINED**

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

**CK_MECHANISM_TYPE_PTR** is a pointer to a **CK_MECHANISM_TYPE**.

♦ **CK_MECHANISM; CK_MECHANISM_PTR**

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

```c
typedef struct CK_MECHANISM {
    CK_MECHANISM_TYPE mechanism;
    CK_VOID_PTR pParameter;
    CK_ULONG ulParameterLen;
} CK_MECHANISM;
```
The fields of the structure have the following meanings:

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

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

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

♦ **CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR**

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

```c
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>
<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_ENCRYPT</td>
<td>0x00000100</td>
<td>True if the mechanism can be used with C_EncryptInit</td>
</tr>
<tr>
<td>CKF_DECRYPT</td>
<td>0x00000200</td>
<td>True if the mechanism can be used with C_DecryptInit</td>
</tr>
<tr>
<td>CKF_DIGEST</td>
<td>0x00000400</td>
<td>True if the mechanism can be used with C_DigestInit</td>
</tr>
<tr>
<td>CKF_SIGN</td>
<td>0x00000800</td>
<td>True if the mechanism can be used with C_SignInit</td>
</tr>
<tr>
<td>CKF_SIGN_RECOVER</td>
<td>0x00001000</td>
<td>True if the mechanism can be used with C_SignRecoverInit</td>
</tr>
<tr>
<td>CKF_VERIFY</td>
<td>0x00002000</td>
<td>True if the mechanism can be used with C_VerifyInit</td>
</tr>
<tr>
<td>CKF_VERIFY_RECOVER</td>
<td>0x00004000</td>
<td>True if the mechanism can be used with C_VerifyRecoverInit</td>
</tr>
<tr>
<td>CKF_GENERATE</td>
<td>0x00008000</td>
<td>True if the mechanism can be used with C_GenerateKey</td>
</tr>
<tr>
<td>CKF_GENERATE_KEY_PAIR</td>
<td>0x00010000</td>
<td>True if the mechanism can be used with C_GenerateKeyPair</td>
</tr>
<tr>
<td>CKF_WRAP</td>
<td>0x00020000</td>
<td>True if the mechanism can be used with C_WrapKey</td>
</tr>
<tr>
<td>CKF_UNWRAP</td>
<td>0x00040000</td>
<td>True if the mechanism can be used with C_UnwrapKey</td>
</tr>
<tr>
<td>CKF_DERIVE</td>
<td>0x00080000</td>
<td>True if the mechanism can be used with C_DeriveKey</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**.

### 9.6 Function types

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

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

```c
typedef CK_ULONG CK_RV;
```

Vendor defined values for this type may also be specified.

CK_VENDOR_DEFINED

Section 11.1 defines the meaning of each CK_RV value. Return values CKR_VENDOR_DEFINED and above are permanently reserved for token vendors. For interoperability, vendors should register their return values through the PKCS process.

CK_NOTIFY

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

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

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

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

CK_C_XXX

Cryptoki also defines an entire family of other function pointer types. For each function C_XXX in the Cryptoki API (see Section 10.12 for detailed information about each of them), Cryptoki defines a type CK_C_XXX, which is a pointer to a function with the same arguments and return value as C_XXX has. An appropriately-set variable of type CK_C_XXX may be used by an application to call the Cryptoki function C_XXX.
CK_FUNCTION_LIST 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_C_GetInfo C_GetInfo;
    CK_C_GetFunctionList C_GetFunctionList;
    CK_C_GetSlotList C_GetSlotList;
    CK_C_GetSlotInfo C_GetSlotInfo;
    CK_C_GetTokenInfo C_GetTokenInfo;
    CK_C_GetMechanismList C_GetMechanismList;
    CK_C_GetMechanismInfo C_GetMechanismInfo;
    CK_C_InitToken C_InitToken;
    CK_C_InitPIN C_InitPIN;
    CK_C_SetPIN C_SetPIN;
    CK_C_OpenSession C_OpenSession;
    CK_C_CloseSession C_CloseSession;
    CK_C_CloseAllSessions C_CloseAllSessions;
    CK_C_GetSessionInfo C_GetSessionInfo;
    CK_C_GetOperationState C_GetOperationState;
    CK_C_SetOperationState C_SetOperationState;
    CK_C_Login C_Login;
    CK_C_Logout C_Logout;
    CK_C_CreateObject C_CreateObject;
    CK_C_CopyObject C_CopyObject;
    CK_C_DestroyObject C_DestroyObject;
    CK_C_GetObjectSize C_GetObjectSize;
    CK_C_GetAttributeValue C_GetAttributeValue;
    CK_C_SetAttributeValue C_SetAttributeValue;
    CK_C_FindObjectsInit C_FindObjectsInit;
    CK_C_FindObjects C_FindObjects;
    CK_C_FindObjectsFinal C_FindObjectsFinal;
    CK_C_EncryptInit C_EncryptInit;
    CK_C_Encrypt C_Encrypt;
    CK_C_EncryptUpdate C_EncryptUpdate;
    CK_C_EncryptFinal C_EncryptFinal;
    CK_C_DecryptInit C_DecryptInit;
    CK_C_Decrypt C_Decrypt;
    CK_C_DecryptUpdate C_DecryptUpdate;
    CK_C_DecryptFinal C_DecryptFinal;
    CK_C_DigestInit C_DigestInit;
    CK_C_Digest C_Digest;
    CK_C_DigestUpdate C_DigestUpdate;
    CK_C_DigestKey C_DigestKey;
};
```
Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section 11.2). The value that this pointer points to can be used by an application to quickly find out where the executable code for each function in the Cryptoki API is located. *Every function in the Cryptoki API must have an entry point defined in the Cryptoki library’s **CK_FUNCTION_LIST** structure.* If a particular function in the Cryptoki API is not supported by a library, then the function pointer for that function in the library’s **CK_FUNCTION_LIST** structure should point to a function stub which simply returns CKR_FUNCTION_NOT_SUPPORTED.

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

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

**CK_FUNCTION_LIST_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_PTR**.
9.7 Locking-related types

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

♦ **CK_CREATEMUTEX**

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

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

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

♦ **CK_DESTROYMUTEX**

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

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

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

♦ **CK_LOCKMUTEX** and **CK_UNLOCKMUTEX**

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

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

• If a CK_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:
  • If exactly one thread was blocking on that particular mutex, then that thread stops blocking, obtains a lock on that mutex, and its CK_LOCKMUTEX call returns.
  • If more than one thread was blocking on that particular mutex, then exactly one of the blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on the mutex, and its CK_LOCKMUTEX call returns. All other threads blocking on that particular mutex continue to block.

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

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

CK_LOCKMUTEX is defined as follows:

```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; CK_C_INITIALIZE_ARGS_PTR

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

```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 14, C_Initialize Parameter Flags**

<table>
<thead>
<tr>
<th>Bit Flag</th>
<th>Mask</th>
<th>Meaning</th>
<th></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 <em>not</em> use native operating system calls to spawn new threads; false if they may</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

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

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

![Object Attribute Hierarchy Diagram]

**Figure 5, Object Attribute Hierarchy**

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

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

Tables throughout most of Section 10 define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (e.g., `CK_OBJECT_CLASS`). Attribute values may also take the following types:
10. OBJECTS

Byte array  an arbitrary string (array) of CK_BYTEs

Big integer  a string of CK_BYTEs representing an unsigned integer of arbitrary size, most-significant byte first (e.g., the integer 32768 is represented as the 2-byte string 0x80 0x00)

Local string  an unpadded string of CK_CHARs (see Table 3) with no null-termination

RFC2279 string  an unpadded string of CK_UTF8CHARs with no null-termination

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

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

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

10.1 Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 11.14) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see Section 12). In any case, all the required attributes supported by an object class that do not have default values must be specified when an object is created, either in the template or by the function itself.

10.1.1 Creating objects

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

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

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

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

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

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

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

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

10.1.2 Modifying objects

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

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

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

10.1.3 Copying objects

Objects may be copied with the Cryptoki function C_CopyObject (see Section 11.7). In the process of copying an object, C_CopyObject also modifies the attributes of the newly-created copy according to an application-supplied template.

The Cryptoki attributes which can be modified during the course of a C_CopyObject operation are the same as the Cryptoki attributes which are described as being modifiable, plus the three special attributes CKA_TOKEN, CKA_PRIVATE, and CKA_MODIFIABLE. To be more precise, these attributes are modifiable during the course of a C_CopyObject operation insofar as the Cryptoki specification is concerned. A particular token might not actually support modification of some such attributes during the course of a C_CopyObject operation. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable during the course of a C_CopyObject operation might depend on the values of certain attributes of the object. For example, a secret key object’s CKA_SENSITIVE attribute can be changed from
CK_FALSE to CK_TRUE during the course of a **C_CopyObject** operation, but not the other way around.

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

### 10.2 Common attributes

**Table 15, Common footnotes for object attribute tables**

<table>
<thead>
<tr>
<th>Footnote</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Must be specified when object is created with <strong>C_CreateObject</strong>.</td>
</tr>
<tr>
<td>2</td>
<td>Must <strong>not</strong> be specified when object is created with <strong>C_CreateObject</strong>.</td>
</tr>
<tr>
<td>3</td>
<td>Must be specified when object is generated with <strong>C_GenerateKey</strong> or <strong>C_GenerateKeyPair</strong>.</td>
</tr>
<tr>
<td>4</td>
<td>Must <strong>not</strong> be specified when object is generated with <strong>C_GenerateKey</strong> or <strong>C_GenerateKeyPair</strong>.</td>
</tr>
<tr>
<td>5</td>
<td>Must be specified when object is unwrapped with <strong>C_UnwrapKey</strong>.</td>
</tr>
<tr>
<td>6</td>
<td>Must <strong>not</strong> be specified when object is unwrapped with <strong>C_UnwrapKey</strong>.</td>
</tr>
<tr>
<td>7</td>
<td>Cannot be revealed if object has its <strong>CKA_SENSITIVE</strong> attribute set to CK_TRUE or its <strong>CKA_EXTRACTABLE</strong> attribute set to CK_FALSE.</td>
</tr>
<tr>
<td>8</td>
<td>May be modified after object is created with a <strong>C_SetAttributeValue</strong> call, or in the process of copying object with a <strong>C_CopyObject</strong> call. However, it is possible that a particular token may not permit modification of the attribute during the course of a <strong>C_CopyObject</strong> call.</td>
</tr>
<tr>
<td>9</td>
<td>Default value is token-specific, and may depend on the values of other attributes.</td>
</tr>
<tr>
<td>10</td>
<td>Can only be set to CK_TRUE by the SO user.</td>
</tr>
<tr>
<td>11</td>
<td>Attribute cannot be changed once set to CK_TRUE. It becomes a read only attribute.</td>
</tr>
<tr>
<td>12</td>
<td>Attribute cannot be changed once set to CK_FALSE. It becomes a read only attribute.</td>
</tr>
</tbody>
</table>
Table 16, Common Object Attributes

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

\(^1\) Refer to table Table 15 for footnotes

The above table defines the attributes common to all objects.

10.3 Hardware Feature Objects

10.3.1 Definitions

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

10.3.2 Overview

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

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

Table 17, Hardware Feature Common Attributes

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

\(^1\) Refer to table Table 15 for footnotes

10.3.3 Clock

10.3.3.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type CK_HW_FEATURE.
10.3.3.2 Description

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

Table 18, Clock Object Attributes

<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 YYYYMMDDhhmmsxxx (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.

10.3.4 Monotonic Counter Objects

10.3.4.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type CK_HW_FEATURE.

10.3.4.2 Description

Monotonic counter objects represent hardware counters that exist on the device. The counter is guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an application for generating serial numbers to get some assurance of uniqueness per token.
### Table 19, Monotonic Counter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_RESET_ON_INIT&lt;sup&gt;1&lt;/sup&gt;</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_InitializeToken.</td>
</tr>
<tr>
<td>CKA_HAS_RESET&lt;sup&gt;1&lt;/sup&gt;</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&lt;sup&gt;1&lt;/sup&gt;</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>

<sup>1</sup>Read Only

The CKA_VALUE attribute may not be set by the client.

### 10.3.5 User Interface Objects

#### 10.3.5.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type CK_HW_FEATURE.

#### 10.3.5.2 Description

User interface objects represent the presentation capabilities of the device.
<table>
<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/e&quot; and &quot;a/d&quot; would set the attribute value to “a/b;a/e;a/d”.</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.

### 10.4 Storage Objects

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

**Table 21, Common Storage Object Attributes**

<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>
</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. It may or may not be the case that an unmodifiable object can be deleted.

The CKA_LABEL attribute is intended to assist users in browsing.
10.5 Data objects

10.5.1 Definitions

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

10.5.2 Overview

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

Table 22, Data Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_APPLICATION</td>
<td>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)}
};
```
10.6 Certificate objects

10.6.1 Definitions

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

10.6.2 Overview

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

Table 23, Common Certificate Object Attributes

<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>CK_ULONG</td>
<td>Categorization of the certificate: 0 = unspecified (default value), 1 = token user, 2 = authority, 3 = other entity</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>
</tbody>
</table>

Refer to table Table 15 for footnotes

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

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

The CKA_TRUSTED attribute cannot be set to CK_TRUE by an application. It must be set by a token initialization application or by the token’s SO. Trusted certificates cannot be modified.
The **CKA_CERTIFICATECATEGORY** attribute is used to indicate if a stored certificate is a user certificate for which the corresponding private key is available on the token (“token user”), a CA certificate (“authority”), or an other end-entity certificate (“other entity”). This attribute may not be modified after an object is created.

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

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

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

### 10.6.3 X.509 public key certificate objects

X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes defined for this object class:
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT(^1)</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_VALUE(^1)</td>
<td>Byte array</td>
<td>BER-encoding of the certificate</td>
</tr>
<tr>
<td>CKA_URL(^3)</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_KEY(^4)</td>
<td>Byte array</td>
<td>SHA-1 hash of the subject public key (default empty)</td>
</tr>
<tr>
<td>CKA_HASH_OF_ISSUER_PUBLIC_KEY(^4)</td>
<td>Byte array</td>
<td>SHA-1 hash of the issuer public key (default empty)</td>
</tr>
<tr>
<td>CKA_JAVA_MIDP_SECURITY_DOMAIN</td>
<td>CK_ULONG</td>
<td>Java MIDP security domain: 0 = unspecified (default value), 1 = manufacturer, 2 = operator, 3 = third party</td>
</tr>
</tbody>
</table>

\(^1\) Must be specified when the object is created.
\(^2\) Must be specified when the object is created. Must be non-empty if CKA_URL is empty.
\(^3\) Must be non-empty if CKA_VALUE is empty.
\(^4\) Can only be empty if CKA_URL is empty.

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

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

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

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

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

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

The 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)}
};
```

### 10.6.4 WTLS public key certificate objects

WTLS certificate objects (certificate type CKC_WTLS) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes defined for this object class.
Table 25: WTLS Certificate Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT^1</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^2</td>
<td>Byte array</td>
<td>WTLS-encoding of the certificate</td>
</tr>
<tr>
<td>CKA_URL^3</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^4</td>
<td>Byte array</td>
<td>SHA-1 hash of the subject public key (default empty)</td>
</tr>
<tr>
<td>CKA_HASH_OF_ISSUER_PUBLIC_KEY^4</td>
<td>Byte array</td>
<td>SHA-1 hash of the issuer public key (default empty)</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] (see References).

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

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

The 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)}
};

10.6.5 X.509 attribute certificate objects

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_OWNER¹</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_VALUE¹</td>
<td>Byte Array</td>
<td>BER-encoding of the certificate.</td>
</tr>
</tbody>
</table>

¹Must 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)}
};

10. Key objects

10.7 Definitions

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

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

10.7.2 Overview

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

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

Table 27, Common Key Attributes

<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 generated locally (i.e., on the token) with a C_GenerateKey or</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_KEY_GEN_MECHANISM</td>
<td>CK_MECHANISM_TYPE</td>
<td>C_GenerateKeyPair call • created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE</td>
</tr>
<tr>
<td>CKA_ALLOWED_MECHANISMS</td>
<td>CK_MECHANISM_TYPE_PTR, pointer to a CK_MECHANISM_TYPE array</td>
<td>Identifier of the mechanism used to generate the key material.</td>
</tr>
<tr>
<td></td>
<td></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>
</tbody>
</table>

*Refer to table Table 15 for footnotes*

The CKA_ID field is intended to distinguish among multiple keys. In the case of public and private keys, this field assists in handling multiple keys held by the same subject; the key identifier for a public key and its corresponding private key should be the same. The key identifier should also be the same as for the corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See Section 10.6 for further commentary.)

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

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

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

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

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

Public key objects (object class \texttt{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>
<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 the key subject name (default empty)</td>
</tr>
<tr>
<td>CKA_ENCRYPT^8</td>
<td>\texttt{CK_BBOOL}</td>
<td>\texttt{CK_TRUE} if key supports encryption(^9)</td>
</tr>
<tr>
<td>CKA_VERIFY^8</td>
<td>\texttt{CK_BBOOL}</td>
<td>\texttt{CK_TRUE} if key supports verification where the signature is an appendix to the data(^9)</td>
</tr>
<tr>
<td>CKA_VERIFY_RECOVER^8</td>
<td>\texttt{CK_BBOOL}</td>
<td>\texttt{CK_TRUE} if key supports verification where the data is recovered from the signature(^9)</td>
</tr>
<tr>
<td>CKA_WRAP^8</td>
<td>\texttt{CK_BBOOL}</td>
<td>\texttt{CK_TRUE} if key supports wrapping (i.e., can be used to wrap other keys)(^9)</td>
</tr>
<tr>
<td>CKA_TRUSTED^10</td>
<td>\texttt{CK_BBOOL}</td>
<td>The key can be trusted for the application that it was created. The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to \texttt{CK_TRUE}.</td>
</tr>
<tr>
<td>CKA_WRAP_TEMPLATE</td>
<td>\texttt{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 \texttt{ulValueLen} component of the attribute divided by the size of \texttt{CK_ATTRIBUTE}.</td>
</tr>
</tbody>
</table>

\(^8\) Refer to table Table 15 for footnotes

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

To map between ISO/IEC 9594-8 (X.509) \texttt{keyUsage} flags for public keys and the PKCS #11 attributes for public keys, use the following table.
Table 29, Mapping of X.509 key usage flags to cryptoki attributes for public keys

<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>

10.9 Private key objects

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

Table 30, Common Private Key Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_SUBJECT8</td>
<td>Byte array</td>
<td>DER-encoding of certificate subject name (default empty)</td>
</tr>
<tr>
<td>CKA_SENSITIVE8,11</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key is sensitive</td>
</tr>
<tr>
<td>CKA_DECRYPT8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports decryption</td>
</tr>
<tr>
<td>CKA_SIGN8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports signatures where the signature is an appendix to the data</td>
</tr>
<tr>
<td>CKA_SIGN_RECOVER8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports signatures where the data can be recovered from the signature</td>
</tr>
<tr>
<td>CKA_UNWRAP8</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)</td>
</tr>
<tr>
<td>CKA_EXTRACTABLE8,12</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key is extractable and can be wrapped</td>
</tr>
<tr>
<td>CKA_ALWAYS_SENSITIVE2,4,6</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_EXTRACTABLE2,4,6</td>
<td>CK_BBOOL</td>
<td>CK_TRUE if key has never had the CKA_SENSITIVE attribute 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_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_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>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>
</tbody>
</table>

\(^{11}\)Refer to table Table 15 for footnotes

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

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

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

Re-authentication occurs by calling C_Login with userType set to CKU_CONTEXT_SPECIFIC immediately after a cryptographic operation using the key has been initiated (e.g. after C_SignInit). In this call, the actual user type is
implicitly given by the usage requirements of the active key. If \texttt{C\_Login} returns \texttt{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 \texttt{C\_Sign}, \texttt{C\_SignFinal}...). A return value \texttt{CKR\_PIN\_INCORRECT} from \texttt{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 \texttt{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. \texttt{C\_Login} returns in this case \texttt{CKR\_PIN\_LOCKED} and this also logs the user out from the token. Failing or omitting to re-authenticate when \texttt{CKA\_ALWAYS\_AUTHENTICATE} is set to \texttt{CK\_TRUE} will result in \texttt{CKR\_USER\_NOT\_LOGGED\_IN} to be returned from calls using the key. \texttt{C\_Login} will return \texttt{CKR\_OPERATION\_NOT\_INITIALIZED}, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when \texttt{CKA\_ALWAYS\_AUTHENTICATE} is set to \texttt{CK\_FALSE}.

10.10 Secret key objects

Secret key objects (object class \texttt{CKO\_SECRET\_KEY}) hold secret keys. The following table defines the attributes common to all secret keys, in addition to the common attributes defined for this object class:
Table 31, Common Secret Key Attributes

<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 (\textit{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 (\textit{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 (\textit{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 (\textit{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 \textit{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 \textit{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</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>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>
</tbody>
</table>

Refer to table Table 15 for footnotes

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

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

Properties:

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

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

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

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

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

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

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

10.11 Domain parameter objects

10.11.1 Definitions

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

10.11.2 Overview

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

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

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

The following table defines the attributes common to domain parameter objects in addition to the common attributes defined for this object class:
Table 32, Common Domain Parameter Attributes

<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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which had its CKA_LOCAL attribute set to CK_TRUE</td>
</tr>
</tbody>
</table>

¹Refer to table Table 15 for footnotes

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

10.12 Mechanism objects

10.12.1 Definitions

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

10.12.2 Overview

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

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

Table 33, Common Mechanism Attributes

<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.
11 Functions

Cryptoki's functions are organized into the following categories:

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

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

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

- If a Cryptoki function executes successfully, it returns the value CKR_OK.

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

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

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

11.1 Function return values

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

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

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

11.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:
• CKR_GENERAL_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.

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

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

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

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

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

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

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

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

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

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

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

11.1.3 Cryptoki function return values for functions that use a token

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

- CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested function.

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

11.1.4 Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be returned by an application-supplied callback function. It is:
• CKR_CANCEL: When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a CKN_SURRENDER callback (see Section 11.17). If the callback returns the value CKR_CANCEL, then the function aborts and returns CKR_FUNCTION_CANCELED.

11.1.5 Special return values for mutex-handling functions

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

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

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

11.1.6 All other Cryptoki function return values

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

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

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

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

• CKR_ATTRIBUTE_TYPE_INVALID: An invalid attribute type was specified in a template. See Section 10.1 for more information.

• CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for a particular attribute in a template. See Section 10.1 for more information.
- CKR_BUFFER_TOO_SMALL: The output of the function is too large to fit in the supplied buffer.

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

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

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

- CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This return value has lower priority than CKR_DATA_LEN_RANGE.

- CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length. Depending on the operation’s mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than CKR_DATA_INVALID.

- CKR_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.

- CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been determined to be invalid ciphertext. This return value has lower priority than CKR_ENCRYPTED_DATA_LEN_RANGE.

- CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation’s mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular blocksize. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.

- CKR_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_UNEXTRACTABLE attribute set to CK_TRUE, 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_UNEXTRACTABLE attribute is set to CK_TRUE. Compare with CKR_KEY_NOT_WRAPPABLE.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **CKR_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`.

- **CKRTEMPLATE_INCOMPLETE**: The template specified for creating an object is incomplete, and lacks some necessary attributes. See Section 10.1 for more information.

- **CKR_TEMPLATE_INCONSISTENT**: The template specified for creating an object has conflicting attributes. See Section 10.1 for more information.

- **CKR_TOKEN_NOT_RECOGNIZED**: The Cryptoki library and/or slot does not recognize the token in the slot.
• CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.

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

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

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

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

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

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

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

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

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

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

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

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

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

11.1.7 More on relative priorities of Cryptoki errors

In general, when a Cryptoki call is made, error codes from Section 11.1.1 (other than CKR_OK) take precedence over error codes from Section 11.1.2, which take precedence over error codes from Section 11.1.3, which take precedence over error codes from Section 11.1.6. One minor implication of this is that functions that use a session handle (i.e., most functions!) never return the error code CKR_TOKEN_NOT_PRESENT (they return CKR_SESSION_HANDLE_INVALID instead). Other than these precedences, if more than one error code applies to the result of a Cryptoki call, any of the applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the descriptions of functions.
11. FUNCTIONS

11.1.8 Error code “gotchas”

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

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

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

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

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

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

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

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

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

1. If pBuf is NULL_PTR, then all that the function does is return (in *pulBufLen) a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may somewhat exceed the precise number of bytes needed, but should not exceed it by a large amount. CKR_OK is returned by the function.
2. If $pBuf$ is not NULL_PTR, then *$pulBufLen$ must contain the size in bytes of the buffer pointed to by $pBuf$. If that buffer is large enough to hold the cryptographic output produced from the input to the function, then that cryptographic output is placed there, and CKR_OK is returned by the function. If the buffer is not large enough, then CKR_BUFFER_TOO_SMALL is returned. In either case, *$pulBufLen$ is set to hold the exact number of bytes needed to hold the cryptographic output produced from the input to the function.

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

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

11.3 Disclaimer concerning sample code

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

11.4 General-purpose functions

Cryptoki provides the following general-purpose functions:

♦ C_Initialize

```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 operation system calls to spawn off new threads. In other words, the library’s code may not create its own threads. If the library is unable to function properly under this restriction, **C_Initialize** should return with the value **CKR_NEED_TO_CREATE_THREADS**.

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

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

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

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

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

If some, but not all, of the supplied function pointers to **C_Initialize** are non-NULL_PTR, then **C_Initialize** should return with the value **CKR_ARGUMENTS_BAD**.
A call to `C_Initialize` with `pInitArgs` set to NULL_PTR is treated like a call to `C_Initialize` with `pInitArgs` pointing to a `CK_C_INITIALIZE_ARGS` which has the `CreateMutex`, `DestroyMutex`, `LockMutex`, `UnlockMutex`, and `pReserved` fields set to NULL_PTR, and has the `flags` field set to 0.

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

If several applications are using Cryptoki, each one should call `C_Initialize`. Every call to `C_Initialize` should (eventually) be succeeded by a single call to `C_Finalize`. See Section 6.6 for more details.

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

Example: see `C_GetInfo`.

♦ `C_Finalize`

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

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

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

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

Example: see C_GetInfo.

♦ C_GetInfo

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

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

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

Example:

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

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

rv = C_Initialize((CK_VOID_PTR)&InitArgs);
assert(rv == CKR_OK);

rv = C_GetInfo(&info);
assert(rv == CKR_OK);
if(info.version.major == 2) {
    /* Do lots of interesting cryptographic things with the token */

    .
    .
}

rv = C_Finalize(NULL_PTR);
assert(rv == CKR_OK);
```
11.4 C_GetFunctionList

CK_FUNCTION_LIST_PTR_PTR ppFunctionList = NULL;

/* It's OK to call C_GetFunctionList before calling C_Initialize */

C_GetFunctionList(&ppFunctionList);
assert(ppFunctionList);

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

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

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

Example:

CK_FUNCTION_LIST_PTR pFunctionList;
CK_C_Initialize pC_Initialize;
CK_RV rv;

/* It's OK to call C_GetFunctionList before calling C_Initialize */
rv = C_GetFunctionList(&pFunctionList);
assert(rv == CKR_OK);
pC_Initialize = pFunctionList -> C_Initialize;

/* Call the C_Initialize function in the library */
rv = (*pC_Initialize)(NULL_PTR);

11.5 Slot and token management functions

Cryptoki provides the following functions for slot and token management:

♦ C_GetSlotList

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

There are two ways for an application to call **C_GetSlotList**:

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

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

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

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

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

Example:

```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 == CKR_OK) {
pSlotList =
    (CK_SLOT_ID_PTR)
    malloc(ulSlotCount*sizeof(CK SLOT_ID));
rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
if (rv == CKR_OK) {
    /* Now use that list of all slots */
      .
      .
}
free(pSlotList);
}

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

♦ C_GetSlotInfo

CK_DEFINE_FUNCTION(CK_RV, C_GetSlotInfo)(
    CK_SLOT_ID slotID,
    CK SLOT_INFO_PTR pInfo
    );

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

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID.
Example: see **C_GetTokenInfo**.

♦ **C_GetTokenInfo**

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

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


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);
    }
```

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

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

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

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

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

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

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

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

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

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

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

♦ **C_GetMechanismList**

```c
CK_DEFINE_FUNCTION(CK_RV, C_GetMechanismList)(
    CK_SLOT_ID slotID,
    CK_MECHANISM_TYPE_PTR pMechanismList,
    CK_ULONG_PTR pulCount
)
```

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

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

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

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

Because *C_GetMechanismList* does not allocate any space of its own, an application will often call *C_GetMechanismList* twice. However, this behavior is by no means required.
Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

Example:

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

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

♦ C_GetMechanismInfo

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

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

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

Example:
CK_SLOT_ID slotID;
CK_MECHANISM_INFO info;
CK_RV rv;

/* Get information about the CKM_MD2 mechanism for this token */
rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
if (rv == CKR_OK) {
    if (info.flags & CKF_DIGEST) {
        
    }
}

♦ C_InitToken

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

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

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

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

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

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

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_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_PTR pin = "MyPIN";
CK_UTF8CHAR label[32];
CK_RV rv;
.
.
memset(label, ' ', sizeof(label));
memcpy(label, "My first token", strlen("My first token"));
rv = C_InitToken(slotID, pin, strlen(pin), label);
if (rv == CKR_OK) {
  .
  .
}
```
11. FUNCTIONS

♦ C_InitPIN

CK_DEFINE_FUNCTION(CK_RV, C_InitPIN) (  
    CK_SESSION_HANDLE hSession,  
    CK_UTF8CHAR_PTR pPin,  
    CK_ULONG ulPinLen  
);  

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

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

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_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));  
if (rv == CKR_OK) {  
    .  
    .  
}
```
**C_SetPIN**

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

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

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

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

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

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

Example:

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

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

11.6 Session management functions

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

1. Select a token.

2. Make one or more calls to C_OpenSession to obtain one or more sessions with the token.

3. Call C_Login to log the user into the token. Since all sessions an application has with a token have a shared login state, C_Login only needs to be called for one of the sessions.

4. Perform cryptographic operations using the sessions with the token.

5. Call C_CloseSession once for each session that the application has with the token, or call C_CloseAllSessions to close all the application’s sessions simultaneously.

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

Cryptoki provides the following functions for session management:

♦ C_OpenSession

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

C_OpenSession opens a session between an application and a token in a particular slot. slotID is the slot’s ID; flags indicates the type of session; pApplication is an application-
defined pointer to be passed to the notification callback; \textit{Notify} is the address of the
notification callback function (see Section 11.17); \textit{phSession} points to the location that
receives the handle for the new session.

When opening a session with \textbf{C\_OpenSession}, the \textit{flags} parameter consists of the logical
OR of zero or more bit flags defined in the \textbf{CK\_SESSION\_INFO} data type. For legacy
reasons, the \textbf{CKF\_SERIAL\_SESSION} bit must always be set; if a call to
\textbf{C\_OpenSession} does not have this bit set, the call should return unsuccessfully with the
error code \textbf{CKR\_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 \textbf{CKR\_SESSION\_COUNT}.

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

If the application calling \textbf{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
\textbf{CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS} (see Section 6.7.7).

The \textit{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
\texttt{NULL\_PTR} as the \textit{Notify} parameter. See Section 11.17 for more information about
application callbacks.

Return values: \textbf{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \textbf{CKR\_DEVICE\_ERROR},
\textbf{CKR\_DEVICE\_MEMORY}, \textbf{CKR\_DEVICE\_REMOVED}, \textbf{CKR\_FUNCTION\_FAILED},
\textbf{CKR\_GENERAL\_ERROR}, \textbf{CKR\_HOST\_MEMORY}, \textbf{CKR\_OK},
\textbf{CKR\_SESSION\_COUNT}, \textbf{CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED},
\textbf{CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS}, \textbf{CKR\_SLOT\_ID\_INVALID},
\textbf{CKR\_TOKEN\_NOT\_PRESENT}, \textbf{CKR\_TOKEN\_NOT\_RECOGNIZED},
\textbf{CKR\_TOKEN\_WRITE\_PROTECTED}, \textbf{CKR\_ARGUMENTS\_BAD}.

Example: see \textbf{C\_CloseSession}.

\begin{verbatim}
♦ \textbf{C\_CloseSession}

\texttt{CK\_DEFINE\_FUNCTION(CK\_RV, C\_CloseSession)(
    CK\_SESSION\_HANDLE hSession
);}
\end{verbatim}

\textbf{C\_CloseSession} closes a session between an application and a token. \textit{hSession} is the
session’s handle.
When a session is closed, all session objects created by the session are destroyed automatically, even if the application has other sessions “using” the objects (see Sections 6.7.5-6.7.7 for more details).

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

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

Despite the fact this \texttt{C\_CloseSession} is supposed to close a session, the return value \texttt{CKR\_SESSION\_CLOSED} is an \textit{error} return. It actually indicates the (probably somewhat unlikely) event that while this function call was executing, another call was made to \texttt{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: \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\_OK}, \texttt{CKR\_SESSION\_CLOSED}, \texttt{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);
}
```
♦ **C_CloseAllSessions**

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

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

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

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

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

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

Example:

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

♦ **C_GetSessionInfo**

```
CK_DEFINE_FUNCTION(CK_RV, C_GetSessionInfo)(
    CK_SESSION_HANDLE hSession,
    CK_SESSION_INFO_PTR pInfo
);
```

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

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

Example:
CK_SESSION_HANDLE hSession;
CK_SESSION_INFO info;
CK_RV rv;

. 
rv = C_GetSessionInfo(hSession, &info);
if (rv == CKR_OK) {
    if (info.state == CKS_RW_USER_FUNCTIONS) {
        .
    }
}
.
.

♦ C_GetOperationState

CK_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 11.2 on producing output.

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

Consider a session which is performing a message digest operation using SHA-1 (i.e., the session is using the CKM_SHA_1 mechanism). Suppose that the message digest operation was initialized properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The application now wants to “save the state” of this digest operation, so that it can continue it later. In this particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic operations state of the session most likely consists of three distinct parts: the state of SHA-1’s 160-bit internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken together, these three pieces of information suffice to continue the current hashing operation at a later time.
Consider next a session which is performing an encryption operation with DES (a block cipher with a block size of 64 bits) in CBC (cipher-block chaining) mode (i.e., the session is using the **CKM_DES_CBC** mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already been produced and output. In this case, the cryptographic operations state of the session most likely consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some administrative data indicating that this saved state comes from a session which was performing DES encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState** for more information on whether or not the key is present in the saved state).

If a session is performing two cryptographic operations simultaneously (see Section 11.13), then the cryptographic operations state of the session will contain all the necessary information to restore both operations.

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

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


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

The state need not have been obtained from the same session (the “source session”) as it is being restored to (the “destination session”). However, the source session and destination session should have a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token. There is also no guarantee that cryptographic operations state may be carried across logins, or across different Cryptoki implementations.

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

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

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

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

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

An application can look at the CKF_RESTORE_KEY_NOT_NEEDED flag in the
flags field of the CK_TOKEN_INFO field for a token to determine whether or not it
needs to supply key handles to C_SetOperationState calls. If this flag is true, then a call
to C_SetOperationState never needs a key handle to be supplied to it. If this flag is
false, then at least some of the time, C_SetOperationState requires a key handle, and so
the application should probably always pass in any relevant key handles when restoring
cryptographic operations state to a session.

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

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

Example:

    CK_SESSION_HANDLE hSession;
    CK_MECHANISM digestMechanism;
    CK_ULONG ulStateLen;
    CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};
    CK_BYTE data2[] = {0x02, 0x04, 0x08};
    CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};
    CK_BYTE pDigest[20];
    CK_ULONG ulDigestLen;
    CK_RV rv;

    .
    .
    /* Initialize hash operation */
rv = C_DigestInit(hSession, &digestMechanism);
assert(rv == CKR_OK);

/* Start hashing */
rv = C_DigestUpdate(hSession, data1, sizeof(data1));
assert(rv == CKR_OK);

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

/* Allocate some memory and then get the state */
pState = (CK_BYTE_PTR) malloc(ulStateLen);
rv = C_GetOperationState(hSession, pState, &ulStateLen);

/* Continue hashing */
rv = C_DigestUpdate(hSession, data2, sizeof(data2));
assert(rv == CKR_OK);

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

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

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

    /* ... */
}

♦ C_Login

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

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

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

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

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

If the application calling C_Login has a R/O session open with the token, then it will be unable to log the SO into a session (see Section 6.7.7). An attempt to do this will result in the error code CKR_SESSION_READ_ONLY_EXISTS.

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

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

♦ C_Logout

```
CKDEFINE_FUNCTION(CK_RV, C_Logout) (
    CK_SESSION_HANDLE hSession
);  
```

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

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

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

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

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

Example:

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

rv = C_Login(hSession, CKU_USER, userPIN,
            sizeof(userPIN));
if (rv == CKR_OK) {
    
    rv = C_Logout(hSession);
    if (rv == CKR_OK) {
        
        }
    }
```
11.7 Object management functions

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

♦ C_CreateObject

```
CK(define_function(CK_RV, C_CreateObject),
   CK_SESSION_HANDLE hSession,
   CK_ATTRIBUTE_PTR pTemplate,
   CK_ULONG ulCount,
   CK_OBJECT_HANDLE_PTR phObject)
```

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

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

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

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

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

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hData,
   hCertificate,
   hKey;
CK_OBJECT_CLASS
dataClass = CKO_DATA,
certificateClass = CKO_CERTIFICATE,
keyClass = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_CHAR 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)},
    {CKA_VALUE, dataValue, sizeof(dataValue)}
};
CK_ATTRIBUTE certificateTemplate[] = {
    {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificateValue, sizeof(certificateValue)}
};
CK_ATTRIBUTE keyTemplate[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};
CK_RV rv;

/* Create a data object */
rv = C_CreateObject(hSession, &dataTemplate, 4, &hData);
if (rv == CKR_OK) {
    
}

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

♦ C_CopyObject

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

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

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

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

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

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

Example:

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

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

♦ **C_DestroyObject**

```c
CK_DEFINE_FUNCTION(CK_RV, C_DestroyObject)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject
);
```

**C_DestroyObject** destroys an object. *hSession* is the session’s handle; and *hObject* is the object’s handle.

Only session objects can be destroyed during a read-only session. Only public objects can be destroyed unless the normal user is logged in.
Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

Example: see C_GetObjectSize.

♦ C_GetObjectSize

```c
CK_DEFINE_FUNCTION(CK_RV, C_GetObjectSize)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ULONG_PTR pulSize
);
```

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

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

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

Example:

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

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

♦  **C_GetAttributeValue**

```c
CK_DEFINE_FUNCTION(CK_RV, C_GetAttributeValue)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);
```

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

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

1. If the specified attribute *(i.e., the attribute specified by the type field)* for the object cannot be revealed because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to hold the value -1 *(i.e., when it is cast to a CK_LONG, it holds -1)*.

2. Otherwise, if the specified attribute for the object is invalid *(the object does not possess such an attribute)*, then the *ulValueLen* field in that triple is modified to hold the value -1.

3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold the exact length of the specified attribute for the object.

4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified attribute for the object, then that attribute is copied into the buffer.
located at *pValue*, and the *ulValueLen* field is modified to hold the exact length of the attribute.

5. Otherwise, the *ulValueLen* field is modified to hold the value -1.

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

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

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

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

Example:

```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;

rv = C_GetAttributeValue(hSession, hObject, &template, 2);
if (rv == CKR_OK) {
    pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
    template[0].pValue = pModulus;
    // template[0].ulValueLen was set by C_GetAttributeValue */
    pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
    template[1].pValue = pExponent;
    // template[1].ulValueLen was set by C_GetAttributeValue */
    rv = C_GetAttributeValue(hSession, hObject, &template, 2);
    if (rv == CKR_OK) {
        free(pModulus);
        free(pExponent);
    }
}

♦ C_SetAttributeValue

CK_DEFINE_FUNCTION(CK_RV, C_SetAttributeValue)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hObject,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulCount
);

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

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

Not all attributes can be modified; see Section 9.7 for more details.

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_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:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hObject;
CK_UTF8CHAR label[] = {"New label");
CK_ATTRIBUTE template[] = {
    CKA_LABEL, label, sizeof(label)-1
};
CK_RV rv;
```

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

♦ C_FindObjectsInit

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

C_FindObjectsInit initializes a search for token and session objects that match a template. hSession is the session’s handle; pTemplate points to a search template that specifies the attribute values to match; ulCount is the number of attributes in the search template. The matching criterion is an exact byte-for-byte match with all attributes in the template. To find all objects, set ulCount to 0.
After calling \texttt{C\_FindObjectsInit}, the application may call \texttt{C\_FindObjects} one or more times to obtain handles for objects matching the template, and then eventually call \texttt{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 \texttt{C\_FindObjectsInit} can return the values \texttt{CKR\_ATTRIBUTE\_TYPE\_INVALID} and \texttt{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 \texttt{CKR\_ATTRIBUTE\_TYPE\_INVALID}, or it can initialize a search operation which will match no objects and return \texttt{CKR\_OK}.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_ATTRIBUTE\_TYPE\_INVALID}, \texttt{CKR\_ATTRIBUTE\_VALUE\_INVALID}, \texttt{CKR\_CRYPTOIKI\_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\_OK}, \texttt{CKR\_OPERATION\_ACTIVE}, \texttt{CKR\_PIN\_EXPIRED}, \texttt{CKR\_SESSION\_CLOSED}, \texttt{CKR\_SESSION\_HANDLE\_INVALID}.

Example: see \texttt{C\_FindObjectsFinal}.

\begin{verbatim}
♦ C\_FindObjects

CK\_DEFINE\_FUNCTION(CK\_RV, C\_FindObjects)(
    CK\_SESSION\_HANDLE hSession,
    CK\_OBJECT\_HANDLE\_PTR phObject,
    CK\_ULONG ulMaxObjectCount,
    CK\_ULONG\_PTR pulObjectCount
    );

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

If there are no more objects matching the template, then the location that pulObjectCount points to receives the value 0.
\end{verbatim}
The search must have been initialized with \texttt{C\_FindObjectInit}.

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 \texttt{C\_FindObjectFinal}.

\textbf{♦ \texttt{C\_FindObjectFinal}}

\begin{verbatim}
CK\_DEFINE\_FUNCTION(CK\_RV, C\_FindObjectFinal)(
   CK\_SESSION\_HANDLE hSession);
\end{verbatim}

\texttt{C\_FindObjectFinal} terminates a search for token and session objects. \texttt{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:

\begin{verbatim}
CK\_SESSION\_HANDLE hSession;
CK\_OBJECT\_HANDLE hObject;
CK\_ULONG ulObjectCount;
CK\_RV rv;

.
.
rv = C\_FindObjectInit(hSession, NULL\_PTR, 0);
assert(rv == CK\_OK);
while (1) {
   rv = C\_FindObject(hSession, &hObject, 1,
                  &ulObjectCount);
   if (rv != CK\_OK || ulObjectCount == 0)
      break;
   .
}

rv = C\_FindObjectFinal(hSession);
assert(rv == CK\_OK);
\end{verbatim}
11.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

♦ C_EncryptInit

```c
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. hSession is the session’s handle; pMechanism points to the encryption mechanism; hKey is the handle of the encryption key.

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

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

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_ERROR, CKRDEVICE_MEMORY, CKRDEVICE_REMOVED, CKRFUNCTION_CANCELED, CKRFUNCTION_FAILED, CKRGENERAL_ERROR, CKRHOST_MEMORY, CKRKEY_FUNCTION_NOT_PERMITTED, CKRKEY_HANDLE_INVALID, CKRKEY_SIZE_RANGE, CKRKEY_TYPE_INCONSISTENT, CKRMECHANISM_INVALID, CKRMECHANISM_PARAM_INVALID, CKR_OK, CKROPERATION_ACTIVE, CKRPIN_EXPIRED, CKRSESSION_CLOSED, CKRSESSION_HANDLE_INVALID, CKRUSER_NOT_LOGGED_IN.

Example: see C_EncryptFinal.
C_Encrypt

CK_DEFINE_FUNCTION(CK_RV, C_Encrypt)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pEncryptedData,
    CK_ULONG_PTR pulEncryptedDataLen
);

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

C_Encrypt uses the convention described in Section 11.2 on producing output.

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

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

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

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

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

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

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

C_EncryptUpdate continues a multiple-part encryption operation, processing another data part. hSession is the session’s handle; pPart points to the data part; ulPartLen is the length of the data part; pEncryptedPart points to the location that receives the encrypted data part; pulEncryptedPartLen points to the location that holds the length in bytes of the encrypted data part.

C_EncryptUpdate uses the convention described in Section 11.2 on producing output.

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

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

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

Example: see C_EncryptFinal.

C_EncryptFinal

C_EncryptFinal finishes a multiple-part encryption operation. hSession is the session’s handle; pLastEncryptedPart points to the location that receives the last encrypted data part, if any; pulLastEncryptedPartLen points to the location that holds the length of the last encrypted data part.
C_EncryptFinal uses the convention described in Section 11.2 on producing output.

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

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism’s input data must consist of an integral number of blocks. If these constraints are not satisfied, then C_EncryptFinal will fail with return code CKR_DATA_LEN_RANGE.

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

Example:

```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[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) {
  .
  .
}
11.9 Decryption functions

Cryptoki provides the following functions for decrypting data:

♦ C_DecryptInit

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

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

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

After calling C_DecryptInit, the application can either call C_Decrypt to decrypt data in a single part; or call C_DecryptUpdate zero or more times, followed by C_DecryptFinal, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to C_Decrypt or C_DecryptFinal to actually obtain the final piece of plaintext. To process additional data (in single or multiple parts), the application must call C_DecryptInit again.

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

Example: see C_DecryptFinal.
11. FUNCTIONS

♦ C_Decrypt

CK_DEFINE_FUNCTION(CK_RV, C_Decrypt)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedData,
    CK_ULONG ulEncryptedDataLen,
    CK_BYTE_PTR pData,
    CK_ULONG_PTR pulDataLen
);

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

C_Decrypt uses the convention described in Section 11.2 on producing output.

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

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

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

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

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

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

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

```c
CK DEFINE_FUNCTION (CK_RV, C_DecryptUpdate) (  
    CK_SESSION_HANDLE hSession,  
    CK_BYTE_PTR pEncryptedPart,  
    CK_ULONG ulEncryptedPartLen,  
    CK_BYTE_PTR pPart,  
    CK_ULONG_PTR pulPartLen
);
```

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

C_DecryptUpdate uses the convention described in Section 11.2 on producing output.

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

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

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

Example: See C_DecryptFinal.

C_DecryptFinal

```c
CK DEFINE_FUNCTION (CK_RV, C_DecryptFinal) (  
    CK_SESSION_HANDLE hSession,  
    CK_BYTE_PTR pLastPart,  
    CK_ULONG_PTR pulLastPartLen
);
```

C_DecryptFinal finishes a multiple-part decryption operation. **hSession** is the session’s handle; **pLastPart** points to the location that receives the last recovered data part, if any; **pulLastPartLen** points to the location that holds the length of the last recovered data part.
C_DecryptFinal uses the convention described in Section 11.2 on producing output.

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

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

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

Example:

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

CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
    CKM_DES_CBC_PAD, iv, sizeof(iv)
};
CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulData1Len, ulData2Len, ulData3Len;
CK_RV rv;
.

firstEncryptedPieceLen = 90;
secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ - firstEncryptedPieceLen;
rv = C_DecryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    /* Decrypt first piece */
    ulData1Len = sizeof(data);
    rv = C_DecryptUpdate(
        hSession,
        encryptedData[0], firstEncryptedPieceLen,
        &data,
```
&data[0], &ulData1Len);
if (rv != CKR_OK) {
  .
  .
}

/* Decrypt second piece */
ulData2Len = sizeof(data)-ulData1Len;
rv = C_DecryptUpdate(
    hSession,
    &encryptedData[firstEncryptedPieceLen],
    secondEncryptedPieceLen,
    &data[ulData1Len], &ulData2Len);
if (rv != CKR_OK) {
  .
  .
}

/* Get last little decrypted bit */
ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
rv = C_DecryptFinal(
    hSession,
    &data[ulData1Len+ulData2Len], &ulData3Len);
if (rv != CKR_OK) {
  .
  .
}

11.10 Message digesting functions

Cryptoki provides the following functions for digesting data:

♦ C_DigestInit

CK_DEFINE_FUNCTION(CK_RV, C_DigestInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism
);

C_DigestInit initializes a message-digesting operation. hSession is the session’s handle; pMechanism points to the digesting mechanism.

After calling C_DigestInit, the application can either call C_Digest to digest data in a single part; or call C_DigestUpdate zero or more times, followed by C_DigestFinal, to digest data in multiple parts. The message-digesting operation is active until the application uses a call to C_Digest or C_DigestFinal to actually obtain the message digest. To process additional data (in single or multiple parts), the application must call C_DigestInit again.
Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see C_DigestFinal.

♦ C_Digest

CK_DEFINE_FUNCTION(CK_RV, C_Digest)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pDigest,
    CK_ULONG_PTR pulDigestLen
);

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

C_Digest uses the convention described in Section 11.2 on producing output.

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

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

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

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

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

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

♦ **C_DigestUpdate**

```c
CK_DEFINE_FUNCTION(CK_RV, C_DigestUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen
);
```

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

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

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

Example: see **C_DigestFinal**.

♦ **C_DigestKey**

```c
CK_DEFINE_FUNCTION(CK_RV, C_DigestKey)(
    CK_SESSION_HANDLE hSession,
    CK_OBJECT_HANDLE hKey
);
```

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

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

If the value of the supplied key cannot be digested purely for some reason related to its length, **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.
Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example: see \texttt{C\_DigestFinal}.

\begin{center}
\tiny
\textbf{C\_DigestFinal}\par
\end{center}

\begin{verbatim}
CK_DEFINE_FUNCTION(CK_RV, C_DigestFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pDigest,
    CK_ULONG_PTR pulDigestLen
);
\end{verbatim}

\texttt{C\_DigestFinal} finishes a multiple-part message-digesting operation, returning the message digest. \textit{hSession} is the session’s handle; \textit{pDigest} points to the location that receives the message digest; \textit{pulDigestLen} points to the location that holds the length of the message digest.

\texttt{C\_DigestFinal} uses the convention described in Section 11.2 on producing output.

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

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

\begin{verbatim}
CK_SESSION_HANDLE hSession;
CK_MECHANISM mechanism = {
    CKM\_MD5, NULL\_PTR, 0
};
CK_BYTE data[] = { ... };
CK_BYTE digest[16];
CK_ULONG ulDigestLen;
CK_RV rv;
\end{verbatim}
rv = C_DigestInit(hSession, &mechanism);
if (rv != CKR_OK) {
    .
    .
}
rv = C_DigestUpdate(hSession, data, sizeof(data));
if (rv != CKR_OK) {
    .
    .
}
rv = C_DigestKey(hSession, hKey);
if (rv != CKR_OK) {
    .
    .
}
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);

11.11 Signing and MACing functions

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

♦ C_SignInit

```c
CK_DEFINE_FUNCTION(CK_RV, C_SignInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_SignInit initializes a signature operation, where the signature is an appendix to the data. hSession is the session’s handle; pMechanism points to the signature mechanism; hKey is the handle of the signature key.

The CKA_SIGN attribute of the signature key, which indicates whether the key supports signatures with appendix, must be CK_TRUE.

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

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

Example: see C_SignFinal.

♦ C_Sign

```c
CK_DEFINE_FUNCTION(CK_RV, C_Sign)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG_PTR pulSignatureLen
);
```

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

C_Sign uses the convention described in Section 11.2 on producing output.

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

C_Sign can not be used to terminate a multi-part operation, and must be called after C_SignInit without intervening C_SignUpdate calls.

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

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

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

♦ **C_SignUpdate**

```c
CK_DEFINE_FUNCTION(CK_RV, C_SignUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen
);
```

*C_SignUpdate* continues a multiple-part signature operation, processing another data part. *hSession* is the session’s handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

The signature operation must have been initialized with **C_SignInit**. This function may be called any number of times in succession. A call to **C_SignUpdate** which results in an error terminates the current signature operation.

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

Example: see **C_SignFinal**.

♦ **C_SignFinal**

```c
CK_DEFINE_FUNCTION(CK_RV, C_SignFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pSignature,
    CK_ULONG_PTR pulSignatureLen
);
```

*C_SignFinal* finishes a multiple-part signature operation, returning the signature. *hSession* is the session’s handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

*C_SignFinal* uses the convention described in Section 11.2 on producing output.

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

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

Example:

```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);
}
```

♦ C_SignRecoverInit

```c
CK_DEFINE_FUNCTION(CK_RV, C_SignRecoverInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
); 
```

C_SignRecoverInit initializes a signature operation, where the data can be recovered from the signature. hSession is the session’s handle; pMechanism points to the structure that specifies the signature mechanism; hKey is the handle of the signature key.
The **CKA_SIGN_RECOVER** attribute of the signature key, which indicates whether the key supports signatures where the data can be recovered from the signature, must be **CK_TRUE**.

After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The signature operation is active until the application uses a call to **C_SignRecover** to actually obtain the signature. To process additional data in a single part, the application must call **C_SignRecoverInit** again.


Example: see **C_SignRecover**.

**♦ C_SignRecover**

```c
CK_DEFINE_FUNCTION(CK_RV, C_SignRecover)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG_PTR pulSignatureLen
);
```

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

**C_SignRecover** uses the convention described in Section 11.2 on producing output.

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

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:

```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];
CKULONG ulSignatureLen;
CK_RV rv;
.
rv = C_SignRecoverInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    ulSignatureLen = sizeof(signature);
    rv = C_SignRecover(
        hSession, data, sizeof(data), signature,
        &ulSignatureLen);
    if (rv == CKR_OK) {
        
    }
}
```

11.12 Functions for verifying signatures and MACs

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

♦ C_VerifyInit

```c
CK_DEFINE_FUNCTION(CK_RV, C_VerifyInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

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

The CKA_VERIFY attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, must be CK_TRUE.
After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**. To process additional data (in single or multiple parts), the application must call **C_VerifyInit** again.

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

Example: see **C_VerifyFinal**.

♦ **C_Verify**

```c
CK_DEFINE_FUNCTION(CK_RV, C_Verify)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG ulSignatureLen
);
```

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

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

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

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

For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by **C_VerifyFinal**.
Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE.

Example: see C_VerifyFinal for an example of similar functions.

♦ C_VerifyUpdate

```
CK_DEFINE_FUNCTION(CK_RV, C_VerifyUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen
);
```

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

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

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

Example: see C_VerifyFinal.

♦ C_VerifyFinal

```
CK_DEFINE_FUNCTION(CK_RV, C_VerifyFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pSignature,
    CK_ULONG ulSignatureLen
);
```

C_VerifyFinal finishes a multiple-part verification operation, checking the signature. `hSession` is the session’s handle; `pSignature` points to the signature; `ulSignatureLen` is the length of the signature.
The verification operation must have been initialized with \texttt{C\_VerifyInit}. A call to \texttt{C\_VerifyFinal} always terminates the active verification operation.

A successful call to \texttt{C\_VerifyFinal} should return either the value \texttt{CKR\_OK} (indicating that the supplied signature is valid) or \texttt{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 \texttt{CKR\_SIGNATURE\_LEN\_RANGE} should be returned. In any of these cases, the active verifying operation is terminated.

Return values: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \texttt{CKR\_DATA\_LEN\_RANGE}, \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\_SIGNATURE\_INVALID}, \texttt{CKR\_SIGNATURE\_LEN\_RANGE}.

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));

}
```
♦ C_VerifyRecoverInit

```c
CK_DEFINE_FUNCTION(CK_RV, C_VerifyRecoverInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

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

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

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

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

Example: see C_VerifyRecover.

♦ C_VerifyRecover

```c
CK_DEFINE_FUNCTION(CK_RV, C_VerifyRecover)(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pSignature,
  CK_ULONG ulSignatureLen,
  CK_BYTE_PTR pData,
  CK_ULONG_PTR pulDataLen
);
```

C_VerifyRecover verifies a signature in a single-part operation, where the data is recovered from the signature. hSession is the session’s handle; pSignature points to the signature; ulSignatureLen is the length of the signature; pData points to the location that receives the recovered data; and pulDataLen points to the location that holds the length of the recovered data.
C_VerifyRecover uses the convention described in Section 11.2 on producing output.

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

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

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

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_MECHANISM mechanism = {
    CKM_RSA_9796, NULL_PTR, 0
};
CK_BYTE data[] = {...};
CK_ULONG ulDataLen;
CK_BYTE signature[128];
CK_RV rv;

...

rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    ulDataLen = sizeof(data);
    rv = C_VerifyRecover(
        hSession, signature, sizeof(signature), data,
        &ulDataLen);
    ...
    ...
}
```
11.13 Dual-function cryptographic functions

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

♦ **C_DigestEncryptUpdate**

```
CK_DEFINE_FUNCTION(CK_RV, C_DigestEncryptUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG_PTR pulEncryptedPartLen
);
```

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

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

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

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

Example:

```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));
meme(mset(data, 'A', ((2*BUF_SZ)+5));
rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
if (rv != CKR_OK) {
   .
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK) {
   .
}
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_DigestEncryptUpdate(
   hSession,
   &data[0], BUF_SZ,
   encryptedData, &ulEncryptedDataLen);
.
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_DigestEncryptUpdate(
   hSession,
   &data[BUF_SZ], BUF_SZ,
   encryptedData, &ulEncryptedDataLen);
.

/*
 * The last portion of the buffer needs to be handled
 with
 * separate calls to deal with padding issues in ECB mode
 */
/* First, complete the digest on the buffer */
rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
.
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
.

/* Then, pad last part with 3 0x00 bytes, and complete encryption */
for(i=0;i<3;i++)
    data[((BUF_SZ*2)+5)+i] = 0x00;

/* Now, get second-to-last piece of ciphertext */
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_EncryptUpdate(
    hSession,
    &data[BUF_SZ*2], 8,
    encryptedData, &ulEncryptedDataLen);
.

/* Get last piece of ciphertext (should have length 0, here) */
ulEncryptedDataLen = sizeof(encryptedData);
rv = C_EncryptFinal(hSession, encryptedData,
    &ulEncryptedDataLen);
.
.
♦ C_DecryptDigestUpdate

CK_DEFINE_FUNCTION(CK_RV, C_DecryptDigestUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG ulEncryptedPartLen,
    CK_BYTE_PTR pPart,
    CK_ULONG_PTR pulPartLen
);

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

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

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

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

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and digest the original plaintext thereby obtained.

After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES blocks) into C_DecryptDigestUpdate. C_DecryptDigestUpdate returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

Since there is no more ciphertext, the application calls C_DecryptFinal. This tells Cryptoki that there’s no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and digesting operations are linked only through the C_DecryptDigestUpdate call, these 2 bytes of plaintext are not passed on to be digested.

A call to C_DigestFinal, therefore, would compute the message digest of the first 16 bytes of the plaintext, not the message digest of the entire plaintext. It is crucial that, before C_DigestFinal is called, the last 2 bytes of plaintext get passed into the active digesting operation via a C_DigestUpdate call.

Because of this, it is critical that when an application uses a padded decryption mechanism with C_DecryptDigestUpdate, it knows exactly how much plaintext has been passed into the active digesting operation. Extreme caution is warranted when using a padded decryption mechanism with C_DecryptDigestUpdate.

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

Example:

```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_ULONG ulDigestLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;

memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
if (rv != CKR_OK) {
   .
}
rv = C_DigestInit(hSession, &digestMechanism);
if (rv != CKR_OK) {
   .
}
ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
   hSession,
   &encryptedData[0], BUF_SZ,
   data, &ulDataLen);
   .
```
ulDataLen = sizeof(data);
rv = C_DecryptDigestUpdate(
    hSession,
    &encryptedData[BUF_SZ], BUF_SZ,
    data, &ulDataLen);
.
.
/*
 * The last portion of the buffer needs to be handled
 * with
 * separate calls to deal with padding issues in ECB mode
 */

/* First, complete the decryption of the buffer */
ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate(
    hSession,
    &encryptedData[BUF_SZ*2], 8,
    data, &ulLastUpdateSize);
.
.
/* Get last piece of plaintext (should have length 0, here) */
ulDataLen = sizeof(data)-ulLastUpdateSize;
rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize],
    &ulDataLen);
if (rv != CKR_OK) {
    .
.
}

/* Digest last bit of plaintext */
rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
if (rv != CKR_OK) {
    .
.
}
ulDigestLen = sizeof(digest);
rv = C_DigestFinal(hSession, digest, &ulDigestLen);
if (rv != CKR_OK) {
    .
.
}
C_SignEncryptUpdate 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 pulEncryptedPart points to the location that holds the length of the encrypted data part.

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

Signature and encryption operations must both be active (they must have been initialized with C_SignInit and C_EncryptInit, respectively). This function may be called any number of times in succession, and may be interspersed with C_SignUpdate and C_EncryptUpdate calls.

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

Example:

```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);

♦ C_DecryptVerifyUpdate

CK_DEFINE_FUNCTION(CK_RV, C_DecryptVerifyUpdate)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG ulEncryptedPartLen,
    CK_BYTE_PTR pPart,
    CK_ULONG_PTR pulPartLen
);

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

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

Decryption and signature operations must both be active (they must have been initialized with C_DecryptInit and C_VerifyInit, respectively). This function may be called any
number of times in succession, and may be interspersed with C_DecryptUpdate and C_VerifyUpdate calls.

Use of C_DecryptVerifyUpdate involves a pipelining issue that does not arise when using C_SignEncryptUpdate, the “inverse function” of C_DecryptVerifyUpdate. This is because when C_SignEncryptUpdate is called, precisely the same input is passed to both the active signing operation and the active encryption operation; however, when C_DecryptVerifyUpdate is called, the input passed to the active verifying operation is the output of the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and verify a signature on the original plaintext thereby obtained.

After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3 DES blocks) into C_DecryptVerifyUpdate. C_DecryptVerifyUpdate returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

Since there is no more ciphertext, the application calls C_DecryptFinal. This tells Cryptoki that there’s no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and verification operations are linked only through the C_DecryptVerifyUpdate call, these 2 bytes of plaintext are not passed on to the verification mechanism.

A call to C_VerifyFinal, therefore, would verify whether or not the signature supplied is a valid signature on the first 16 bytes of the plaintext, not on the entire plaintext. It is crucial that, before C_VerifyFinal is called, the last 2 bytes of plaintext get passed into the active verification operation via a C_VerifyUpdate call.

Because of this, it is critical that when an application uses a padded decryption mechanism with C_DecryptVerifyUpdate, it knows exactly how much plaintext has been passed into the active verification operation. Extreme caution is warranted when using a padded decryption mechanism with C_DecryptVerifyUpdate.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKRDEVICE_REMOVED, CKR_ENCRYPTEDDATA_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 hDecryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM decryptionMechanism = {
    CKM_DES_ECB, iv, sizeof(iv)
};
CK_MECHANISM verifyMechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_BYTE encryptedData[(2*BUF_SZ)+8];
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[BUF_SZ];
CK_ULONG ulDataLen, ulLastUpdateSize;
CK_RV rv;
.
.
memset(iv, 0, sizeof(iv));
memset(encryptedData, 'A', ((2*BUF_SZ)+8));
rv = C_DecryptInit(hSession, &decryptionMechanism,
    hDecryptionKey);
if (rv != CKR_OK) {
    .
    .
}
rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
if (rv != CKR_OK){
    .
    .
}
ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
    hSession,
    &encryptedData[0], BUF_SZ,
    data, &ulDataLen);
.
.
ulDataLen = sizeof(data);
rv = C_DecryptVerifyUpdate(
    hSession,
    &encryptedData[BUF_SZ], BUF_SZ,
    data, &uldataLen);
.
.
```
/ * The last portion of the buffer needs to be handled with separate calls to deal with padding issues in ECB mode */

/* First, complete the decryption of the buffer */ ulLastUpdateSize = sizeof(data);
rv = C_DecryptUpdate( hSession, &encryptedData[BUF_SZ*2], 8, data, &ulLastUpdateSize);
.
.
/* Get last little piece of plaintext. Should have length 0 */ ulDataLen = sizeof(data)-ulLastUpdateSize;
rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
if (rv != CKR_OK) {
  .
  .
}

/* Send last bit of plaintext to verification operation */ rv = C_VerifyUpdate(hSession, &data[BUF_SZ*2], 5);
if (rv != CKR_OK) {
  .
  .
}
rv = C_VerifyFinal(hSession, MAC, ulMacLen);
if (rv == CKR_SIGNATURE_INVALID) {
  .
  .
}

11.14 Key management functions

Cryptoki provides the following functions for key management:
C_GenerateKey generates a secret key or set of domain parameters, creating a new object. *hSession* is the session’s handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey* points to the location that receives the handle of the new key or set of domain parameters.

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

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

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

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

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

Example:

```c
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
```
CK_MECHANISM mechanism = {
    CKM_DES_KEY_GEN, NULL_PTR, 0
};
CK_RV rv;
.
.
rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0,
    &hKey);
if (rv == CKR_OK) {
    
    
}

♦ C_GenerateKeyPair

CK_DEFINE_FUNCTION(CK_RV, C_GenerateKeyPair)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_ATTRIBUTE_PTR pPublicKeyTemplate,
    CK_ULONG ulPublicKeyAttributeCount,
    CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
    CK_ULONG ulPrivateKeyAttributeCount,
    CK_OBJECT_HANDLE_PTR phPublicKey,
    CK_OBJECT_HANDLE_PTR phPrivateKey
);

C_GenerateKeyPair generates a public/private key pair, creating new key objects. hSession is the session’s handle; pMechanism points to the key generation mechanism; pPublicKeyTemplate points to the template for the public key; ulPublicKeyAttributeCount is the number of attributes in the public-key template; pPrivateKeyTemplate points to the template for the private key; ulPrivateKeyAttributeCount is the number of attributes in the private-key template; phPublicKey points to the location that receives the handle of the new public key; phPrivateKey points to the location that receives the handle of the new private key.

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, C_GenerateKeyPair fails and returns the error code CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

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

A call to C_GenerateKeyPair will never create just one key and return. A call can fail, and create no keys; or it can succeed, and create a matching public/private key pair.
The key objects created by a successful call to \texttt{C\_GenerateKeyPair} will have their \texttt{CKA\_LOCAL} attributes set to \texttt{CK\_TRUE}.

\textit{Note carefully the order of the arguments to \texttt{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: \texttt{CKR\_ARGUMENTS\_BAD}, \texttt{CKR\_ATTRIBUTE\_READ\_ONLY}, \texttt{CKR\_ATTRIBUTE\_TYPE\_INVALID}, \texttt{CKR\_ATTRIBUTE\_VALUE\_INVALID}, \texttt{CKR\_CRYPTOKI\_NOT\_INITIALIZED}, \texttt{CKR\_DEVICE\_ERROR}, \texttt{CKR\_DEVICE\_MEMORY}, \texttt{CKR\_DEVICE\_REMOVED}, \texttt{CKR\_DOMAIN\_PARAMS\_INVALID}, \texttt{CKR\_FUNCTION\_CANCELED}, \texttt{CKR\_FUNCTION\_FAILED}, \texttt{CKR\_GENERAL\_ERROR}, \texttt{CKR\_HOST\_MEMORY}, \texttt{CKR\_MECHANISM\_INVALID}, \texttt{CKR\_MECHANISM\_PARAM\_INVALID}, \texttt{CKR\_OK}, \texttt{CKR\_OPERATION\_ACTIVE}, \texttt{CKR\_PIN\_EXPIRED}, \texttt{CKR\_SESSION\_CLOSED}, \texttt{CKR\_SESSION\_HANDLE\_INVALID}, \texttt{CKR\_SESSION\_READ\_ONLY}, \texttt{CKR\_TEMPLATE\_INCOMPLETE}, \texttt{CKR\_TEMPLATE\_INCONSISTENT}, \texttt{CKR\_TOKEN\_WRITE\_PROTECTED}, \texttt{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 = 768;
CK\_BYTE publicExponent[] = { 3 };
CK\_BYTE subject[] = {...};
CK\_BYTE id[] = {123};
CK\_BBBOOL true = CK\_TRUE;
CK\_ATTRIBUTE publicKeyTemplate[] = {
    {CKA\_ENCRYPT, &true, sizeof(true)},
    {CKA\_VERIFY, &true, sizeof(true)},
    {CKA\_WRAP, &true, sizeof(true)},
    {CKA\_MODULUS\_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA\_PUBLIC\_EXPONENT, publicExponent, sizeof(publicExponent)}
};
CK\_ATTRIBUTE privateKeyTemplate[] = {
    {CKA\_TOKEN, &true, sizeof(true)},
    {CKA\_PRIVATE, &true, sizeof(true)},
    {CKA\_SUBJECT, subject, sizeof(subject)},
    {CKA\_ID, id, sizeof(id)},
    {CKA\_SENSITIVE, &true, sizeof(true)},
    {CKA\_DECRYPT, &true, sizeof(true)},
    {CKA\_SIGN, &true, sizeof(true)},
    {CKA\_UNWRAP, &true, sizeof(true)}
};
```
CK_RV rv;

rv = C_GenerateKeyPair(
    hSession, &mechanism, 
    publicKeyTemplate, 5, 
    privateKeyTemplate, 8, 
    &hPublicKey, &hPrivateKey);
if (rv == CKR_OK) {

}

♦ C_WrapKey

CK_DEFINE_FUNCTION(CK_RV, C_WrapKey) (  
    CK_SESSION_HANDLE hSession,  
    CK_MECHANISM_PTR pMechanism,  
    CK_OBJECT_HANDLE hWrappingKey,  
    CK_OBJECT_HANDLE hKey,  
    CK_BYTE_PTR pWrappedKey,  
    CK_ULONG_PTR pulWrappedKeyLen  
);

C_WrapKey wraps (i.e., encrypts) a private or secret key. hSession is the session’s handle; pMechanism points to the wrapping mechanism; hWrappingKey is the handle of the wrapping key; hKey is the handle of the key to be wrapped; pWrappedKey points to the location that receives the wrapped key; and pulWrappedKeyLen points to the location that receives the length of the wrapped key.

C_WrapKey uses the convention described in Section 11.2 on producing output.

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

If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its CKA_EXTRACTABLE attribute set to CK_TRUE, then C_WrapKey fails with error code CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism solely because of its length, then C_WrapKey fails with error code CKR_KEY_SIZE_RANGE.

C_WrapKey can be used in the following situations:

• To wrap any secret key with a public key that supports encryption and decryption.

• To wrap any secret key with any other secret key. Consideration must be given to key size and mechanism strength or the token may not allow the operation.

• To wrap a private key with any secret key.
Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

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

Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE.Removed, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE, CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

Example:

```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,
    wrappedKey, &ulWrappedKeyLen);
if (rv == CKR_OK) {
    .
    .
```
C_UnwrapKey

CK_DEFINE_FUNCTION(CK_RV, C_UnwrapKey)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hUnwrappingKey,
    CK_BYTE_PTR pWrappedKey,
    CK_ULONG ulWrappedKeyLen,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulAttributeCount,
    CK_OBJECT_HANDLE_PTR phKey
);

C_UnwrapKey unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object. hSession is the session’s handle; pMechanism points to the unwrapping mechanism; hUnwrappingKey is the handle of the unwrapping key; pWrappedKey points to the wrapped key; ulWrappedKeyLen is the length of the wrapped key; pTemplate points to the template for the new key; ulAttributeCount is the number of attributes in the template; phKey points to the location that receives the handle of the recovered key.

The CKA_UNWRAP attribute of the unwrapping key, which indicates whether the key supports unwrapping, must be CK_TRUE.

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

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

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

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

To partition the unwrapping keys so they can only unwrap a subset of keys the attribute CKA_UNWRAP_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied attribute template, in ‘pTemplate’, then the unwrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute conflict occurs on an attempt to unwrap a key then the function shall return CKR_TEMPLATE_INCONSISTENT.
Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKRGENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.

Example:

```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;
.
rv = C_UnwrapKey(
    hSession, &mechanism, hUnwrappingKey, 
    wrappedKey, sizeof(wrappedKey), template, 4, &hKey);
if (rv == CKR_OK) {
    .
    .
```
C_DeriveKey

CK_DEFINE_FUNCTION(CK_RV, C_DeriveKey)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hBaseKey,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulAttributeCount,
    CK_OBJECT_HANDLE_PTR phKey
);

C_DeriveKey derives a key from a base key, creating a new key object. hSession is the session’s handle; pMechanism points to a structure that specifies the key derivation mechanism; hBaseKey is the handle of the base key; pTemplate points to the template for the new key; ulAttributeCount is the number of attributes in the template; and phKey points to the location that receives the handle of the derived key.

The values of the CK_SENSITIVE, CK_ALWAYS_SENSITIVE, CK_EXTRACTABLE, and CK_NEVER_EXTRACTABLE attributes for the base key affect the values that these attributes can hold for the newly-derived key. See the description of each particular key-derivation mechanism in Section 11.17.2 for any constraints of this type.

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

    CK_SESSION_HANDLE hSession;
    CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;
    CK_MECHANISM keyPairMechanism = {
CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE publicValue[128];
CK_BYTE otherPublicValue[128];
CK_MECHANISM mechanism = {
    CKM_DH_PKCS_DERIVE, otherPublicValue,
    sizeof(otherPublicValue)
};
CK_ATTRIBUTE pTemplate[] = {
    CKA_VALUE, &publicValue, sizeof(publicValue)}
};
CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE publicKeyTemplate[] = {
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)}
};
CK_ATTRIBUTE privateKeyTemplate[] = {
    {CKA_DERIVE, &true, sizeof(true)}
};
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)}},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)}},
    {CKA_ENCRYPT, &true, sizeof(true)}},
    {CKA_DECRYPT, &true, sizeof(true)}
};
CK_RV rv;

rv = C_GenerateKeyPair(
    hSession, &keyPairMechanism,
    publicKeyTemplate, 2,
    privateKeyTemplate, 1,
    &hPublicKey, &hPrivateKey);
if (rv == CKR_OK) {
    rv = C_GetAttributeValue(hSession, hPublicKey,
        &pTemplate, 1);
    if (rv == CKR_OK) {
        /* Put other guy’s public value in otherPublicValue */
        rv = C_DeriveKey(
            hSession, &mechanism,
            hPrivateKey, template, 4, &hKey);
        if (rv == CKR_OK) {

11.15 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

♦  C_SeedRandom

```c
CK_DEFINE_FUNCTION(CK_RV, C_SeedRandom)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pSeed,
    CK_ULONG ulSeedLen
);
```

C_SeedRandom mixes additional seed material into the token’s random number generator. hSession is the session’s handle; pSeed points to the seed material; and ulSeedLen is the length in bytes of the seed material.

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

Example: see C_GenerateRandom.

♦  C_GenerateRandom

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

C_GenerateRandom generates random or pseudo-random data. hSession is the session’s handle; pRandomData points to the location that receives the random data; and ulRandomLen is the length in bytes of the random or pseudo-random data to be generated.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKRDEVICE_ERROR, CKRDEVICE_MEMORY, CKRDEVICE_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:

```c
CK_SESSION_HANDLE hSession;
CK_BYTE seed[] = {...};
CK_BYTE randomData[] = {...};
CK_RV rv;

rv = C_SeedRandom(hSession, seed, sizeof(seed));
if (rv != CKR_OK) {
    ...
}
rv = C_GenerateRandom(hSession, randomData,
                      sizeof(randomData));
if (rv == CKR_OK) {
    ...
}
```

11.16  Parallel function management functions

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

- **C_GetFunctionStatus**

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

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

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

```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**.


11.17 Callback functions

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

11.17.1 Surrender callbacks

Cryptographic functions (i.e., any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session’s callback with the arguments `(hSession, CKN_SURRENDER, pApplication)`, where `hSession` is the session’s handle and `pApplication` was supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the value **CKR_OK** (to indicate that Cryptoki should continue executing the function) or the value **CKR_CANCEL** (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation, if desired.

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

A Cryptoki library is not **required** to make any surrender callbacks.
11.17.2 Vendor-defined callbacks

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

A mechanism specifies precisely how a certain cryptographic process is to be performed.

The following table shows which Cryptoki mechanisms are supported 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 which supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation). For example, even if a token is able to create RSA digital signatures with the \texttt{CKM\_RSA\_PKCS} mechanism, it may or may not be the case that the same token can also perform RSA encryption with \texttt{CKM\_RSA\_PKCS}.

Table 34, 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_RSA_PKCS_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_X9_31_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_PKCS</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_PKCS_OAEP</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_PKCS_PSS</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_9796</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_X_509</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RSA_X9_31</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_MD2_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_MD5_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA1_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA256_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD128_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_RIPEMD160_RSA_PKCS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA1_RSA_PKCS_PSS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA256_RSA_PKCS_PSS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA384_RSA_PKCS_PSS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA512_RSA_PKCS_PSS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_SHA1_RSA_X9_31</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_KEY_PAIR_GEN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_PARAMETER_GEN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DSA</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DSA_SHA1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_FORTEZZA_TIMESTAMP</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_EC_KEY_PAIR_GEN</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>(CKM_ECDSA_KEY_PAIR_GEN)</td>
<td>✓²</td>
<td>✓²</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_ECDSA</td>
<td>✓²</td>
<td>✓²</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_ECDH1_DERIVE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2004 RSA Security Inc.  June 2004
<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_ECDH1_COFACTOR_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_ECMQV_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_DH_PKCS_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DH_PKCS_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DH_PKCS_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_X9_42_DH_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X9_42_DH_PKCS_PARAMETER_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X9_42_DH_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X9_42_DH_HYBRID_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_X9_42_MQV_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_KEA_KEY_PAIR_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_KEA_KEY_DERIVE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_GENERIC_SECRET_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC2_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC4_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC4</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC5_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES2_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Functions</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_CAST_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST3_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST3_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST3_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST3_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>(CKM_CAST5_KEY_GEN)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_ECB (CKM_CAST5_ECB)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_CBC (CKM_CAST5_CBC)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_CBC_PAD (CKM_CAST5_CBC_PAD)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_MAC_GENERAL (CKM_CAST5_MAC_GENERAL)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CAST128_MAC (CKM_CAST5_MAC)</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_IDEA_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_ECB</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_CBC_PAD</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_MAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_CDMF_MAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_ECB_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DES3_CBC_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_ECB_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_AES_CBC_ENCRYPT_DATA</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_KEY_GEN</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_ECB64</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_CBC64</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_OFB64</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_CFB64</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_CFB32</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_CFB16</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_CFB8</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_WRAP</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_PRIVATE_WRAP</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SKIPJACK_RELAYX</td>
<td>✓</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Functions</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td></td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>CKM_BATON_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_BATON_ECB128</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BATON_ECB96</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BATON_CBC128</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BATON_COUNTER</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BATON_SHUFFLE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_BATON_WRAP</td>
<td></td>
</tr>
<tr>
<td>CKM_JUNIPER_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_JUNIPER_ECB128</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_JUNIPER_CBC128</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_JUNIPER_COUNTER</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_JUNIPER_SHUFFLE</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_JUNIPER_WRAP</td>
<td></td>
</tr>
<tr>
<td>CKM_MD2</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD2_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD2_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD2_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD5</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD5_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD5_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_MD5_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA1_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA256_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA384_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA512_KEY_DERIVATION</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD128</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD128_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD128_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD160</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD160_HMAC_GENERAL</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RIPEMD160_HMAC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_FASTHASH</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_PBE_MD2_DES_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_PBE_MD5_DES_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_PBE_MD5ICAST_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>CKM_PBE_MD5ICAST3_CBC</td>
<td>✓</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Encrypt &amp; Decrypt</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>CKM_PBE_MD5_CAST128_CBC</td>
<td></td>
</tr>
<tr>
<td>(CKM_PBE_MD5_CAST5_CBC)</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_CAST128_CBC</td>
<td></td>
</tr>
<tr>
<td>(CKM_PBE_SHA1_CAST5_CBC)</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC4_128</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC4_40</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_DES3_EDE_CBC</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_DES2_EDE_CBC</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC2_128_CBC</td>
<td></td>
</tr>
<tr>
<td>CKM_PBE_SHA1_RC2_40_CBC</td>
<td></td>
</tr>
<tr>
<td>CKM_PBA_SHA1_WITH_SHA1_HMAC</td>
<td></td>
</tr>
<tr>
<td>CKM_PKCS5_PBKD2</td>
<td></td>
</tr>
<tr>
<td>CKM_KEY_WRAP_SET_OAEP</td>
<td></td>
</tr>
<tr>
<td>CKM_KEY_WRAP_LYNKS</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_PRE_MASTER_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_MASTER_KEY_DERIVE_DH</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_MD5_MAC</td>
<td></td>
</tr>
<tr>
<td>CKM_SSL3_SHA1_MAC</td>
<td></td>
</tr>
<tr>
<td>CKM_TLS_PRE_MASTER_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_TLS_MASTER_KEY_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_TLS_MASTER_KEY_DERIVE_DH</td>
<td></td>
</tr>
<tr>
<td>CKM_TLS_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_TLS_PRF</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_PRE_MASTER_KEY_GEN</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE</td>
<td></td>
</tr>
<tr>
<td>CKM_WTLS_PRF</td>
<td></td>
</tr>
<tr>
<td>CKM_CMS_SIG</td>
<td></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>

¹ SR = SignRecover, VR = VerifyRecover.
² Single-part operations only.
³ Mechanism can only be used for wrapping, not unwrapping.

The remainder of this section will present in detail the mechanisms supported by Cryptoki and the parameters which are supplied to them.

In general, if a mechanism makes no mention of the ulMinKeyLen and ulMaxKeyLen fields of the CK_MECHANISM_INFO structure, then those fields have no meaning for that particular mechanism.
12.1 RSA

12.1.1 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:

```plaintext
CKM_RSA_PKCS_KEY_PAIR_GEN
CKM_RSA_PKCS
CKM_RSA_9796
CKM_RSA_X_509
CKM_MD2_RSA_PKCS
CKM_MD5_RSA_PKCS
CKM_SHA1_RSA_PKCS
CKM_SHA256_RSA_PKCS
CKM_SHA384_RSA_PKCS
CKM_SHA512_RSA_PKCS
CKM_RIPEMD128_RSA_PKCS
CKM_RIPEMD160_RSA_PKCS
CKM_RSA_PKCS_OAEP
CKM_RSA_X9_31_KEY_PAIR_GEN
CKM_RSA_X9_31
CKM_SHA1_RSA_X9_31
CKM_RSA_PKCS_PSS
CKM_SHA1_RSA_PKCS_PSS
CKM_SHA256_RSA_PKCS_PSS
CKM_SHA512_RSA_PKCS_PSS
CKM_SHA384_RSA_PKCS_PSS
```

12.1.2 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 35, RSA Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_MODULEUSD1,4</td>
<td>Big integer</td>
<td>Modulus $n$</td>
</tr>
<tr>
<td>CKA_MODULEUS_BITS2,3</td>
<td>CK_ULONG</td>
<td>Length in bits of modulus $n$</td>
</tr>
<tr>
<td>CKA_PUBLIC_EXPONENT1</td>
<td>Big integer</td>
<td>Public exponent $e$</td>
</tr>
</tbody>
</table>

*Refer to table Table 15 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:

```c
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)}
};
```

### 12.1.3 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>CKA_MODULUS&lt;sup&gt;1,4,6&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Modulus n</td>
</tr>
<tr>
<td>CKA_PUBLIC_EXPONENT&lt;sup&gt;4,6&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Public exponent e</td>
</tr>
<tr>
<td>CKA_PRIVATE_EXPONENT&lt;sup&gt;1,4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Private exponent d</td>
</tr>
<tr>
<td>CKA_PRIME_1&lt;sup&gt;4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime p</td>
</tr>
<tr>
<td>CKA_PRIME_2&lt;sup&gt;4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime q</td>
</tr>
<tr>
<td>CKA_EXPONENT_1&lt;sup&gt;4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Private exponent d modulo p-1</td>
</tr>
<tr>
<td>CKA_EXPONENT_2&lt;sup&gt;4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Private exponent d modulo q-1</td>
</tr>
<tr>
<td>CKA_COEFFICIENT&lt;sup&gt;4,6,7&lt;/sup&gt;</td>
<td>Big integer</td>
<td>CRT coefficient q&lt;sup&gt;-1&lt;/sup&gt; mod p</td>
</tr>
</tbody>
</table>

<sup>1</sup> Refer to table Table 15 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.
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 36 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 36 for which a Cryptoki
implementation is required to be able to return values are CKA_MODULUS and
CKA_PRIVATE_EXPONENT.

If an RSA private key object is created on a token, and more attributes from Table 36 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)}
},
```
12.1.4 PKCS #1 RSA key pair generation

The PKCS #1 RSA key pair generation mechanism, denoted 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 CKA_MODULUS_BITS and CKA_PUBLIC_EXPONENT attributes of the template for the public key. The 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 CKA_PUBLIC_EXPONENT attribute to be omitted from the template, and behaved as described above. The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, CKA_MODULUS, and CKA_PUBLIC_EXPONENT attributes to the new public key. CKA_PUBLIC_EXPONENT will be copied from the template if supplied. CKR_TEMPLATE_INCONSISTENT shall be returned if the implementation cannot use the supplied exponent value. It contributes the CKA_CLASS and CKA_KEY_TYPE attributes to the new private key; it may also contribute some of the following attributes to the new private key: CKA_MODULUS, CKA_PUBLIC_EXPONENT, CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, 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 *ulMinKeySize* and *ulMaxKeySize* fields of the `CK_MECHANISM_INFO` structure specify the supported range of RSA modulus sizes, in bits.

### 12.1.5 X9.31 RSA key pair generation

The X9.31 RSA key pair generation mechanism, denoted `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 `CKA_MODULUS_BITS` and `CKA_PUBLIC_EXPONENT` attributes of the template for the public key.

The mechanism contributes the `CKA_CLASS`, `CKA_KEY_TYPE`, `CKA_MODULUS`, and `CKA_PUBLIC_EXPONENT` attributes to the new public key. It contributes the `CKA_CLASS` and `CKA_KEY_TYPE` attributes to the new private key; it may also contribute some of the following attributes to the new private key: `CKA_MODULUS`, `CKA_PUBLIC_EXPONENT`, `CKA_PRIVATE_EXPONENT`, `CKA_PRIME_1`, `CKA_PRIME_2`, `CKA_EXPONENT_1`, `CKA_EXPONENT_2`, `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 `CKM_RSA_PKCS_KEY_PAIR_GEN` mechanism, this mechanism is guaranteed to generate $p$ and $q$ values, `CKA_PRIME_1` and `CKA_PRIME_2` respectively, that meet the strong primes requirement of X9.31.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the `CK_MECHANISM_INFO` structure specify the supported range of RSA modulus sizes, in bits.

### 12.1.6 PKCS #1 v1.5 RSA

The PKCS #1 v1.5 RSA mechanism, denoted `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>
<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 01</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>( \leq k-11, k^2 )</td>
<td>N/A</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 01</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.

12.1.7 PKCS #1 RSA OAEP mechanism parameters

\textbf{CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR}

The \texttt{CK_RSA_PKCS_MGF_TYPE} is used to indicate the Message 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:

\[
\text{typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;}
\]
The following MGFs are defined in PKCS #1. The following table lists the defined functions.

Table 38, 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>0x00000001</td>
</tr>
<tr>
<td>CKG_MGF1_SHA256</td>
<td>0x00000002</td>
</tr>
<tr>
<td>CKG_MGF1_SHA384</td>
<td>0x00000003</td>
</tr>
<tr>
<td>CKG_MGF1_SHA512</td>
<td>0x00000004</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 39, 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>0x00000001</td>
<td>Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, pSourceData must be NULL and ulSourceDataLen 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:

```c
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.

### 12.1.8 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 \( \text{hashAlg} \) field of the \text{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(^1)</td>
<td>RSA public key</td>
<td>( \leq k-2-2hLen )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Decrypt(^1)</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 \text{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

### 12.1.9 PKCS #1 RSA PSS mechanism parameters

- **\text{CK_RSA_PKCS_PSS_PARAMS}; \text{CK_RSA_PKCS_PSS_PARAMS_PTR}**

\text{CK_RSA_PKCS_PSS_PARAMS} is a structure that provides the parameters to the \text{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:

- \( \text{hashAlg} \) 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

\textbf{CK_RSA_PKCS_PSS_PARAMS_PTR} is a pointer to a \textbf{CK_RSA_PKCS_PSS_PARAMS}.

\section*{12.1.10 PKCS #1 RSA PSS}

The PKCS #1 RSA PSS mechanism, denoted \textbf{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 \textbf{CK_RSA_PKCS_PSS_PARAMS} structure. The \textit{sLen} field must be less than or equal to \( k^* -2-hLen \) and \textit{hLen} is the length of the input to the \textit{C_Sign} or \textit{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.

\textbf{Table 41, PKCS #1 RSA PSS: 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>\textit{hLen}</td>
<td>\textit{k}</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>RSA public key</td>
<td>\textit{hLen, k}</td>
<td>N/A</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 \textbf{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.
12.1.11 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$^\dagger$</td>
<td>RSA private key</td>
<td>$\leq \lceil k/2 \rceil$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_SignRecover</td>
<td>RSA private key</td>
<td>$\leq \lceil k/2 \rceil$</td>
<td>$k$</td>
</tr>
<tr>
<td>C_Verify$^\dagger$</td>
<td>RSA public key</td>
<td>$\leq \lceil k/2 \rceil, k^2$</td>
<td>N/A</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>RSA public key</td>
<td>$k$</td>
<td>$\leq \lceil k/2 \rceil$</td>
</tr>
</tbody>
</table>

$^\dagger$ 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.

12.1.12 X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted 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 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 $P=2^n b_1+2^{n-1} b_2+\ldots+b_n$. 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 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 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 43, 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 \(ulMinKeySize\) and \(ulMaxKeySize\) fields of the \textbf{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.

### 12.1.13 ANSI X9.31 RSA

The ANSI X9.31 RSA mechanism, denoted \textbf{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>
<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>( \leq k-2 )</td>
<td>( k )</td>
</tr>
<tr>
<td>C_Verify</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 \texttt{CK_MECHANISM_INFO} structure specify the supported range of RSA modulus sizes, in bits.

12.1.14 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 \texttt{CKM_MD2_RSA_PKCS}, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described initially in PKCS #1 v1.5 with the object identifier md2WithRSAEncryption, and as in the scheme RSASSA-PKCS1-v1_5 in the current version of PKCS #1, where the underlying hash function is MD2.

Similarly, the PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted \texttt{CKM_MD5_RSA_PKCS}, performs the same operations described in PKCS #1 with the object identifier md5WithRSAEncryption. The PKCS #1 v1.5 RSA signature with SHA-1 mechanism, denoted \texttt{CKM_SHA1_RSA_PKCS}, performs the same operations, except that it uses the hash function SHA-1 with object identifier sha1WithRSAEncryption.

Likewise, the PKCS #1 v1.5 RSA signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted \texttt{CKM_SHA256_RSA_PKCS}, \texttt{CKM_SHA384_RSA_PKCS}, and \texttt{CKM_SHA512_RSA_PKCS} respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions with the object identifiers sha256WithRSAEncryption, sha384WithRSAEncryption and sha384WithRSAEncryption respectively.
The PKCS #1 v1.5 RSA signature with RIPEMD-128 or RIPEMD-160, denoted \texttt{CKM\_RIPEMD128\_RSA\_PKCS} and \texttt{CKM\_RIPEMD160\_RSA\_PKCS} respectively, perform the same operations using the RIPE-MD 128 and RIPE-MD 160 hash functions.

None of these mechanisms has a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, \( k \) is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms, \( k \) must be at least 27; for the PKCS #1 v1.5 RSA signature with SHA-1 mechanism, \( k \) must be at least 31, and so on for other underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

<table>
<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_Sign</td>
<td>RSA private key</td>
<td>any</td>
<td>( k )</td>
<td>block type 01</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RSA public key</td>
<td>any, ( k^2 )</td>
<td>N/A</td>
<td>block type 01</td>
</tr>
</tbody>
</table>

\(^2\) Data length, signature length.

For these mechanisms, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of RSA modulus sizes, in bits.

### 12.1.15 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 \texttt{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 \texttt{CKM\_SHA256\_RSA\_PKCS\_PSS}, \texttt{CKM\_SHA384\_RSA\_PKCS\_PSS}, and \texttt{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 \texttt{CK\_RSA\_PKCS\_PSS\_PARAMS} structure. The \texttt{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 46, PKCS #1 RSA PSS Signatures with Various Hash Functions: 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>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>

\(^2\) 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.

**12.1.16 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 47, ANSI X9.31 RSA Signatures 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>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>

\(^2\) Data length, signature length.

For these mechanisms, the \( ulMinKeySize \) and \( ulMaxKeySize \) fields of the **CK_MECHANISM_INFO** structure specify the supported range of RSA modulus sizes, in bits.
12.2 DSA

12.2.1 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_PARAMETER_GEN
- CKM_FORTEZZA_TIMESTAMP

12.2.2 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 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME(^{1,3})</td>
<td>Big integer</td>
<td>Subprime (q) (160 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>

\(^{1}\) Refer to table Table 15 for footnotes

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-2 for more information on DSA keys.

The following is a sample template for creating a DSA public key object:

```c
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)}
};

12.2.3 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 49, DSA Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME1,4,6</td>
<td>Big integer</td>
<td>Prime $p$ (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME1,4,6</td>
<td>Big integer</td>
<td>Subprime $q$ (160 bits)</td>
</tr>
<tr>
<td>CKA_BASE1,4,6</td>
<td>Big integer</td>
<td>Base $g$</td>
</tr>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Big integer</td>
<td>Private value $x$</td>
</tr>
</tbody>
</table>

*Refer to table Table 15 for footnotes

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-2 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)},
    ...}
```

Copyright © 2004 RSA Security Inc.  June 2004
12.2.4 DSA domain parameter objects

DSA domain parameter objects (object class \texttt{CKO\_DOMAIN\_PARAMETERS}, key type \texttt{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:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Attribute & Data type & Meaning \\
\hline
\texttt{CKA\_PRIME}\textsuperscript{1,4} & Big integer & Prime $p$ (512 to 1024 bits, in steps of 64 bits) \\
\texttt{CKA\_SUBPRIME}\textsuperscript{1,4} & Big integer & Subprime $q$ (160 bits) \\
\texttt{CKA\_BASE}\textsuperscript{1,4} & Big integer & Base $g$ \\
\texttt{CKA\_PRIME\_BITS}\textsuperscript{2,3} & CK\_ULONG & Length of the prime value. \\
\hline
\end{tabular}
\end{table}

\textsuperscript{1}Refer to table Table 15 for footnotes

The \texttt{CKA\_PRIME}, \texttt{CKA\_SUBPRIME} and \texttt{CKA\_BASE} attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-2 for more information on DSA domain parameters.

The following is a sample template for creating a DSA domain parameter object:

```c
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\_VALUE, value, sizeof(value)}
};
```
12.2.5 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.

12.2.6 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.
12.2.7 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>
<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</td>
<td>40</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>DSA public key</td>
<td>20, 40(^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 DSA prime sizes, in bits.

12.2.8 DSA with SHA-1

The DSA with SHA-1 mechanism, denoted **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 52, 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>40</td>
</tr>
<tr>
<td>C_Verify</td>
<td>DSA public key</td>
<td>any, 40(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^2\) 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 DSA prime sizes, in bits.

12.2.9 FORTEZZA timestamp

The FORTEZZA timestamp mechanism, denoted \texttt{CKM_FORTEZZA_TIMESTAMP}, is a mechanism for single-part signatures and verification. The signatures it produces and verifies are DSA digital signatures over the provided hash value and the current time.

It has no parameters.

Constraints on key types and the length of data are summarized in the following table. The input and output data may begin at the same location in memory.

Table 53, FORTEZZA Timestamp: 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</td>
<td>40</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>DSA public key</td>
<td>20, 40(^2)</td>
<td>N/A</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 DSA prime sizes, in bits.

12.3 Elliptic Curve

The Elliptic Curve (EC) cryptosystem (also related to ECDSA) in this document is the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

Table 54, Mechanism Information Flags

| CKF_EC_F_P | 0x00100000 | True if the mechanism can be used |
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 three representation methods to define the domain parameters for an EC key. Only the ecParameters and the namedCurve 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_NAMEDCURVE flag identifies a Cryptoki library supporting the namedCurve choice. A Cryptoki library that can perform EC mechanisms must set either or both of these flags 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. 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 variety (or of an unsupported size of a supported variety) is made, that attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT. 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.

12.3.1 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.

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.

12.3.2 Definitions

This section defines the key type “CKK_ECDSA” and “CKK_EC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.
Mechanisms:

Note: CKM_ECDSA_KEY_PAIR_GEN is deprecated in v2.11
CKM_ECDSA_KEY_PAIR_GEN
CKM_EC_KEY_PAIR_GEN
CKM_ECDSA
CKM_ECDSA_SHA1
CKM_ECDH1_DERIVE
CKM_ECDH1_COFACTOR_DERIVE
CKM_ECMQV_DERIVE

12.3.3 ECDSA public key objects

EC (also related to ECDSA) public key objects (object class CKO_PUBLIC_KEY, key type CKK_EC or CKK_ECDSA) 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 55, 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 an ANSI X9.62 Parameters value</td>
</tr>
<tr>
<td>(CKA_ECDSA_PARAMS)</td>
<td></td>
<td></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>

\(^1\) Refer to table Table 15 for footnotes

The CKA_EC_PARAMS or CKA_ECDSA_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:

\[
\text{Parameters ::= CHOICE } \{
  \text{ecParameters ECPARAMETERS,}
  \text{namedCurve CURVES.}\text{id\{CurveNames\},}
  \text{implicitlyCA NULL}
\}
\]

This allows detailed specification of all required values using choice ecParameters, the use of a namedCurve 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. The use of a namedCurve is recommended over the choice ecParameters. The choice implicitlyCA must not be used in Cryptoki.

The following is a sample template for creating an EC (ECDSA) public key object:

\[
\text{CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;}
\]
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)}
};

12.3.4 Elliptic curve private key objects

EC (also related to ECDSA) private key objects (object class CKO_PRIVATE_KEY, key type CKK_EC or CKK_ECDSA) hold EC private keys. See Section 12.3 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 56, 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 an ANSI X9.62 Parameters value</td>
</tr>
<tr>
<td>(CKA_ECDSA_PARAMS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CKA_VALUE^{1,4,6,7}</td>
<td>Big integer</td>
<td>ANSI X9.62 private value (d)</td>
</tr>
</tbody>
</table>

\(^1\) Refer to table Table 15 for footnotes

The CKA_EC_PARAMS or CKA_ECDSA_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:

\[
\text{Parameters ::= CHOICE} \{ \text{ecParameters ECPARAMETERS, namedCurve CURVES.id({CurveNames}), implicitlyCA NULL} \}
\]

This allows detailed specification of all required values using choice ecParameters, the use of a namedCurve 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. The use of a namedCurve 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 EC (ECDSA) 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)}
};
```

12.3.5 Elliptic curve key pair generation

The EC (also related to ECDSA) key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN or CKM_ECDSA_KEY_PAIR_GEN, is a key pair generation mechanism for EC.

This mechanism does not have a parameter.

The mechanism generates EC public/private key pairs with particular EC domain parameters, as specified in the CKA_EC_PARAMS or CKA_ECDSA_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.

The mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_EC_POINT attributes to the new public key and the CKA_CLASS, CKA_KEY_TYPE, CKA_EC_PARAMS or CKA_ECDSA_PARAMS and CKA_CKA_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 \(2^{200}\) 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).

### 12.3.6 ECDSA without hashing

Refer section 12.3.1 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 57, ECDSA: 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>ECDSA private key</td>
<td>any(^3)</td>
<td>2nLen</td>
</tr>
<tr>
<td>C_Verify(^1)</td>
<td>ECDSA public key</td>
<td>any(^3), ≤2nLen(^2)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^1\) Single-part operations only.

\(^2\) Data length, signature length.

\(^3\) 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).

### 12.3.7 ECDSA with SHA-1

Refer section 12.3.1 for signature encoding.
The ECDSA with SHA-1 mechanism, denoted **CKM_ECDSA_SHA1**, 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.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

**Table 58, ECDSA 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>ECDSA private key</td>
<td>any</td>
<td>$2nLen$</td>
</tr>
<tr>
<td>C_Verify</td>
<td>ECDSA public key</td>
<td>any, $\leq 2nLen$</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).

### 12.3.8 EC mechanism parameters

♦ **CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR**

*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:

```c
typedef CK_ULONG CK_EC_KDF_TYPE;
```

The following table lists the defined functions.

**Table 59, EC: Key Derivation Functions**

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
<td>0x00000001</td>
</tr>
<tr>
<td>CKD_SHA1_KDF</td>
<td>0x00000002</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 function
CKD_SHA1_KDF, which is based on SHA-1, derives keying data from the shared secret value as defined in ANSI X9.63.

CK_EC_KDF_TYPE_PTR is a pointer to a CK_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_COFACCTOR_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
- `pPublicData` pointer to other party’s EC public key value

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_ECDH1_DERIVE_PARAMS_PTR is a pointer to a CK_ECDH1_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
- **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
- **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 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_ECMQV_DERIVE_PARAMS_PTR is a pointer to a CK_ECMQV_DERIVE_PARAMS.
12.3.9 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, 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_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 $2^{200}$ 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).

12.3.10 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 $2^{200}$ 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).

### 12.3.11 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 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 $2^{200}$ 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).

### 12.4 Diffie-Hellman

#### 12.4.1 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_DERIVE
- CKM_X9_42_DH_KEY_PAIR_GEN
- CKM_X9_42_DH_DERIVE
- CKM_X9_42_DH_HYBRID_DERIVE
- CKM_X9_42_MQV_DERIVE
- CKM_DH_PKCS_PARAMETER_GEN
- CKM_X9_42_DH_PARAMETER_GEN

### 12.4.2 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 60, 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 Table 15 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)}
```
12.4.3 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 61, 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(^{1,3})</td>
<td>Big integer</td>
<td>Prime (p \geq 1024) bits, in steps of 256 bits</td>
</tr>
<tr>
<td>CKA_BASE(^{1,3})</td>
<td>Big integer</td>
<td>Base (g)</td>
</tr>
<tr>
<td>CKA_SUBPRIME(^{1,3})</td>
<td>Big integer</td>
<td>Subprime (q \geq 160) bits</td>
</tr>
<tr>
<td>CKA_VALUE(^{1,4})</td>
<td>Big integer</td>
<td>Public value (y)</td>
</tr>
</tbody>
</table>

\(^1\text{Refer to table Table 15 for footnotes}\)

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)}
};
```
12.4.4 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><strong>CKA_PRIME</strong></td>
<td>Big integer</td>
<td>Prime ( p )</td>
</tr>
<tr>
<td><strong>CKA_BASE</strong></td>
<td>Big integer</td>
<td>Base ( g )</td>
</tr>
<tr>
<td><strong>CKA_VALUE</strong></td>
<td>Big integer</td>
<td>Private value ( x )</td>
</tr>
<tr>
<td><strong>CKA_VALUE_BITS</strong></td>
<td>CK_ULONG</td>
<td>Length in bits of private value ( x )</td>
</tr>
</tbody>
</table>

Refer to table Table 15 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.

Note that when generating an 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.

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)}},
```
12.4.5 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(^1,4,6)</td>
<td>Big integer</td>
<td>Prime (p) ((\geq 1024) bits, in steps of 256 bits)</td>
</tr>
<tr>
<td>CKA_BASE(^1,4,6)</td>
<td>Big integer</td>
<td>Base (g)</td>
</tr>
<tr>
<td>CKA_SUBPRIME(^1,4,6)</td>
<td>Big integer</td>
<td>Subprime (q) ((\geq 160) bits)</td>
</tr>
<tr>
<td>CKA_VALUE(^1,4,6,7)</td>
<td>Big integer</td>
<td>Private value (x)</td>
</tr>
</tbody>
</table>

\(^1\) Refer to table Table 15 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:

```
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)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```
12.4.6 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&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime $p$</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_PRIME_BITS&lt;sup&gt;2,3&lt;/sup&gt;</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
</tbody>
</table>

* Refer to table Table 15 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 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},
};
```
12.4.7 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(^{1,4})</td>
<td>Big integer</td>
<td>Prime (p \geq 1024) bits, in steps of 256 bits</td>
</tr>
<tr>
<td>CKA_BASE(^{1,4})</td>
<td>Big integer</td>
<td>Base (g)</td>
</tr>
<tr>
<td>CKA_SUBPRIME(^{1,4})</td>
<td>Big integer</td>
<td>Subprime (q \geq 160) bits</td>
</tr>
<tr>
<td>CKA_PRIME_BITS(^{2,3})</td>
<td>CK_ULONG</td>
<td>Length of the prime value.</td>
</tr>
<tr>
<td>CKA_SUBPRIME_BITS(^{2,3})</td>
<td>CK_ULONG</td>
<td>Length of the subprime value.</td>
</tr>
</tbody>
</table>

\(^{1}\) Refer to table Table 15 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)},
};
```
12.4.8 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. 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.

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.

12.4.9 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.
12.4.10 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.
12.4.11 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 66, X9.42 Diffie-Hellman Key Derivation Functions**

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKD_NULL</td>
<td>0x00000001</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_ASN1</td>
<td>0x00000003</td>
</tr>
<tr>
<td>CKD_SHA1_KDF_CONCATENATE</td>
<td>0x00000004</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 CKM_X9_42_DH_DERIVE 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_DH2_DERIVE_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:

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
<table>
<thead>
<tr>
<th><strong>pOtherInfo</strong></th>
<th>some data shared between the two parties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ulPublicDataLen</strong></td>
<td>the length in bytes of the other party’s first X9.42 Diffie-Hellman public key</td>
</tr>
<tr>
<td><strong>pPublicData</strong></td>
<td>pointer to other party’s first X9.42 Diffie-Hellman public key value</td>
</tr>
<tr>
<td><strong>ulPrivateDataLen</strong></td>
<td>the length in bytes of the second X9.42 Diffie-Hellman private key</td>
</tr>
<tr>
<td><strong>hPrivateData</strong></td>
<td>key handle for second X9.42 Diffie-Hellman private key value</td>
</tr>
<tr>
<td><strong>ulPublicDataLen2</strong></td>
<td>the length in bytes of the other party’s second X9.42 Diffie-Hellman public key</td>
</tr>
<tr>
<td><strong>pPublicData2</strong></td>
<td>pointer to other party’s second X9.42 Diffie-Hellman public key value</td>
</tr>
</tbody>
</table>

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 \texttt{CKD\_NULL}, \texttt{pOtherInfo} must be NULL and \texttt{ulOtherInfoLen} must be zero. With the key derivation function \texttt{CKD\_SHA1\_KDF\_ASN1}, \texttt{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 \texttt{CKD\_SHA1\_KDF\_CONCATENATE}, an optional \texttt{pOtherInfo} may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, \texttt{pOtherInfo} must be NULL and \texttt{ulOtherInfoLen} must be zero.

\texttt{CK\_X9\_42\_MQV\_DERIVE\_PARAMS\_PTR} is a pointer to a \texttt{CK\_X9\_42\_MQV\_DERIVE\_PARAMS}.

### 12.4.12 X9.42 Diffie-Hellman key pair generation

The X9.42 Diffie-Hellman key pair generation mechanism, denoted \texttt{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.

The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and subprime, as specified in the \texttt{CKA\_PRIME}, \texttt{CKA\_BASE} and \texttt{CKA\_SUBPRIME} attributes of the template for the public key.

The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new public key and the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, \texttt{CKA\_PRIME}, \texttt{CKA\_BASE}, \texttt{CKA\_SUBPRIME}, and \texttt{CKA\_VALUE} attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman public and private key types must be specified in the templates.

For this mechanism, the \texttt{ulMinKeySize} and \texttt{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{CKA\_PRIME} attribute.

### 12.4.13 X9.42 Diffie-Hellman domain parameter generation

The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted \texttt{CKM\_X9\_42\_DH\_PARAMETER\_GEN}, is a domain parameters generation mechanism based on X9.42 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

It does not have a parameter.
The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime length in bits, as specified in the **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes of the template for the domain parameters.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, **CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the template for the domain parameters, 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 X9.42 Diffie-Hellman prime sizes, in bits.

### 12.4.14 X9.42 Diffie-Hellman key derivation

The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM_X9_42_DH_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a **CK_X9_42_DH1_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. Note that in order to validate this mechanism it may be required to use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM_SHA_1_HMAC_GENERAL**) over some test data.

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 X9.42 Diffie-Hellman prime sizes, in bits, for the CKA_PRIME attribute.

12.4.15 X9.42 Diffie-Hellman hybrid key derivation

The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted CKM_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 pair, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a CK_X9_42_DH2_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. Note that in order to validate this mechanism it may be required to use the CKA_VALUE attribute as the key of a general-length MAC mechanism (e.g. CKM_SHA_1_HMAC_GENERAL) over some test data.

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{CKA_PRIME} attribute.

\section*{12.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation}

The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted \texttt{CKM\_X9\_42\_MQV\_DERIVE}, is a mechanism for key derivation based on 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{CKA\_KEY\_TYPE} attribute of the template and, if it has one and the key type supports it, the \texttt{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 \texttt{CKA\_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{CKA\_VALUE} attribute as the key of a general-length MAC mechanism (e.g. \texttt{CKM\_SHA\_1\_HMAC\_GENERAL}) over some test data.

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 \textit{opposite} value from its \texttt{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{CKA\_PRIME} attribute.
12.5 KEA

12.5.1 Definitions
This section defines the key type “CKK KEA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_KEA_KEY_PAIR_GEN
- CKM_KEA_KEY_DERIVE

12.5.2 KEA mechanism parameters

♦ CK_KEA_DERIVE_PARAMS; CK_KEA_DERIVE_PARAMS_PTR

CK_KEA_DERIVE_PARAMS is a structure that provides the parameters to the CKM_KEA_DERIVE mechanism. It is defined as follows:

```c
typedef struct CK_KEA_DERIVE_PARAMS {
    CK_BBOOL isSender;
    CK_ULONG ulRandomLen;
    CK_BYTE_PTR pRandomA;
    CK_BYTE_PTR pRandomB;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
} CK_KEA_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

- **isSender** Option for generating the key (called a TEK). The value is CK_TRUE if the sender (originator) generates the TEK, CK_FALSE if the recipient is regenerating the TEK.
- **ulRandomLen** size of random Ra and Rb, in bytes
- **pRandomA** pointer to Ra data
- **pRandomB** pointer to Rb data
- **ulPublicDataLen** other party’s KEA public key size
- **pPublicData** pointer to other party’s KEA public key value

CK_KEA_DERIVE_PARAMS_PTR is a pointer to a CK_KEA_DERIVE_PARAMS.
12.5.3 KEA public key objects

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

Table 67, KEA Public Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Prime p (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Subprime q (160 bits)</td>
</tr>
<tr>
<td>CKA_BASE&lt;sup&gt;1,3&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Base g (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_VALUE&lt;sup&gt;1,4&lt;/sup&gt;</td>
<td>Big integer</td>
<td>Public value y</td>
</tr>
</tbody>
</table>

Refer to table Table 15 for footnotes

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “KEA domain parameters”.

The following is a sample template for creating a KEA public key object:

```c
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_KEA;
CK_UTF8CHAR label[] = "A KEA 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)}
};
```

12.5.4 KEA private key objects

KEA private key objects (object class CKO_PRIVATE_KEY, key type CKK KEA) hold KEA private keys. The following table defines the KEA private key object attributes, in addition to the common attributes defined for this object class:
Table 68, KEA Private Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_PRIME¹,⁴,⁶</td>
<td>Big integer</td>
<td>Prime ( p ) (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_SUBPRIME¹,⁴,⁶</td>
<td>Big integer</td>
<td>Subprime ( q ) (160 bits)</td>
</tr>
<tr>
<td>CKA_BASE¹,⁴,⁶</td>
<td>Big integer</td>
<td>Base ( g ) (512 to 1024 bits, in steps of 64 bits)</td>
</tr>
<tr>
<td>CKA_VALUE¹,⁴,⁶,⁷</td>
<td>Big integer</td>
<td>Private value ( x )</td>
</tr>
</tbody>
</table>

¹ Refer to table Table 15 for footnotes

The CKA_PRIME, CKA_SUBPRIME and CKA_BASE attribute values are collectively the “KEA domain parameters”.

Note that when generating a KEA private key, the KEA parameters are *not* specified in the key’s template. This is because KEA private keys are only generated as part of a KEA key *pair*, and the KEA parameters for the pair are specified in the template for the KEA public key.

The following is a sample template for creating a KEA private key object:

```c
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_KEA;
CK_UTF8CHAR label[] = “A KEA 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_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```
12.5.5 KEA key pair generation

The KEA key pair generation mechanism, denoted CKM_KEA_KEY_PAIR_GEN, generates key pairs for the Key Exchange Algorithm, as defined by NIST’s “SKIPJACK and KEA Algorithm Specification Version 2.0”, 29 May 1998.

It does not have a parameter.

The mechanism generates KEA 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. Note that this version of Cryptoki does not include a mechanism for generating these KEA domain parameters.

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 KEA 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 KEA prime sizes, in bits.

12.5.6 KEA key derivation

The KEA key derivation mechanism, denoted CKM_KEA_DERIVE, is a mechanism for key derivation based on KEA, the Key Exchange Algorithm, as defined by NIST’s “SKIPJACK and KEA Algorithm Specification Version 2.0”, 29 May 1998.

It has a parameter, a CK_KEA_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.

As defined in the Specification, KEA can be used in two different operational modes: full mode and e-mail mode. Full mode is a two-phase key derivation sequence that requires real-time parameter exchange between two parties. E-mail mode is a one-phase key derivation sequence that does not require real-time parameter exchange. By convention, e-mail mode is designated by use of a fixed value of one (1) for the KEA parameter R_b (pRandomB).
The operation of this mechanism depends on two of the values in the supplied `CK_KEA_DERIVE_PARAMS` structure, as detailed in the table below. Note that, in all cases, the data buffers pointed to by the parameter structure fields `pRandomA` and `pRandomB` must be allocated by the caller prior to invoking `C_DeriveKey`. Also, the values pointed to by `pRandomA` and `pRandomB` are represented as Cryptoki “Big integer” data (i.e., a sequence of bytes, most-significant byte first).

### Table 69, KEA Parameter Values and Operations

<table>
<thead>
<tr>
<th>Value of boolean</th>
<th>Value of big integer</th>
<th>Token Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isSender</code></td>
<td><code>pRandomB</code></td>
<td>(after checking parameter and template values)</td>
</tr>
<tr>
<td><code>CK_TRUE</code></td>
<td>0</td>
<td>Compute KEA R_a value, store it in <code>pRandomA</code>, return CKR_OK. No derived key object is created.</td>
</tr>
<tr>
<td><code>CK_TRUE</code></td>
<td>1</td>
<td>Compute KEA R_a value, store it in <code>pRandomA</code>, derive key value using e-mail mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td><code>CK_TRUE</code></td>
<td>&gt;1</td>
<td>Compute KEA R_a value, store it in <code>pRandomA</code>, derive key value using full mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td><code>CK_FALSE</code></td>
<td>0</td>
<td>Compute KEA R_b value, store it in <code>pRandomB</code>, return CKR_OK. No derived key object is created.</td>
</tr>
<tr>
<td><code>CK_FALSE</code></td>
<td>1</td>
<td>Derive key value using e-mail mode, create key object, return CKR_OK.</td>
</tr>
<tr>
<td><code>CK_FALSE</code></td>
<td>&gt;1</td>
<td>Derive key value using full mode, create key object, return CKR_OK.</td>
</tr>
</tbody>
</table>

Note that the parameter value `pRandomB == 0` is a flag that the KEA mechanism is being invoked to compute the party’s public random value (R_a or R_b, for sender or recipient, respectively), not to derive a key. In these cases, any object template supplied as the `C_DeriveKey pTemplate` argument should be ignored.

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.

‡ 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`.
• 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 KEA prime sizes, in bits.

12.6 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, EC (also related to ECDSA) private keys and DSA private keys. For wrapping, a private key is BER-encoded according to PKCS #8’s 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:

```plaintext
rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }
dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x942(10046) number-type(2) 1 }

id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
id-dsa OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }
```

where

```plaintext
pkcs-1 OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1
}

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_EXPONENT2, and CKA_COEFFICIENT values, it cannot create an RSAPrivateKey BER-encoding of the key, and so it cannot 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.
- EC (also related with ECDSA) private keys are BER-encoded according to SECG SEC 1 ECPrivateKey ASN.1 type:

  ECPublicKey ::= 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 privateKeyName 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 must be done in CBC mode with PKCS padding.

Unwrapping a wrapped private key undoes the above procedure. The CBC-encrypted ciphertext is decrypted, and the PKCS padding is removed. The data thereby obtained are parsed as a PrivateKeyInfo type, and the wrapped key is produced. An error will result if the original wrapped key does not decrypt properly, or if the decrypted unpadded 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

    DSA OBJECT IDENTIFIER ::= { algorithm 12 }
    algorithm OBJECT IDENTIFIER ::= {
      iso(1) identifier-organization(3) oiw(14) secsig(3)
      algorithm(2) }
with associated parameters

\[
\text{DSAParameters ::= SEQUENCE } \{
\text{prime1 INTEGER, -- modulus p}
\text{prime2 INTEGER, -- modulus q}
\text{base INTEGER -- base g}
\}\]

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.

### 12.7 Generic secret key

#### 12.7.1 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:

\[
\text{CKM_GENERIC_SECRET_KEY_GEN}
\]

#### 12.7.2 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, decryption, signatures or verification; however, other keys can be derived from them. The following table defines the generic secret key object attributes, in addition to the common attributes defined for this object class:

These key types are used in several of the mechanisms described in this section.

<table>
<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 value (arbitrary length)</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>

*Refer to table Table 15 for footnotes*

The following is a sample template for creating a generic secret key object:

\[
\text{CK_OBJECT_CLASS class = CKO_SECRET_KEY;}
\]
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.

12.7.3 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:

\[
\text{CK_OBJECT_CLASS} = \text{CKO_SECRET_KEY};
\]
\[
\text{CK_KEY_TYPE} = \text{CKK_GENERIC_SECRET};
\]

For this mechanism, the \text{ulMinKeySize} and \text{ulMaxKeySize} fields of the CK_MECHANISM_INFO structure specify the supported range of key sizes, in bits.

12.8 HMAC mechanisms

Refer RFC2104 and FIPS 198 for HMAC algorithm description. The HMAC secret key shall correspond to the PKCS11 generic secret key type. 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.
12.9 RC2

RC2 is a block cipher which is trademarked by RSA Security. It has a variable keysize and an additional parameter, the “effective number of bits in the RC2 search space”, which can take on values in the range 1-1024, inclusive. The effective number of bits in the RC2 search space is sometimes specified by an RC2 “version number”; this “version number” is not the same thing as the “effective number of bits”, however. There is a canonical way to convert from one to the other.

12.9.1 Definitions

This section defines the key type “CKK_RC2” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```
CKM_RC2_KEY_GEN
CKM_RC2_ECB
CKM_RC2_CBC
CKM_RC2_MAC
CKM_RC2_MAC_GENERAL
CKM_RC2_CBC_PAD
```

12.9.2 RC2 secret key objects

RC2 secret key objects (object class CKO_SECRET_KEY, key type CKK_RC2) hold RC2 keys. The following table defines the RC2 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[^1,4,6,7]</td>
<td>Byte array</td>
<td>Key value (1 to 128 bytes)</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>

Refer to table Table 15 for footnotes

The following is a sample template for creating an RC2 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC2;
CK_UTF8CHAR label[] = “An RC2 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)}
};

12.9.3 RC2 mechanism parameters

♦ CK_RC2_PARAMS; CK_RC2_PARAMS_PTR

CK_RC2_PARAMS provides the parameters to the CKM_RC2_ECB and CKM_RC2_MAC mechanisms. It holds the effective number of bits in the RC2 search space. It is defined as follows:

typedef CK_ULONG CK_RC2_PARAMS;

CK_RC2_PARAMS_PTR is a pointer to a CK_RC2_PARAMS.

♦ CK_RC2_CBC_PARAMS; CK_RC2_CBC_PARAMS_PTR

CK_RC2_CBC_PARAMS is a structure that provides the parameters to the CKM_RC2_CBC and CKM_RC2_CBC_PAD mechanisms. It is defined as follows:

typedef struct CK_RC2_CBC_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_BYTE iv[8];
} CK_RC2_CBC_PARAMS;

The fields of the structure have the following meanings:

ulEffectiveBits   the effective number of bits in the RC2 search space
iv                the initialization vector (IV) for cipher block chaining mode

CK_RC2_CBC_PARAMS_PTR is a pointer to a CK_RC2_CBC_PARAMS.
CK_RC2_MAC_GENERAL_PARAMS;
CK_RC2_MAC_GENERAL_PARAMS_PTR

CK_RC2_MAC_GENERAL_PARAMS is a structure that provides the parameters to
the CKM_RC2_MAC_GENERAL mechanism. It is defined as follows:

typedef struct CK_RC2_MAC_GENERAL_PARAMS {
    CK_ULONG ulEffectiveBits;
    CK_ULONG ulMacLength;
} CK_RC2_MAC_GENERAL_PARAMS;

The fields of the structure have the following meanings:

ulEffectiveBits       the effective number of bits in the RC2 search space
ulMacLength           length of the MAC produced, in bytes

CK_RC2_MACGENERAL_PARAMS_PTR is a pointer to a
CK_RC2_MAC_GENERAL_PARAMS.

12.9.4 RC2 key generation

The RC2 key generation mechanism, denoted CKM_RC2_KEY_GEN, is a key
generation mechanism for RSA Security’s block cipher RC2.

It does not have a parameter.

The mechanism generates RC2 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 RC2 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 RC2 key sizes, in
bits.

12.9.5 RC2-ECB

RC2-ECB, denoted CKM_RC2_ECB, is a mechanism for single- and multiple-part
encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s
block cipher RC2 and electronic codebook mode as defined in FIPS PUB 81.
It has a parameter, a `CK_RC2_PARAMS`, which indicates the effective number of bits in the RC2 search space.

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 seven null bytes so that the resulting length is a multiple of eight. 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>RC2</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC2</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC2</td>
<td>any</td>
<td>input length rounded up to multiple of 8</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC2</td>
<td>multiple of 8</td>
<td>determined by type of key being unwrapped or <code>CKA_VALUE_LEN</code></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 RC2 effective number of bits.

### 12.9.6 RC2-CBC

RC2-CBC, denoted `CKM_RC2_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC2 and cipher-block chaining mode as defined in FIPS PUB 81.

It has a parameter, a `CK_RC2_CBC_PARAMS` structure, where the first field indicates the effective number of bits in the RC2 search space, and the next field is the initialization vector for cipher block chaining mode.
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 \texttt{CKA\_VALUE} attribute of the key that is wrapped, padded on the trailing end with up to seven null bytes so that the resulting length is a multiple of eight. 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 \texttt{CKA\_KEY\_TYPE} attribute of the template and, if it has one, and the key type supports it, the \texttt{CKA\_VALUE\_LEN} attribute of the template. The mechanism contributes the result as the \texttt{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:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Function & Key type & Input length & Output length & Comments \\
\hline
C\_Encrypt & RC2 & multiple of 8 & same as input length & no final part \\
C\_Decrypt & RC2 & multiple of 8 & same as input length & no final part \\
C\_WrapKey & RC2 & any & input length rounded up to multiple of 8 & \\
C\_UnwrapKey & RC2 & multiple of 8 & determined by type of key being unwrapped or \texttt{CKA\_VALUE\_LEN} & \\
\hline
\end{tabular}
\caption{RC2-CBC: Key And Data Length}
\end{table}

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of RC2 effective number of bits.

### 12.9.7 RC2-CBC with PKCS padding

RC2-CBC with PKCS padding, denoted \texttt{CKM\_RC2\_CBC\_PAD}, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC2; cipher-block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a \texttt{CK\_RC2\_CBC\_PARAMS} structure, where the first field indicates the effective number of bits in the RC2 search space, and the next field is the 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, EC (also related to ECDSA) and DSA private keys (see Section 12.6 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 74, RC2-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>RC2</td>
<td>any</td>
<td>input length rounded up to multiple of 8</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC2</td>
<td>multiple of 8</td>
<td>between 1 and 8 bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC2</td>
<td>any</td>
<td>input length rounded up to multiple of 8</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC2</td>
<td>multiple of 8</td>
<td>between 1 and 8 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 RC2 effective number of bits.

### 12.9.8 General-length RC2-MAC

General-length RC2-MAC, denoted `CKM_RC2_MAC_GENERAL`, is a mechanism for single- and multiple-part signatures and verification, based on RSA Security’s block cipher RC2 and data authentication as defined in FIPS PUB 113.

It has a parameter, a `CK_RC2_MAC_GENERAL_PARAMS` structure, which specifies the effective number of bits in the RC2 search space and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC2 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:
Table 75, General-length RC2-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>RC2</td>
<td>any</td>
<td>0-8, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC2</td>
<td>any</td>
<td>0-8, 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 RC2 effective number of bits.

12.9.9 RC2-MAC

RC2-MAC, denoted by CKM_RC2_MAC, is a special case of the general-length RC2-MAC mechanism (see Section 12.9.8). Instead of taking a CK_RC2_MAC_GENERAL_PARAMS parameter, it takes a CK_RC2_PARAMS parameter, which only contains the effective number of bits in the RC2 search space. RC2-MAC always produces and verifies 4-byte MACs.

Constraints on key types and the length of data are summarized in the following table:

Table 76, RC2-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>RC2</td>
<td>any</td>
<td>4</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC2</td>
<td>any</td>
<td>4</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RC2 effective number of bits.

12.10 RC4

12.10.1 Definitions

This section defines the key type “CKK_RC4” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_RC4_KEY_GEN
CKM_RC4
12.10.2 RC4 secret key objects

RC4 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_RC4**) hold RC4 keys. The following table defines the RC4 secret key object attributes, in addition to the common attributes defined for this object class:

**Table 77, RC4 Secret Key Object**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CKA_VALUE</strong></td>
<td>Byte array</td>
<td>Key value (1 to 256 bytes)</td>
</tr>
<tr>
<td><strong>CKA_VALUE_LEN</strong></td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

`Refer to table Table 15 for footnotes`

The following is a sample template for creating an RC4 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC4;
CK_UTF8CHAR label[] = "An RC4 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)}
};
```

12.10.3 RC4 key generation

The RC4 key generation mechanism, denoted **CKM_RC4_KEY_GEN**, is a key generation mechanism for RSA Security’s proprietary stream cipher RC4.

It does not have a parameter.

The mechanism generates RC4 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 RC4 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 RC4 key sizes, in bits.

### 12.10.4 RC4 mechanism

RC4, denoted `CKM_RC4`, is a mechanism for single- and multiple-part encryption and decryption based on RSA Security’s proprietary stream cipher RC4.

It does not have a parameter.

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>RC4</td>
<td>any</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC4</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 RC4 key sizes, in bits.

### 12.11 RC5

RC5 is a parametrizable block cipher patented by RSA Security. It has a variable wordsize, a variable keysize, and a variable number of rounds. The blocksize of RC5 is always equal to twice its wordsize.

#### 12.11.1 Definitions

This section defines the key type “CKK_RC5” for type `CK_KEY_TYPE` as used in the `CKA_KEY_TYPE` attribute of key objects.

Mechanisms:

- `CKM_RC5_KEY_GEN`
- `CKM_RC5_ECB`
- `CKM_RC5_CBC`
- `CKM_RC5_MAC`
- `CKM_RC5_MAC_GENERAL`
- `CKM_RC5_CBC_PAD`
12.11.2 RC5 secret key objects

RC5 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_RC5`) hold RC5 keys. The following table defines the RC5 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[^1,4,6,7]</td>
<td>Byte array</td>
<td>Key value (0 to 255 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN[^2,3,6]</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

[^1]: Refer to table Table 15 for footnotes

The following is a sample template for creating an RC5 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_RC5;
CK_UTF8CHAR label[] = "An RC5 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)}
};
```

12.11.3 RC5 mechanism parameters

♦ `CK_RC5_PARAMS`, `CK_RC5_PARAMS_PTR`

`CK_RC5_PARAMS` provides the parameters to the `CKM_RC5_ECB` and `CKM_RC5_MAC` mechanisms. It is defined as follows:

```c
typedef struct CK_RC5_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
} CK_RC5_PARAMS;
```

The fields of the structure have the following meanings:

\[ ulWordsize \] wordsize of RC5 cipher in bytes
ulRounds number of rounds of RC5 encipherment

CK_RC5_PARAMS_PTR is a pointer to a CK_RC5_PARAMS.

♦ CK_RC5_CBC_PARAMS; CK_RC5_CBC_PARAMS_PTR

CK_RC5_CBC_PARAMS is a structure that provides the parameters to the CKM_RC5_CBC and CKM_RC5_CBC_PAD mechanisms. It is defined as follows:

```c
typedef struct CK_RC5_CBC_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_BYTE_PTR pIv;
    CK_ULONG ulIvLen;
} CK_RC5_CBC_PARAMS;
```

The fields of the structure have the following meanings:

- **ulWordsize** wordsize of RC5 cipher in bytes
- **ulRounds** number of rounds of RC5 encipherment
- **pIv** pointer to initialization vector (IV) for CBC encryption
- **ulIvLen** length of initialization vector (must be same as blocksize)

CK_RC5_CBC_PARAMS_PTR is a pointer to a CK_RC5_CBC_PARAMS.

♦ CK_RC5_MAC_GENERAL_PARAMS; CK_RC5_MAC_GENERAL_PARAMS_PTR

CK_RC5_MAC_GENERAL_PARAMS is a structure that provides the parameters to the CKM_RC5_MAC_GENERAL mechanism. It is defined as follows:

```c
typedef struct CK_RC5_MAC_GENERAL_PARAMS {
    CK_ULONG ulWordsize;
    CK_ULONG ulRounds;
    CK_ULONG ulMacLength;
} CK_RC5_MAC_GENERAL_PARAMS;
```

The fields of the structure have the following meanings:

- **ulWordsize** wordsize of RC5 cipher in bytes
- **ulRounds** number of rounds of RC5 encipherment
ulMacLength  
length of the MAC produced, in bytes

CK_RC5_MAC_GENERAL_PARAMS_PTR  
is a pointer to a
CK_RC5_MAC_GENERAL_PARAMS.

12.11.4  
RC5 key generation

The RC5 key generation mechanism, denoted CKM_RC5_KEY_GEN, is a key

generation mechanism for RSA Security’s block cipher RC5.

It does not have a parameter.

The mechanism generates RC5 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 RC5 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 RC5 key sizes, in

bytes.

12.11.5  
RC5-ECB

RC5-ECB, denoted CKM_RC5_ECB, is a mechanism for single- and multiple-part

encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s

block cipher RC5 and electronic codebook mode as defined in FIPS PUB 81.

It has a parameter, a CK_RC5_PARAMS, which indicates the wordsize and number of

rounds of encryption to use.

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 null bytes so that the resulting length is a multiple of the

cipher blocksize (twice the wordsize). 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 attributes 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 80, RC5-ECB: 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>RC5</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC5</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC5</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC5</td>
<td>multiple of blocksize</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 RC5 key sizes, in bytes.

### 12.11.6 RC5-CBC

RC5-CBC, denoted `CKM_RC5_CBC`, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC5 and cipher-block chaining mode as defined in FIPS PUB 81.

It has a parameter, a `CK_RC5_CBC_PARAMS` structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

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 seven null bytes so that the resulting length is a multiple of eight. 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 81, RC5-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>RC5</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC5</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC5</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC5</td>
<td>multiple of blocksize</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 RC5 key sizes, in bytes.

12.11.7 RC5-CBC with PKCS padding

RC5-CBC with PKCS padding, denoted CKM_RC5_CBC_PAD, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on RSA Security’s block cipher RC5; cipher-block chaining mode as defined in FIPS PUB 81; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a CK_RC5_CBC_PARAMS structure, which specifies the wordsize and number of rounds of encryption to use, as well as the initialization vector for cipher block chaining mode.

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, EC (also related to ECDSA) and DSA private keys (see Section 12.6 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 82, RC5-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>RC5</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>RC5</td>
<td>multiple of blocksize</td>
<td>between 1 and blocksize bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>RC5</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>RC5</td>
<td>multiple of blocksize</td>
<td>between 1 and blocksize bytes shorter than input length</td>
</tr>
</tbody>
</table>

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RC5 key sizes, in bytes.

12.11.8 General-length RC5-MAC

General-length RC5-MAC, denoted \texttt{CKM_RC5_MAC_GENERAL}, is a mechanism for single- and multiple-part signatures and verification, based on RSA Security’s block cipher RC5 and data authentication as defined in FIPS PUB 113.

It has a parameter, a \texttt{CK_RC5_MAC_GENERAL_PARAMS} structure, which specifies the wordsize and number of rounds of encryption to use and the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final RC5 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 83, General-length RC2-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>RC5</td>
<td>any</td>
<td>0-blocksize, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC5</td>
<td>any</td>
<td>0-blocksize, as specified in parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of RC5 key sizes, in bytes.
12.11.9 RC5-MAC

RC5-MAC, denoted by CKM_RC5_MAC, is a special case of the general-length RC5-MAC mechanism. Instead of taking a CK_RC5_MAC_GENERAL_PARAMS parameter, it takes a CK_RC5_PARAMS parameter. RC5-MAC always produces and verifies MACs half as large as the RC5 blocksize.

Constraints on key types and the length of data are summarized in the following table:

Table 84, RC5-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>RC5</td>
<td>any</td>
<td>RC5 wordsize = ⌊blocksize/2⌋</td>
</tr>
<tr>
<td>C_Verify</td>
<td>RC5</td>
<td>any</td>
<td>RC5 wordsize = ⌊blocksize/2⌋</td>
</tr>
</tbody>
</table>

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure specify the supported range of RC5 key sizes, in bytes.

12.12 AES

For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

12.12.1 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

12.12.2 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 85, AES Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE¹,²,³,⁶,⁷</td>
<td>Byte array</td>
<td>Key value (16, 24, or 32 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN²,³,⁶</td>
<td>CK ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to table Table 15 for footnotes

The following is a sample template for creating an AES secret key object:

```c
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.

12.12.3 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.

### 12.12.4 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>
<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 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 <code>CKA_VALUE_LEN</code></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.
12.12.5 AES-CBC

AES-CBC, denoted \texttt{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 \texttt{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 \texttt{CKA_KEY_TYPE} attribute of the template and, if it has one, and the key type supports it, the \texttt{CKA_VALUE_LEN} attribute of the template. The mechanism contributes the result as the \texttt{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:

\begin{table}[h]
\centering
\caption{AES-CBC: Key And Data Length}
\begin{tabular}{|l|l|l|l|l|}
\hline
\textbf{Function} & \textbf{Key type} & \textbf{Input length} & \textbf{Output length} & \textbf{Comments} \\
\hline
C\_Encrypt & AES & multiple of block size & same as input length & no final part \\
\hline
C\_Decrypt & AES & multiple of block size & same as input length & no final part \\
\hline
C\_WrapKey & AES & any & input length rounded up to multiple of the block size & \\
\hline
C\_UnwrapKey & AES & multiple of block size & determined by type of key being unwrapped or CKA\_VALUE\_LEN & \\
\hline
\end{tabular}
\end{table}

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK\_MECHANISM\_INFO} structure specify the supported range of AES key sizes, in bytes.
12.12.6 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, EC (also related to ECDSA) and DSA private keys (see Section 12.6 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>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.

12.12.7 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 89, General-length AES-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>AES</td>
<td>any</td>
<td>0-block size, as specified in parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>AES</td>
<td>any</td>
<td>0-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.

### 12.12.8 AES-MAC

AES-MAC, denoted by `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 90, AES-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>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.
12.13 General block cipher

For brevity’s sake, the mechanisms for the DES, CAST, CAST3, CAST128 (CAST5), IDEA, and CDMF block ciphers will be described together here. Each of these ciphers has the following mechanisms, which will be described in a templatized form.

12.13.1 Definitions

This section defines the key types “CKK_DES”, “CKK_CAST”, “CKK_CAST3”, “CKK_CAST5” (deprecated in v2.11), “CKK_CAST128”, “CKK_IDEA” and “CKK_CDMF” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_DES_KEY_GEN
CKM_DES_ECB
CKM_DES_CBC
CKM_DES_MAC
CKM_DES_MAC_GENERAL
CKM_DES_CBC_PAD
CKM_CDMF_KEY_GEN
CKM_CDMF_ECB
CKM_CDMF_CBC
CKM_CDMF_MAC
CKM_CDMF_MAC_GENERAL
CKM_CDMF_CBC_PAD
CKM_DES_OFB64
CKM_DES_OFB8
CKM_DES_CFB64
CKM_DES_CFB8
CKM_CAST_KEY_GEN
CKM_CAST_ECB
CKM_CAST_CBC
CKM_CAST_MAC
CKM_CAST_MAC_GENERAL
CKM_CAST_CBC_PAD
CKM_CAST3_KEY_GEN
CKM_CAST3_ECB
CKM_CAST3_CBC
CKM_CAST3_MAC
CKM_CAST3_MAC_GENERAL
CKM_CAST3_CBC_PAD
CKM_CAST5_KEY_GEN
CKM_CAST128_KEY_GEN
CKM_CAST5_ECB
CKM_CAST128_ECB
CKM_CAST5_CBC
12. MECHANISMS

12.13.2 DES secret key objects

DES secret key objects (object class CKO_SECRET_KEY, key type CKK_DES) hold single-length DES keys. The following table defines the DES 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^1,4,6,7</td>
<td>Byte array</td>
<td>Key value (always 8 bytes long)</td>
</tr>
</tbody>
</table>

Refer to table Table 15 for footnotes

DES keys must always have their parity bits properly set as described in FIPS PUB 46-3. Attempting to create or unwrap a DES key with incorrect parity will return an error.

The following is a sample template for creating a DES secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_UTF8CHAR label[] = "A DES secret key object";
CK_BYTE value[8] = {...};
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.

12.13.3 CAST secret key objects

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

Table 92, CAST 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 (1 to 8 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 Table 15 for footnotes*

The following is a sample template for creating a CAST secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST;
CK_UTF8CHAR label[] = “A CAST 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)}
};
```
12.13.4 CAST3 secret key objects

CAST3 secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_CAST3`) hold CAST3 keys. The following table defines the CAST3 secret key object attributes, in addition to the common attributes defined for this object class:

### Table 93, CAST3 Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (1 to 8 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN2,3,6</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

- Refer to table Table 15 for footnotes

The following is a sample template for creating a CAST3 secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST3;
CK_UTF8CHAR label[] = "A CAST3 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)}
};
```

12.13.5 CAST128 (CAST5) secret key objects

CAST128 (also known as CAST5) secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_CAST128` or `CKK_CAST5`) hold CAST128 keys. The following table defines the CAST128 secret key object attributes, in addition to the common attributes defined for this object class:

### Table 94, CAST128 (CAST5) Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (1 to 16 bytes)</td>
</tr>
<tr>
<td>CKA_VALUE_LEN2,3,6</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

- Refer to table Table 15 for footnotes

The following is a sample template for creating a CAST128 (CAST5) secret key object:
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAST128;
CK_UTF8CHAR label[] = "A CAST128 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)}
};

12.13.6 IDEA secret key objects

IDEA secret key objects (object class CKO_SECRET_KEY, key type CKK_IDEA) hold IDEA keys. The following table defines the IDEA 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 16 bytes long)</td>
</tr>
</tbody>
</table>

Refer to table Table 15 for footnotes

The following is a sample template for creating an IDEA secret key object:

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_IDEA;
CK_UTF8CHAR label[] = "An IDEA 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)}
};

12.13.7 CDMF secret key objects

CDMF secret key objects (object class CKO_SECRET_KEY, key type CKK_CDMF) hold single-length CDMF keys. The following table defines the CDMF secret key object attributes, in addition to the common attributes defined for this object class:
Table 96, CDMF 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 value (always 8 bytes long)</td>
</tr>
</tbody>
</table>

^Refer to table Table 15 for footnotes

CDMF keys must always have their parity bits properly set in exactly the same fashion described for DES keys in FIPS PUB 46-3. Attempting to create or unwrap a CDMF key with incorrect parity will return an error.

The following is a sample template for creating a CDMF secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CDMF;
CK_UTF8CHAR label[] = "A CDMF secret key object";
CK_BYTE value[8] = {...};
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)}
};
```

12.13.8 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), CAST, CAST3, CAST128 (CAST5), IDEA, CDMF and AES ciphers. It also provides the parameters to the general-length HMACing mechanisms (i.e. MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPEMD-128 and RIPEMD-160) and the two SSL 3.0 MACing mechanisms (i.e. MD5 and SHA-1). It holds the length of the MAC that these mechanisms will produce. It is defined as follows:

```c
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

CK_MAC_GENERAL_PARAMS_PTR is a pointer to a CK_MAC_GENERAL_PARAMS.
12.13.9 General block cipher key generation

Cipher <NAME> has a key generation mechanism, “<NAME> key generation”, denoted **CKM_<NAME>_KEY_GEN**.

This mechanism 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 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.

When DES keys or CDMF keys are generated, their parity bits are set properly, as specified in FIPS PUB 46-3. Similarly, when a triple-DES key is generated, each of the DES keys comprising it has its parity bits set properly.

When DES or CDMF keys are generated, it is token-dependent whether or not it is possible for “weak” or “semi-weak” keys to be generated. Similarly, when triple-DES keys are generated, it is token dependent whether or not it is possible for any of the component DES keys to be “weak” or “semi-weak” keys.

When CAST, CAST3, or CAST128 (CAST5) keys are generated, the template for the secret key must specify a **CKA_VALUE_LEN** attribute.

For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for the key generation mechanisms for these ciphers, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

12.13.10 General block cipher ECB

Cipher <NAME> has an electronic codebook mechanism, “<NAME>-ECB”, denoted **CKM_<NAME>_ECB**. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>.

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 null bytes so that the resulting length is a multiple of <NAME>’s blocksize. 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 97, General Block Cipher ECB: 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>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C.WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C.UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>determined by type of key being unwrapped or <code>CKA_VALUE_LEN</code></td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

### 12.13.11 General block cipher CBC

Cipher `<NAME>` has a cipher-block chaining mode, “<NAME>-CBC”, denoted `CKM_<NAME>_CBC`. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with `<NAME>`.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as `<NAME>`’s blocksize.

Constraints on key types and the length of data are summarized in the following table:
Table 98, General Block Cipher 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>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>input length rounded up to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>multiple of blocksize</td>
<td></td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>determined by type of key</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>being unwrapped or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CKA_VALUE_LEN</td>
<td></td>
</tr>
</tbody>
</table>

For this mechanism, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for these ciphers, the \texttt{ulMinKeySize} and \texttt{ulMaxKeySize} fields of the \texttt{CK_MECHANISM_INFO} structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

12.13.12 General block cipher CBC with PKCS padding

Cipher <NAME> has a cipher-block chaining mode with PKCS padding, “<NAME>-CBC with PKCS padding”, denoted \texttt{CKM_<NAME>\_CBC\_PAD}. It is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping with <NAME>. All ciphertext is padded with PKCS padding.

It has a parameter, an initialization vector for cipher block chaining mode. The initialization vector has the same length as <NAME>’s blocksize.

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 \texttt{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, EC (also related to ECDSA) and DSA private keys (see Section 12.6 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 99, General Block Cipher 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>&lt;NAME&gt;</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>between 1 and blocksize bytes shorter than input length</td>
</tr>
<tr>
<td>C_WrapKey</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>input length rounded up to multiple of blocksize</td>
</tr>
<tr>
<td>C_UnwrapKey</td>
<td>&lt;NAME&gt;</td>
<td>multiple of blocksize</td>
<td>between 1 and blocksize bytes shorter than input length</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

#### 12.13.13 General-length general block cipher MAC

Cipher `<NAME>` has a general-length MACing mode, “General-length `<NAME>`-MAC”, denoted `CKM_<NAME>_MAC_GENERAL`. It is a mechanism for single- and multiple-part signatures and verification, based on the `<NAME>` encryption algorithm and data authentication as defined in FIPS PUB 113.

It has a parameter, a `CK_MAC_GENERAL_PARAMS`, which specifies the size of the output.

The output bytes from this mechanism are taken from the start of the final cipher block produced in the MACing process.

Constraints on key types and the length of input and output data are summarized in the following table:
Table 100, General-length General Block Cipher 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>&lt;NAME&gt;</td>
<td>any</td>
<td>0-blocksize, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>0-blocksize, depending on parameters</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

12.13.14 General block cipher MAC

Cipher `<NAME>` has a MACing mechanism, “<NAME>-MAC”, denoted `CKM_<NAME>_MAC`. This mechanism is a special case of the `CKM_<NAME>_MAC_GENERAL` mechanism described above. It always produces an output of size half as large as `<NAME>`’s blocksize.

This mechanism has no parameters.

Constraints on key types and the length of data are summarized in the following table:

Table 101, General Block Cipher 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>&lt;NAME&gt;</td>
<td>any</td>
<td>⌊blocksize/2⌋</td>
</tr>
<tr>
<td>C_Verify</td>
<td>&lt;NAME&gt;</td>
<td>any</td>
<td>⌊blocksize/2⌋</td>
</tr>
</tbody>
</table>

For this mechanism, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure may or may not be used. The CAST, CAST3, and CAST128 (CAST5) ciphers have variable key sizes, and so for these ciphers, the `ulMinKeySize` and `ulMaxKeySize` fields of the `CK_MECHANISM_INFO` structure specify the supported range of key sizes, in bytes. For the DES, DES3 (triple-DES), IDEA, and CDMF ciphers, these fields are not used.

12.14 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.
12.14.1 Definitions

Mechanisms:

- CKM_DES_ECB_ENCRYPT_DATA
- CKM_DES_CBC_ENCRYPT_DATA
- CKM_DES3_ECB_ENCRYPT_DATA
- CKM_DES3_CBC_ENCRYPT_DATA
- CKM_AES_ECB_ENCRYPT_DATA
- CKM_AES_CBC_ENCRYPT_DATA

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;

12.14.2 Mechanism Parameters

Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 12.34.2

Table 102, Mechanism Parameters

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</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></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></td>
</tr>
<tr>
<td>CKM_DES3_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_AES_CBC_ENCRYPT_DATA</td>
<td>Uses CK_AES_CBC_ENCRYPT_DATA_PARAMS.</td>
</tr>
</tbody>
</table>
12.14.3 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 12.21.5.

If the data is too short to make the requested key then the mechanism returns CKR_DATA_LENGTH_INVALID.

12.15 Double and Triple-length DES

12.15.1 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


12.15.2 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 103, DES2 Secret Key Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (always 16 bytes long)</td>
</tr>
</tbody>
</table>

Refer to Table 15 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.
### 12.15.3 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 104, DES3 Secret Key Object Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE4,6,7</td>
<td>Byte array</td>
<td>Key value (always 24 bytes long)</td>
</tr>
</tbody>
</table>

Refer to Table 15 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:

```plaintext
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.

### 12.15.4 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` (these mechanisms are described in templatized form in Section 12.13. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be “weak” or “semi-weak” keys.

### 12.15.5 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:

\[
\text{DES3-E}( \{K1,K2,K3\}, P ) = E( K3, D( K2, E( K1, P ) ) )
\]

\[
\text{DES3-D}( \{K1,K2,K3\}, C ) = D( K1, E( K2, D( K3, P ) ) )
\]

### 12.15.6 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).
12.15.7 DES and Triple length DES in OFB Mode

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 blocksize.

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_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.

12.15.8 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 blocksize.

Constraints on key types and the length of data are summarized in the following table:
Table 106, 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.

12.16 SKIPJACK

12.16.1 Definitions

This section defines the key type “CKK_SKIPJACK” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SKIPJACK_KEY_GEN
- CKM_SKIPJACK_ECB64
- CKM_SKIPJACK_CBC64
- CKM_SKIPJACK_OFB64
- CKM_SKIPJACK_CFB64
- CKM_SKIPJACK_CFB32
- CKM_SKIPJACK_CFB16
- CKM_SKIPJACK_CFB8
- CKM_SKIPJACK_WRAP
- CKM_SKIPJACK_PRIVATE_WRAP
- CKM_SKIPJACK_RELAYX
12.16.2 SKIPJACK secret key objects

SKIPJACK secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_SKIPJACK**) hold a single-length MEK or a TEK. The following table defines the SKIPJACK 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 12 bytes long)</td>
</tr>
</tbody>
</table>

Refer to table Table 15 for footnotes

SKIPJACK keys have 16 checksum bits, and these bits must be properly set. Attempting to create or unwrap a SKIPJACK key with incorrect checksum bits will return an error.

It is not clear that any tokens exist (or will ever exist) which permit an application to create a SKIPJACK key with a specified value. Nonetheless, we provide templates for doing so.

The following is a sample template for creating a SKIPJACK MEK secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SKIPJACK;
CK_UTF8CHAR label[] = "A SKIPJACK MEK secret key object";
CK_BYTE value[12] = {...};
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)}
};
```

The following is a sample template for creating a SKIPJACK TEK secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SKIPJACK;
CK_UTF8CHAR label[] = "A SKIPJACK TEK secret key object";
CK_BYTE value[12] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
```
12.16.3 SKIPJACK Mechanism parameters

♦ CK_SKIPJACK_PRIVATE_WRAP_PARAMS;
CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR

CK_SKIPJACK_PRIVATE_WRAP_PARAMS is a structure that provides the parameters to the CKM_SKIPJACK_PRIVATE_WRAP mechanism. It is defined as follows:

typedef struct CK_SKIPJACK_PRIVATE_WRAP_PARAMS {
    CK_ULONG ulPasswordLen;
    CK_BYTE_PTR pPassword;
    CK_ULONG ulPublicDataLen;
    CK_BYTE_PTR pPublicData;
    CK_ULONG ulPandGLen;
    CK_ULONG ulQLen;
    CK_ULONG ulRandomLen;
    CK_BYTE_PTR pRandomA;
    CK_BYTE_PTR pPrimeP;
    CK_BYTE_PTR pBaseG;
    CK_BYTE_PTR pSubprimeQ;
} CK_SKIPJACK_PRIVATE_WRAP_PARAMS;

The fields of the structure have the following meanings:

ulPasswordLen length of the password
pPassword pointer to the buffer which contains the user-supplied password
ulPublicDataLen other party’s key exchange public key size
pPublicData pointer to other party’s key exchange public key value
ulPandGLen length of prime and base values
ulQLen length of subprime value
ulRandomLen size of random Ra, in bytes
pRandomA pointer to Ra data
pPrimeP  pointer to Prime, p, value
pBaseG  pointer to Base, g, value
pSubprimeQ  pointer to Subprime, q, value

CK_SKIPJACK_PRIVATE_WRAP_PARAMS_PTR is a pointer to a CK_PRIVATE_WRAP_PARAMS.

♦ CK_SKIPJACK_RELAYX_PARAMS;
CK_SKIPJACK_RELAYX_PARAMS_PTR

CK_SKIPJACK_RELAYX_PARAMS is a structure that provides the parameters to the
CKM_SKIPJACK_RELAYX mechanism. It is defined as follows:

typedef struct CK_SKIPJACK_RELAYX_PARAMS {
    CK_ULONG ulOldWrappedXLen;
    CK_BYTE_PTR pOldWrappedX;
    CK_ULONG ulOldPasswordLen;
    CK_BYTE_PTR pOldPassword;
    CK_ULONG ulOldPublicDataLen;
    CK_BYTE_PTR pOldPublicData;
    CK_ULONG ulOldRandomLen;
    CK_BYTE_PTR pOldRandomA;
    CK_ULONG ulNewPasswordLen;
    CK_BYTE_PTR pNewPassword;
    CK_ULONG ulNewPublicDataLen;
    CK_BYTE_PTR pNewPublicData;
    CK_ULONG ulNewRandomLen;
    CK_BYTE_PTR pNewRandomA;
} CK_SKIPJACK_RELAYX_PARAMS;

The fields of the structure have the following meanings:

ulOldWrappedXLen  length of old wrapped key in bytes
pOldWrappedX  pointer to old wrapper key
ulOldPasswordLen  length of the old password
pOldPassword  pointer to the buffer which contains the old user-supplied password
ulOldPublicDataLen  old key exchange public key size
pOldPublicData  pointer to old key exchange public key value
ulOldRandomLen  size of old random Ra in bytes
12. MECHANISMS

\begin{itemize}
  \item \textit{pOldRandomA} \hspace{1cm} pointer to old Ra data
  \item \textit{ulNewPasswordLen} \hspace{1cm} length of the new password
  \item \textit{pNewPassword} \hspace{1cm} pointer to the buffer which contains the new user-supplied password
  \item \textit{ulNewPublicDataLen} \hspace{1cm} new key exchange public key size
  \item \textit{pNewPublicData} \hspace{1cm} pointer to new key exchange public key value
  \item \textit{ulNewRandomLen} \hspace{1cm} size of new random Ra in bytes
  \item \textit{pNewRandomA} \hspace{1cm} pointer to new Ra data
\end{itemize}

\textbf{CK_SKIPJACK_RELAYX_PARAMS_PTR} is a pointer to a \textbf{CK_SKIPJACK_RELAYX_PARAMS}.

12.16.4 SKIPJACK key generation

The SKIPJACK key generation mechanism, denoted \textit{CKM_SKIPJACK_KEY_GEN}, is a key generation mechanism for SKIPJACK. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

The mechanism contributes the \textbf{CKA_CLASS}, \textbf{CKA_KEY_TYPE}, and \textbf{CKA_VALUE} attributes to the new key.

12.16.5 SKIPJACK-ECB64

SKIPJACK-ECB64, denoted \textit{CKM_SKIPJACK_ECB64}, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit electronic codebook mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 108, SKIPJACK-ECB64: Data and 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>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.6 SKIPJACK-CBC64

SKIPJACK-CBC64, denoted **CKM_SKIPJACK_CBC64**, is a mechanism for single-and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher-block chaining mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:

Table 109, SKIPJACK-CBC64: Data and 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>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.7 SKIPJACK-OFB64

SKIPJACK-OFB64, denoted **CKM_SKIPJACK_OFB64**, is a mechanism for single-and multiple-part encryption and decryption with SKIPJACK in 64-bit output feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 110, SKIPJACK-OFB64: Data and 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>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.8 SKIPJACK-CFB64

SKIPJACK-CFB64, denoted CKM_SKIPJACK_CFB64, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 64-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:

Table 111, SKIPJACK-CFB64: Data and 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>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 8</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.9 SKIPJACK-CFB32

SKIPJACK-CFB32, denoted CKM_SKIPJACK_CFB32, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 32-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 112, SKIPJACK-CFB32: Data and 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>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.10 SKIPJACK-CFB16

SKIPJACK-CFB16, denoted CKM_SKIPJACK_CFB16, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 16-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:

Table 113, SKIPJACK-CFB16: Data and 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>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.11 SKIPJACK-CFB8

SKIPJACK-CFB8, denoted CKM_SKIPJACK_CFB8, is a mechanism for single- and multiple-part encryption and decryption with SKIPJACK in 8-bit cipher feedback mode as defined in FIPS PUB 185.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 114, SKIPJACK-CFB8: Data and 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>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>SKIPJACK</td>
<td>multiple of 4</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.16.12 SKIPJACK-WRAP

The SKIPJACK-WRAP mechanism, denoted CKM_SKIPJACK_WRAP, is used to wrap and unwrap a secret key (MEK). It can wrap or unwrap SKIPJACK, BATON, and JUNIPER keys.

It does not have a parameter.

12.16.13 SKIPJACK-PRIVATE-WRAP

The SKIPJACK-PRIVATE-WRAP mechanism, denoted CKM_SKIPJACK_PRIVATE_WRAP, is used to wrap and unwrap a private key. It can wrap KEA and DSA private keys.

It has a parameter, a CK_SKIPJACK_PRIVATE_WRAP_PARAMS structure.

12.16.14 SKIPJACK-RELAYX

The SKIPJACK-RELAYX mechanism, denoted CKM_SKIPJACK_RELAYX, is used with the C_WrapKey function to “change the wrapping” on a private key which was wrapped with the SKIPJACK-PRIVATE-WRAP mechanism (see Section 12.16.13).

It has a parameter, a CK_SKIPJACK_RELAYX_PARAMS structure.

Although the SKIPJACK-RELAYX mechanism is used with C_WrapKey, it differs from other key-wrapping mechanisms. Other key-wrapping mechanisms take a key handle as one of the arguments to C_WrapKey; however, for the SKIPJACK-RELAYX mechanism, the [always invalid] value 0 should be passed as the key handle for C_WrapKey, and the already-wrapped key should be passed in as part of the CK_SKIPJACK_RELAYX_PARAMS structure.

12.17 BATON

12.17.1 Definitions

This section defines the key type “CKK_BATON” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.
Mechanisms:

\[
\begin{align*}
CKM\_BATON\_KEY\_GEN \\
CKM\_BATON\_ECB128 \\
CKM\_BATON\_ECB96 \\
CKM\_BATON\_CBC128 \\
CKM\_BATON\_COUNTER \\
CKM\_BATON\_SHUFFLE \\
CKM\_BATON\_WRAP
\end{align*}
\]

12.17.2 BATON secret key objects

BATON secret key objects (object class `CKO_SECRET_KEY`, key type `CKK_BATON`) hold single-length BATON keys. The following table defines the BATON secret key object attributes, in addition to the common attributes defined for this object class:

**Table 115, BATON 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 value (always 40 bytes long)</td>
</tr>
</tbody>
</table>

\[^{1,4,6,7}\] Refer to table Table 15 for footnotes

BATON keys have 160 checksum bits, and these bits must be properly set. Attempting to create or unwrap a BATON key with incorrect checksum bits will return an error.

It is not clear that any tokens exist (or will ever exist) which permit an application to create a BATON key with a specified value. Nonetheless, we provide templates for doing so.

The following is a sample template for creating a BATON MEK secret key object:

\[
\begin{align*}
CK\_OBJECT\_CLASS \ class = CKO\_SECRET\_KEY; \\
CK\_KEY\_TYPE \ keyType = CKK\_BATON; \\
CK\_UTF8CHAR \ label[] = "A BATON MEK secret key object"; \\
CK\_BYTE \ value[40] = {...}; \\
CK\_BBOOL \ true = CK\_TRUE; \\
CK\_ATTRIBUTE \ template[] = {
    \{CKA\_CLASS, &class, sizeof(class)\},
    \{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)\},
    \{CKA\_TOKEN, &true, sizeof(true)\},
    \{CKA\_LABEL, label, sizeof(label)\},
    \{CKA\_ENCRYPT, &true, sizeof(true)\},
    \{CKA\_VALUE, value, sizeof(value)\}
};
\end{align*}
\]

The following is a sample template for creating a BATON TEK secret key object:
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BATON;
CK_UTF8CHAR label[] = “A BATON TEK secret key object”;
CK_BYTE value[40] = {...};
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_WRAP, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

12.17.3 BATON key generation

The BATON key generation mechanism, denoted CKM_BATON_KEY_GEN, is a key generation mechanism for BATON. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.

This mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key.

12.17.4 BATON-ECB128

BATON-ECB128, denoted CKM_BATON_ECB128, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 116, BATON-ECB128: Data and 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>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.17.5 BATON-ECB96

BATON-ECB96, denoted **CKM_BATON_ECB96**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 96-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:

Table 117, BATON-ECB96: Data and 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>BATON</td>
<td>multiple of 12</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>multiple of 12</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.17.6 BATON-CBC128

BATON-CBC128, denoted **CKM_BATON_CBC128**, is a mechanism for single- and multiple-part encryption and decryption with BATON in 128-bit cipher-block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
12.17.7 BATON-COUNTER

BATON-COUNTER, denoted CKM_BATON_COUNTER, is a mechanism for single- and multiple-part encryption and decryption with BATON in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

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>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.17.8 BATON-SHUFFLE

BATON-SHUFFLE, denoted CKM_BATON_SHUFFLE, is a mechanism for single- and multiple-part encryption and decryption with BATON in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table:
Table 120, BATON-SHUFFLE: Data and 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>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>BATON</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.17.9 BATON WRAP

The BATON wrap and unwrap mechanism, denoted CKM_BATON_WRAP, is a function used to wrap and unwrap a secret key (MEK). It can wrap and unwrap SKIPJACK, BATON, and JUNIPER keys.

It has no parameters.

When used to unwrap a key, this mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to it.

12.18 JUNIPER

12.18.1 Definitions

This section defines the key type “CKK_JUNIPER” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_JUNIPER_KEY_GEN
- CKM_JUNIPER_ECB128
- CKM_JUNIPER_CBC128
- CKM_JUNIPER_COUNTER
- CKM_JUNIPER_SHUFFLE
- CKM_JUNIPER_WRAP

12.18.2 JUNIPER secret key objects

JUNIPER secret key objects (object class CKO_SECRET_KEY, key type CKK_JUNIPER) hold single-length JUNIPER keys. The following table defines the JUNIPER secret key object attributes, in addition to the common attributes defined for this object class:

Table 121, JUNIPER Secret Key Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKA_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value (always 40 bytes long)</td>
</tr>
</tbody>
</table>

\(^\text{1} \text{Refer to table Table 15 for footnotes}\)
JUNIPER keys have 160 checksum bits, and these bits must be properly set. Attempting to create or unwrap a JUNIPER key with incorrect checksum bits will return an error.

It is not clear that any tokens exist (or will ever exist) which permit an application to create a JUNIPER key with a specified value. Nonetheless, we provide templates for doing so.

The following is a sample template for creating a JUNIPER MEK secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_UTF8CHAR label[] = "A JUNIPER MEK secret key object";
CK_BYTE value[40] = {...};
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)}
};
```

The following is a sample template for creating a JUNIPER TEK secret key object:

```c
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_JUNIPER;
CK_UTF8CHAR label[] = "A JUNIPER TEK secret key object";
CK_BYTE value[40] = {...};
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_WRAP, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

12.18.3 JUNIPER key generation

The JUNIPER key generation mechanism, denoted `CKM_JUNIPER_KEY_GEN`, is a key generation mechanism for JUNIPER. The output of this mechanism is called a Message Encryption Key (MEK).

It does not have a parameter.
The mechanism contributes the \texttt{CKA\_CLASS}, \texttt{CKA\_KEY\_TYPE}, and \texttt{CKA\_VALUE} attributes to the new key.

\section*{12.18.4 JUNIPER-ECB128}

JUNIPER-ECB128, denoted \texttt{CKM\_JUNIPER\_ECB128}, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit electronic codebook mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) may begin at the same location in memory.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Function & Key type & Input length & Output length & Comments \\
\hline
C\_Encrypt & JUNIPER & multiple of 16 & same as input length & no final part \\
C\_Decrypt & JUNIPER & multiple of 16 & same as input length & no final part \\
\hline
\end{tabular}
\caption{JUNIPER-ECB128: Data and Length}
\end{table}

\section*{12.18.5 JUNIPER-CBC128}

JUNIPER-CBC128, denoted \texttt{CKM\_JUNIPER\_CBC128}, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in 128-bit cipher-block chaining mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) may begin at the same location in memory.
Table 123, JUNIPER-CBC128: Data and 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>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.18.6 JUNIPER-COUNTER

JUNIPER COUNTER, denoted CKM_JUNIPER_COUNTER, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in counter mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) may begin at the same location in memory.

Table 124, JUNIPER-COUNTER: Data and 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>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>

12.18.7 JUNIPER-SHUFFLE

JUNIPER-SHUFFLE, denoted CKM_JUNIPER_SHUFFLE, is a mechanism for single- and multiple-part encryption and decryption with JUNIPER in shuffle mode.

It has a parameter, a 24-byte initialization vector. During an encryption operation, this IV is set to some value generated by the token—in other words, the application cannot specify a particular IV when encrypting. It can, of course, specify a particular IV when decrypting.

Constraints on key types and the length of data are summarized in the following table. For encryption and decryption, the input and output data (parts) may begin at the same location in memory.

Table 125, JUNIPER-SHUFFLE: Data and 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>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>JUNIPER</td>
<td>multiple of 16</td>
<td>same as input length</td>
<td>no final part</td>
</tr>
</tbody>
</table>
12.18.8 JUNIPER WRAP

The JUNIPER wrap and unwrap mechanism, denoted CKM_JUNIPER_WRAP, is a function used to wrap and unwrap an MEK. It can wrap or unwrap SKIPJACK, BATON, and JUNIPER keys.

It has no parameters.

When used to unwrap a key, this mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to it.

12.19 MD2

12.19.1 Definitions

Mechanisms:

CKM_MD2
CKM_MD2_HMAC
CKM_MD2_HMAC_GENERAL
CKM_MD2_KEY_DERIVATION

12.19.2 MD2 digest

The MD2 mechanism, denoted CKM_MD2, is a mechanism for message digesting, following the MD2 message-digest algorithm defined in RFC 1319.

It does not have a parameter.

Constraints on the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>16</td>
</tr>
</tbody>
</table>

12.19.3 General-length MD2-HMAC

The general-length MD2-HMAC mechanism, denoted CKM_MD2_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD2 hash function. The keys it uses are generic secret keys.

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 0-16 (the output size of MD2 is...
16 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 16-byte HMAC output.

Table 127, General-length MD2-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>0-16, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>0-16, depending on parameters</td>
</tr>
</tbody>
</table>

12.19.4 MD2-HMAC

The MD2-HMAC mechanism, denoted CKM_MD2_HMAC, is a special case of the general-length MD2-HMAC mechanism in Section 12.19.3.

It has no parameter, and always produces an output of length 16.

12.19.5 MD2 key derivation

MD2 key derivation, denoted CKM_MD2_KEY_DERIVATION, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD2.

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 16 bytes (the output size of MD2).
- 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 16 bytes, such as DES3, an error is generated.
This mechanism has the following rules about key sensitivity and extractability:

- The **CKASENSITIVE** and **CKAEXTRACTABLE** 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 **CKAALWAYSSENSITIVE** attribute set to CK_FALSE, then the derived key will as well. If the base key has its **CKAALWAYSSENSITIVE** attribute set to CK_TRUE, then the derived key has its **CKAALWAYSSENSITIVE** attribute set to the same value as its **CKASENSITIVE** attribute.

- Similarly, if the base key has its **CKANEVEREXTRACTABLE** attribute set to CK_FALSE, then the derived key will, too. If the base key has its **CKANEVEREXTRACTABLE** attribute set to CK_TRUE, then the derived key has its **CKANEVEREXTRACTABLE** attribute set to the *opposite* value from its **CKAEXTRACTABLE** attribute.

### 12.20 MD5

#### 12.20.1 Definitions

Mechanisms:

- CKM_MD5
- CKM_MD5_HMAC
- CKM_MD5_HMAC_GENERAL
- CKM_MD5_KEY_DERIVATION

#### 12.20.2 MD5 digest

The MD5 mechanism, denoted **CKM_MD5**, is a mechanism for message digesting, following the MD5 message-digest algorithm defined in RFC 1321.

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.
### 12.20.3 General-length MD5-HMAC

The general-length MD5-HMAC mechanism, denoted **CKM_MD5_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the MD5 hash function. The keys it uses are generic secret keys.

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 0-16 (the output size of MD5 is 16 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 16-byte HMAC output.

#### Table 129, General-length MD5-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>0-16, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>0-16, depending on parameters</td>
</tr>
</tbody>
</table>

### 12.20.4 MD5-HMAC

The MD5-HMAC mechanism, denoted **CKM_MD5_HMAC**, is a special case of the general-length MD5-HMAC mechanism in Section 12.20.3.

It has no parameter, and always produces an output of length 16.

### 12.20.5 MD5 key derivation

MD5 key derivation, denoted **CKM_MD5_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with MD5.

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 16 bytes (the output size of MD5).
- 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 16 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.

12.21 SHA-1

12.21.1 Definitions

Mechanisms:

    CKM_SHA_1
    CKM_SHA_1_HMAC
    CKM_SHA_1_HMAC_GENERAL
    CKM_SHA1_KEY_DERIVATION
12.21.2 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 130, 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>

12.21.3 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.

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 0-20 (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 131, 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>0-20, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>0-20, depending on parameters</td>
</tr>
</tbody>
</table>

12.21.4 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 12.21.3.

It has no parameter, and always produces an output of length 20.
12.21.5 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.
12.22 SHA-256

12.22.1 Definitions

Mechanisms:

- CKM_SHA256
- CKM_SHA256_HMAC
- CKM_SHA256_HMAC_GENERAL
- CKM_SHA256_KEY_DERIVATION

12.22.2 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 132, 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>

12.22.3 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 12.21.3, except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 0-32. The keys it uses are generic secret keys. 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 0-32 (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 133, General-length SHA-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</td>
<td>Any</td>
<td>0-32, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>Any</td>
<td>0-32, depending on parameters</td>
</tr>
</tbody>
</table>

12.22.4 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 12.22.3.

It has no parameter, and always produces an output of length 32.

12.22.5 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 12.21.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

12.23 SHA-384

12.23.1 Definitions

Mechanisms:

- CKM_SHA384
- CKM_SHA384_HMAC
- CKM_SHA384_HMAC_GENERAL
- CKM_SHA384_KEY_DERIVATION

12.23.2 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.
### 12.23.3 General-length SHA-384-HMAC

The general-length SHA-384-HMAC mechanism, denoted `CKM_SHA256_HMAC_GENERAL`, is the same as the general-length SHA-1-HMAC mechanism in Section 12.21.3, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 0-48.

### 12.23.4 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.

### 12.23.5 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 12.21.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

### 12.24 SHA-512

#### 12.24.1 Definitions

Mechanisms:

- `CKM_SHA512`
- `CKM_SHA512_HMAC`
- `CKM_SHA512_HMAC_GENERAL`
- `CKM_SHA512_KEY_DERIVATION`

#### 12.24.2 SHA-512 digest

The SHA-512 mechanism, denoted `CKM_SHA512`, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.
Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

**Table 135, 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>

### 12.24.3 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 12.21.3, except that it uses the HMAC construction based on the SHA-512 hash function and length of the output should be in the range 0-64.

### 12.24.4 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.

### 12.24.5 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 12.21.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

### 12.25 FASTHASH

#### 12.25.1 Definitions

Mechanisms:

- CKM_FASTHASH

#### 12.25.2 FASTHASH digest

The FASTHASH mechanism, denoted CKM_FASTHASH, is a mechanism for message digesting, following the U. S. government’s algorithm.

It does not have a parameter.
Constraints on the length of input and output data are summarized in the following table:

Table 136, FASTHASH: 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>40</td>
</tr>
</tbody>
</table>

12.26 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.

12.26.1 Definitions

Mechanisms:

```
CKM_PBE_MD2_DES_CBC
CKM_PBE_MD5_DES_CBC
CKM_PBE_MD5_CAST_CBC
CKM_PBE_MD5_CAST3_CBC
CKM_PBE_MD5_CAST5_CBC
CKM_PBE_MD5_CAST128_CBC
CKM_PBE_SHA1_CAST5_CBC
CKM_PBE_SHA1_CAST128_CBC
CKM_PBE_SHA1_RC4_128
CKM_PBE_SHA1_RC4_40
CKM_PBE_SHA1_DES3_EDE_CBC
CKM_PBE_SHA1_DES2_EDE_CBC
CKM_PBE_SHA1_RC2_128_CBC
CKM_PBE_SHA1_RC2_40_CBC
CKM_PKCS5_PBKD2
CKM_PBA_SHA1_WITH_SHA1_HMAC
```
12.26.2 Password-based encryption/authentication mechanism parameters

♦ CK_PBE_PARAMS; CK_PBE_PARAMS_PTR

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:

```c
typedef struct CK_PBE_PARAMS {
    CK_BYTE_PTR pInitVector;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG ulPasswordLen;
    CK_BYTE_PTR pSalt;
    CK_ULONG ulSaltLen;
    CK_ULONG ulIteration;
} 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.

12.26.3 MD2-PBE for DES-CBC

MD2-PBE for DES-CBC, denoted CKM_PBE_MD2_DES_CBC, is a mechanism used for generating a DES secret key and an IV from a password and a salt value by using the MD2 digest algorithm and an iteration count. This functionality is defined in PKCS#5 as PBKDF1.

It has a parameter, a CK_PBE_PARAMS structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.
12.26.4 MD5-PBE for DES-CBC

MD5-PBE for DES-CBC, denoted **CKM_PBE_MD5_DES_CBC**, is a mechanism used for generating a DES secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is defined in PKCS#5 as PBKDF1.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

12.26.5 MD5-PBE for CAST-CBC

MD5-PBE for CAST-CBC, denoted **CKM_PBE_MD5_CAST_CBC**, is a mechanism used for generating a CAST secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS#5 PBKDF1 for MD5 and DES.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The length of the CAST key generated by this mechanism may be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

12.26.6 MD5-PBE for CAST3-CBC

MD5-PBE for CAST3-CBC, denoted **CKM_PBE_MD5_CAST3_CBC**, is a mechanism used for generating a CAST3 secret key and an IV from a password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS#5 PBKDF1 for MD5 and DES.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The length of the CAST3 key generated by this mechanism may be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

12.26.7 MD5-PBE for CAST128-CBC (CAST5-CBC)

MD5-PBE for CAST128-CBC (CAST5-CBC), denoted **CKM_PBE_MD5_CAST128_CBC** or **CKM_PBE_MD5_CAST5_CBC**, is a mechanism used for generating a CAST128 (CAST5) secret key and an IV from a
password and a salt value by using the MD5 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS#5 PBKDF1 for MD5 and DES.

It has a parameter, a `CK_PBE_PARAMS` structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The length of the CAST128 (CAST5) key generated by this mechanism may be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

### 12.26.8 SHA-1-PBE for CAST128-CBC (CAST5-CBC)

SHA-1-PBE for CAST128-CBC (CAST5-CBC), denoted `CKM_PBE_SHA1_CAST128_CBC` or `CKM_PBE_SHA1_CAST5_CBC`, is a mechanism used for generating a CAST128 (CAST5) secret key and an IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. This functionality is analogous to that defined in PKCS#5 PBKDF1 for MD5 and DES.

It has a parameter, a `CK_PBE_PARAMS` structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The length of the CAST128 (CAST5) key generated by this mechanism may be specified in the supplied template; if it is not present in the template, it defaults to 8 bytes.

### 12.26.9 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 v2.0. The following table lists the defined functions.
Table 137, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions

<table>
<thead>
<tr>
<th>Source Identifier</th>
<th>Value</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKP_PKCS5_PBKD2_HMAC_SHA1</td>
<td>0x00000001</td>
<td>No Parameter. pPrfData must be NULL and ulPrfDataLen must be zero.</td>
</tr>
</tbody>
</table>

CK_PKS5_PKBD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR is a pointer to a CK_PKS5_PKBD2_PSEUDO_RANDOM_FUNCTION_TYPE.

♦ CK_PKS5_PKBD2_SALT_SOURCE_TYPE;
CK_PKS5_PKBD2_SALT_SOURCE_TYPE_PTR

CK_PKS5_PKBD2_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:

```c
typedef CK_ULONG CK_PKS5_PKBD2_SALT_SOURCE_TYPE;
```

The following salt value sources are defined in PKCS #5 v2.0. The following table lists the defined sources along with the corresponding data type for the pSaltSourceData field in the CK_PKS5_PKBD2_PARAM structure defined below.

Table 138, PKCS #5 PBKDF2 Key Generation: Salt sources

<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_PKS5_PKBD2_SALT_SOURCE_TYPE_PTR is a pointer to a CK_PKS5_PKBD2_SALT_SOURCE_TYPE.

♦ CK_PKCS5_PKBD2_PARAMS; CK_PKCS5_PKBD2_PARAMS_PTR

CK_PKS5_PKBD2_PARAMS is a structure that provides the parameters to the CKM_PKCS5_PKBD2 mechanism. The structure is defined as follows:

```c
typedef struct CK_PKS5_PKBD2_PARAMS {
    CK_PKS5_PKBD2_SALT_SOURCE_TYPE_PTR saltSource;
    CK_VOID_PTR pSaltSourceData;
    CK_ULONG ulSaltSourceDataLen;
    CK_ULONG iterations;
    CK_PKS5_PKBD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR prf;
    CK_VOID_PTR pPrfData;
    CK_ULONG ulPrfDataLen;
    CK_UTF8CHAR_PTR pPassword;
    CK_ULONG_PTR ulPasswordLen;
} CK_PKS5_PKBD2_PARAMS;
```
The fields of the structure have the following meanings:

- \textit{saltSource} : source of the salt value
- \textit{pSaltSourceData} : data used as the input for the salt source
- \textit{ulSaltSourceDataLen} : length of the salt source input
- \textit{iterations} : number of iterations to perform when generating each block of random data
- \textit{prf} : pseudo-random function to used to generate the key
- \textit{pPrfData} : data used as the input for PRF in addition to the salt value
- \textit{ulPrfDataLen} : length of the input data for the PRF
- \textit{pPassword} : points to the password to be used in the PBE key generation
- \textit{ulPasswordLen} : length in bytes of the password information

\texttt{CK\_PKCS5\_PBKD2\_PARAMS\_PTR} is a pointer to a \texttt{CK\_PKCS5\_PBKD2\_PARAMS}.

### 12.26.10 PKCS #5 PBKD2 key generation

PKCS #5 PBKDF2 key generation, denoted \texttt{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 \texttt{CK\_PKCS5\_PBKD2\_PARAMS} 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 \texttt{CKA\_KEY\_TYPE} and \texttt{CKA\_VALUE\_LEN} attributes. If the key type has a fixed length the \texttt{CKA\_VALUE\_LEN} attribute may be omitted.

### 12.27 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 \( f: \mathbb{Z}_2^u \times \mathbb{Z}_2^v \rightarrow \mathbb{Z}_2^u \) (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 \( j=\lceil n/u \rceil \).

6. For \( i=1, 2, \ldots, j \), do the following:
   a) Set \( A_i=H^c(D||I) \), the \( c^{th} \) hash of \( D||I \). That is, compute the hash of \( D||I \); 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 \( I_j=(I_j+B+1) \mod 2^v \) 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.

### 12.27.1 SHA-1-PBE for 128-bit RC4

SHA-1-PBE for 128-bit RC4, denoted $\text{CKM\_PBE\_SHA1\_RC4\_128}$, is a mechanism used for generating a 128-bit RC4 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 $\text{CK\_PBE\_PARAMS}$ 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 RC4 does not require an IV.

The key produced by this mechanism will typically be used for performing password-based encryption.

### 12.27.2 SHA-1-PBE for 40-bit RC4

SHA-1-PBE for 40-bit RC4, denoted $\text{CKM\_PBE\_SHA1\_RC4\_40}$, is a mechanism used for generating a 40-bit RC4 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 $\text{CK\_PBE\_PARAMS}$ 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 RC4 does not require an IV.

The key produced by this mechanism will typically be used for performing password-based encryption.

### 12.27.3 SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted $\text{CKM\_PBE\_SHA1\_DES3\_EDE\_CBC}$, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is
described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### 12.27.4 SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### 12.27.5 SHA-1-PBE for 128-bit RC2-CBC

SHA-1-PBE for 128-bit RC2-CBC, denoted **CKM_PBE_SHA1_RC2_128_CBC**, is a mechanism used for generating a 128-bit RC2 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.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 128. This ensures compatibility with the ASN.1 Object Identifier **pbeWithSHA1And128BitRC2_CBC**.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.
12.27.6 SHA-1-PBE for 40-bit RC2-CBC

SHA-1-PBE for 40-bit RC2-CBC, denoted **CKM_PBE_SHA1_RC2_40_CBC**, is a mechanism used for generating a 40-bit RC2 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.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

When the key and IV generated by this mechanism are used to encrypt or decrypt, the effective number of bits in the RC2 search space should be set to 40. This ensures compatibility with the ASN.1 Object Identifier **pbeWithSHA1And40BitRC2-CBC**.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

12.27.7 SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.
12.28 RIPE-MD

12.28.1 Definitions

Mechanisms:

- CKM_RIPEMD128
- CKM_RIPEMD128_HMAC
- CKM_RIPEMD128_HMAC_GENERAL
- CKM_RIPEMD160
- CKM_RIPEMD160_HMAC
- CKM_RIPEMD160_HMAC_GENERAL

12.28.2 RIPE-MD 128 digest

The RIPE-MD 128 mechanism, denoted CKM_RIPEMD128, is a mechanism for message digesting, following the RIPE-MD 128 message-digest algorithm.

It does not have a parameter.

Constraints on the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>16</td>
</tr>
</tbody>
</table>

12.28.3 General-length RIPE-MD 128-HMAC

The general-length RIPE-MD 128-HMAC mechanism, denoted CKM_RIPEMD128_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the RIPE-MD 128 hash function. The keys it uses are generic secret keys.

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 0-16 (the output size of RIPE-MD 128 is 16 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 16-byte HMAC output.

Table 140, General-length RIPE-MD 128-HMAC:

<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>0-16, depending on parameters</td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any</td>
<td>0-16, depending on</td>
</tr>
</tbody>
</table>
12.28.4 RIPE-MD 128-HMAC

The RIPE-MD 128-HMAC mechanism, denoted CKM_RIPEMD128_HMAC, is a special case of the general-length RIPE-MD 128-HMAC mechanism in Section 12.28.3. It has no parameter, and always produces an output of length 16.

12.28.5 RIPE-MD 160

The RIPE-MD 160 mechanism, denoted CKM_RIPEMD160, is a mechanism for message digesting, following the RIPE-MD 160 message-digest algorithm defined in ISO-10118. It does not have a parameter.

Constraints on the length of data are summarized in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Data length</th>
<th>Digest length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Digest</td>
<td>any</td>
<td>20</td>
</tr>
</tbody>
</table>

12.28.6 General-length RIPE-MD 160-HMAC

The general-length RIPE-MD 160-HMAC mechanism, denoted CKM_RIPEMD160_HMAC_GENERAL, is a mechanism for signatures and verification. It uses the HMAC construction, based on the RIPE-MD 160 hash function. The keys it uses are generic secret keys.

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 0-20 (the output size of RIPE-MD 160 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.
Table 142, General-length RIPE-MD 160-HMAC:

<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 0-20,</td>
<td>0-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>depending on parameters</td>
<td></td>
</tr>
<tr>
<td>C_Verify</td>
<td>generic secret</td>
<td>any 0-20,</td>
<td>0-20, depending on parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>depending on parameters</td>
<td></td>
</tr>
</tbody>
</table>

12.28.7 RIPE-MD 160-HMAC

The RIPE-MD 160-HMAC mechanism, denoted `CKM_RIPEMD160_HMAC`, is a special case of the general-length RIPE-MD 160-HMAC mechanism in Section 12.28.6.

It has no parameter, and always produces an output of length 20.

12.29 SET

12.29.1 Definitions

Mechanisms:

`CKM_KEY_WRAP_SET_OAEP`

12.29.2 SET mechanism parameters

♦ `CK_KEY_WRAP_SET_OAEP_PARAMS;`
`CK_KEY_WRAP_SET_OAEP_PARAMS_PTR`

`CK_KEY_WRAP_SET_OAEP_PARAMS` is a structure that provides the parameters to the `CKM_KEY_WRAP_SET_OAEP` mechanism. It is defined as follows:

```c
typedef struct CK_KEY_WRAP_SET_OAEP_PARAMS {
    CK_BYTE bBC;
    CK_BYTE_PTR pX;
    CK_ULONG ulXLen;
} CK_KEY_WRAP_SET_OAEP_PARAMS;
```

The fields of the structure have the following meanings:

- `bBC` block contents byte
- `pX` concatenation of hash of plaintext data (if present) and extra data (if present)
- `ulXLen` length in bytes of concatenation of hash of plaintext data (if present) and extra data (if present). 0 if neither is present
12.29.3 OAEP key wrapping for SET

The OAEP key wrapping for SET mechanism, denoted CKM_KEY_WRAP_SET_OAEP, is a mechanism for wrapping and unwrapping a DES key with an RSA key. The hash of some plaintext data and/or some extra data may optionally be wrapped together with the DES key. This mechanism is defined in the SET protocol specifications.

It takes a parameter, a CK_KEY_WRAP_SET_OAEP_PARAMS structure. This structure holds the “Block Contents” byte of the data and the concatenation of the hash of plaintext data (if present) and the extra data to be wrapped (if present). If neither the hash nor the extra data is present, this is indicated by the ulXLen field having the value 0.

When this mechanism is used to unwrap a key, the concatenation of the hash of plaintext data (if present) and the extra data (if present) is returned following the convention described in Section 11.2 on producing output. Note that if the inputs to C_UnwrapKey are such that the extra data is not returned (e.g., the buffer supplied in the CK_KEY_WRAP_SET_OAEP_PARAMS structure is NULL_PTR), then the unwrapped key object will not be created, either.

Be aware that when this mechanism is used to unwrap a key, the bBC and pX fields of the parameter supplied to the mechanism may be modified.

If an application uses C_UnwrapKey with CKM_KEY_WRAP_SET_OAEP, it may be preferable for it simply to allocate a 128-byte buffer for the concatenation of the hash of plaintext data and the extra data (this concatenation is never larger than 128 bytes), rather than calling C_UnwrapKey twice. Each call of C_UnwrapKey with CKM_KEY_WRAP_SET_OAEP requires an RSA decryption operation to be performed, and this computational overhead can be avoided by this means.

12.30 LYNKS

12.30.1 Definitions

Mechanisms:

CKM_KEY_WRAP_LYNKS

12.30.2 LYNKS key wrapping

The LYNKS key wrapping mechanism, denoted CKM_KEY_WRAP_LYNKS, is a mechanism for wrapping and unwrapping secret keys with DES keys. It can wrap any 8-
byte secret key, and it produces a 10-byte wrapped key, containing a cryptographic checksum.

It does not have a parameter.

To wrap a 8-byte secret key \( K \) with a DES key \( W \), this mechanism performs the following steps:

1. Initialize two 16-bit integers, \( \text{sum}_1 \) and \( \text{sum}_2 \), to 0.
2. Loop through the bytes of \( K \) from first to last.
   3. Set \( \text{sum}_1 = \text{sum}_1 + \text{the key byte} \) (treat the key byte as a number in the range 0-255).
   4. Set \( \text{sum}_2 = \text{sum}_2 + \text{sum}_1 \).
3. Encrypt \( K \) with \( W \) in ECB mode, obtaining an encrypted key, \( E \).
4. Concatenate the last 6 bytes of \( E \) with \( \text{sum}_2 \), representing \( \text{sum}_2 \) most-significant bit first. The result is an 8-byte block, \( T \).
5. Encrypt \( T \) with \( W \) in ECB mode, obtaining an encrypted checksum, \( C \).
6. Concatenate \( E \) with the last 2 bytes of \( C \) to obtain the wrapped key.

When unwrapping a key with this mechanism, if the cryptographic checksum does not check out properly, an error is returned. In addition, if a DES key or CDMF key is unwrapped with this mechanism, the parity bits on the wrapped key must be set appropriately. If they are not set properly, an error is returned.

12.31 SSL

12.31.1 Definitions

Mechanisms:

- CKM_SSL3_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
12.31.2 SSL mechanism parameters

♦ CK_SSL3_RANDOM_DATA

CK_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;
    CK_ULONG ulClientRandomLen;
    CK_BYTE_PTR pServerRandom;
    CK_ULONG 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:

```c
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 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:

```c
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 is a structure that provides the parameters to the CKM_SSL3_KEY_AND_MAC_DERIVE mechanism. It is defined as follows:

```c
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.

`CK_SSL3_KEY_MAT_PARAMS_PTR` is a pointer to a `CK_SSL3_KEY_MAT_PARAMS`.

### 12.31.3 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.
12.31.4 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 12.31.

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 usable 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.

12.31.5 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 12.31. 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.

### 12.31.6 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 12.31.

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.SSL.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.

If a call to `C_DeriveKey` with this mechanism fails, then none of the four keys will be created on the token.

### 12.31.7 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>
<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.
12.31.8 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 144, 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.

12.32 TLS

Details can be found in [TLS].

12.32.1 Definitions

Mechanisms:

CKM_TLS_PRE_MASTER_KEY_GEN
CKM_TLS_MASTER_KEY_DERIVE
CKM_TLS_KEY_AND_MAC_DERIVE
CKM_TLS_MASTER_KEY_DERIVE_DH
CKM_TLS_PRF
12.32.2 TLS mechanism parameters

♦ 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
- `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_TLS_PRF_PARAMS_PTR is a pointer to a CK_TLS_PRF_PARAMS.

12.32.3 TLS PRF (pseudorandom function)

PRF (pseudo random function) in TLS, denoted CKM_TLS_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_TLS_PRF_PARAMS structure, which allows for 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_TLS_PRF` mechanism returns the requested number of output bytes in the `CK_TLS_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.

---

**12.32.4 Pre_master key generation**

Pre-master key generation in TLS 1.0, denoted `CKM_TLS_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 TLS version 1.0 for RSA-like cipher suites.

It has one parameter, a `CK_VERSION` structure, which provides the client’s TLS 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.

---

**12.32.5 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 12.31.

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 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 usable 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.

### 12.32.6 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in TLS 1.0, denoted 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 `CK_SSL3_MASTER_KEY_DERIVE_PARAMS` structure, which allows for the passing of random data to the token. This structure is defined in Section 12.31. 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.

### 12.32.7 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 12.31.

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_SSL_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.
If a call to \texttt{C\_DeriveKey} with this mechanism fails, then \textit{none} of the four keys will be created on the token.

\subsection*{12.33 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.

\subsection*{12.33.1 Definitions}

Mechanisms:

\begin{verbatim}
CKM_WTLS_PRE_MASTER_KEY_GEN
CKM_WTLS_MASTER_KEY_DERIVE
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
CKM_WTLS_PRF
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE
\end{verbatim}

\subsection*{12.33.2 WTLS mechanism parameters}

\begin{itemize}
  \item \texttt{CK\_WTLS\_RANDOM\_DATA}; \texttt{CK\_WTLS\_RANDOM\_DATA\_PTR}
\end{itemize}

\texttt{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 \texttt{CKM\_WTLS\_MASTER\_KEY\_DERIVE} mechanism. It is defined as follows:

\begin{verbatim}
typedef struct CK\_WTLS\_RANDOM\_DATA {
  CK\_BYTE\_PTR pClientRandom;
  CK\_ULONG ulClientRandomLen;
  CK\_BYTE\_PTR pServerRandom;
  CK\_ULONG ulServerRandomLen;
} CK\_WTLS\_RANDOM\_DATA;
\end{verbatim}

The fields of the structure have the following meanings:

\begin{itemize}
  \item \texttt{pClientRandom} \hspace{1em} pointer to the client's random data
  \item \texttt{ulClientRandomLen} \hspace{1em} length in bytes of the client's random data
  \item \texttt{pServerRandom} \hspace{1em} pointer to the server's random data
\end{itemize}
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; CK_WTLS_PRF_PARAMS_PTR

CK_WTLS_PRF_PARAMS is a structure, which provides the parameters to the CKM_WTLS_PRF mechanism. It is defined as follows:

```c
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:

- **DigestMechanism**: 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; CK_WTLS_KEY_MAT_OUT_PTR**

**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_SEVER_KEY_AND_MAC_DERIVE** or with the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```c
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; CK_WTLS_KEY_MAT_PARAMS_PTR

CK_WTLS_KEY_MAT_PARAMS is a structure that provides the parameters to the CKM_WTLS_SEVER_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:

- **DigestMechanism** the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
- **ulMacSizeInBits** 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.

### 12.33.3 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.

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 indicates the length of the pre master secret key.

For this mechanism, the ulMinKeySize field of the CK_MECHANISM_INFO structure shall indicate 20 bytes.

### 12.33.4 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 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 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.

**12.33.5 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.

### 12.33.6 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.

12.33.7 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.

12.33.8 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 CKASENSITIVE, 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 pphKey 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.

12.34 Miscellaneous simple key derivation mechanisms

12.34.1 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
12.34.2 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; CK_EXTRACT_PARAMS_PTR**

**CK_KEY_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**.

12.34.3 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 `CKASENSITIVE` attribute set to `CK_TRUE`, so does the derived key. If not, then the derived key’s `CKASENSITIVE` attribute is set either from the supplied template or from a default value.

- Similarly, if either of the two original keys has its `CKAEXTRACTABLE` attribute set to `CK_FALSE`, so does the derived key. If not, then the derived key’s `CKAEXTRACTABLE` attribute is set either from the supplied template or from a default value.

- The derived key’s `CKAALWAYSSENSITIVE` attribute is set to `CK_TRUE` if and only if both of the original keys have their `CKAALWAYSSENSITIVE` attributes set to `CK_TRUE`.

- Similarly, the derived key’s `CKANEVEREXTRACTABLE` attribute is set to `CK_TRUE` if and only if both of the original keys have their `CKANEVEREXTRACTABLE` attributes set to `CK_TRUE`. 
12.34.4 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`.

12.34.5 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 \texttt{CKASENSITIVE} attribute set to \texttt{CK_TRUE}, so does the derived key. If not, then the derived key’s \texttt{CKASENSITIVE} attribute is set either from the supplied template or from a default value.

• Similarly, if the base key has its \texttt{CKAEXTRACTABLE} attribute set to \texttt{CK_FALSE}, so does the derived key. If not, then the derived key’s \texttt{CKAEXTRACTABLE} attribute is set either from the supplied template or from a default value.

• The derived key’s \texttt{CKAALWAYSSENSITIVE} attribute is set to \texttt{CK_TRUE} if and only if the base key has its \texttt{CKAALWAYSSENSITIVE} attribute set to \texttt{CK_TRUE}.

• Similarly, the derived key’s \texttt{CKANEVEREXTRACTABLE} attribute is set to \texttt{CK_TRUE} if and only if the base key has its \texttt{CKANEVEREXTRACTABLE} attribute set to \texttt{CK_TRUE}.

12.34.6 XORing of a key and data

XORing key derivation, denoted \texttt{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 \texttt{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 \texttt{0x01234567}, and the value of the data is \texttt{0x89ABCDEF}, then the value of the derived key will be taken from a buffer containing the string \texttt{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.

12.34.7 Extraction of one key from another key

Extraction of one key from another key, denoted 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 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 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 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 b0, b1, …, b31.

2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain the binary string 1001 0101 0010 0110.

3. The value of the new key is thus 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 **CKASENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key’s **CKASENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKAEXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key’s **CKAEXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key’s **CKAALWAYSENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKAALWAYSENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key’s **CKANEVEREXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKANEVEREXTRACTABLE** attribute set to CK_TRUE.
12.35 CMS

12.35.1 Definitions

Mechanisms:

CKM_CMS_SIG

12.35.2 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 145, 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.
12.35.3 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:

   ```c
   typedef struct CK_CMS_SIG_PARAMS {
       CK_OBJECT_HANDLE certificateHandle;
       CK_MECHANISM_PTR pSigningMechanism;
       CK_MECHANISM_PTR pDigestMechanism;
       CK_UTF8CHAR_PTR pContentType;
       CK_BYTE_PTR pRequestedAttributes;
       CK_ULONG ulRequestedAttributesLen;
       CK_BYTE_PTR pRequiredAttributes;
       CK_ULONG ulRequiredAttributesLen;
   } 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.

\( p_{\text{RequestedAttributes}} \) 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.

\( ul_{\text{RequestedAttributesLen}} \) Length in bytes of the value pointed to by \( p_{\text{RequestedAttributes}} \).

\( p_{\text{RequiredAttributes}} \) 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 \( p_{\text{RequestedAttributes}} \) and \( p_{\text{RequiredAttributes}} \) field are set to NULL_PTR.

\( ul_{\text{RequiredAttributesLen}} \) Length in bytes, of the value pointed to by \( p_{\text{RequiredAttributes}} \).

### 12.35.4 CMS signatures

The CMS mechanism, denoted \texttt{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 \( p_{\text{RequestedAttributes}} \). The caller can also require some attributes to be present through the parameters \( p_{\text{RequiredAttributes}} \). The signature is computed in accordance with the parameter \( p_{\text{SigningMechanism}} \).

When this mechanism is used in successful calls to \texttt{C\_Sign} or \texttt{C\_SignFinal}, the \( p_{\text{Signature}} \) return value will point to a DER-encoded value of type \texttt{SignerInfo}. \texttt{SignerInfo} is defined in ASN.1 as follows (for a complete definition of all fields and types, see RFC 2630):

\[
\text{SignerInfo} ::= \text{SEQUENCE} \{
\text{version CMSVersion,}
\text{sid SignerIdentifier,}
\text{digestAlgorithm DigestAlgorithmIdentifier,}
\text{signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,}
\text{signatureAlgorithm SignatureAlgorithmIdentifier,}
\text{signature SignatureValue,}
\text{unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL} \}
\]
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).

In order for an application to find out what attributes are supported by a token, what attributes that will be added by default, and what attributes that always will be added, it shall analyze the contents of the CKH_CMS_ATTRIBUTES hardware feature object.

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 CKM_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 that were possible to present.

12.36 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. Ref. http://www.counterpane.com/bfsverlag.html

12.36.1 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

12.36.2 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(^{1,4,6,7})</td>
<td>Byte array</td>
<td>Key value the key can be any length up to 448 bits. Bit length restricted to an 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>

*Refer to table Table 15 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)}
};
```
12.36.3 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.

12.36.4 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 16-byte initialization vector.

12.37 Twofish

- 128-bit block
- 128-, 192-, or 256-bit key
- 16 rounds
- Works in all standard modes
- Efficient key setup on large microprocessors
- Efficient on smart cards
- Efficient in hardware
- Extensively cryptanalyzed
- Unpatented
- Uncopyrighted
- Free

12.37.1 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

12.37.2 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_VALUE1,4,6,7</td>
<td>Byte array</td>
<td>Key value 128-, 192-, or 256-bit key</td>
</tr>
<tr>
<td>CKA_VALUE_LEN2,3</td>
<td>CK_ULONG</td>
<td>Length in bytes of key value</td>
</tr>
</tbody>
</table>

Refer to table Table 15 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)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_LABEL, label, sizeof(label)-1},
  {CKA_VALUE, value, sizeof(value)}
};
```

12.37.3 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.

### 12.37.4 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.

**13 Cryptoki tips and reminders**

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

**13.1 Operations, sessions, and threads**

In Cryptoki, there are several different types of operations which can be “active” in a 
session. An active operation is essentially one which takes more than one Cryptoki 
function call to perform. The types of active operations are object searching; encryption; 
decryption; message-digesting; signature with appendix; signature with recovery; 
verification with appendix; and verification with recovery.

A given session can have 0, 1, or 2 operations active at a time. It can only have 2 
operations active simultaneously if the token supports this; moreover, those two 
operations must be one of the four following pairs of operations: digesting and 
decryption; decryption and digesting; signing and encryption; decryption and 
verification.

If an application attempts to initialize an operation (make it active) in a session, but this 
cannot be accomplished because of some other active operation(s), the application 
receives the error value **CKR_OPERATION_ACTIVE**. This error value can also be 
received if a session has an active operation and the application attempts to use that 
session to perform any of various operations which do not become “active”, but which 
require cryptographic processing, such as using the token’s random number generator, or 
generating/wrapping/unwrapping/deriving a key.
To abandon an active operation an application may have to complete the operation and discard the result. Closing the session will also have this effect. Alternatively, the library may allow active operations to be abandoned by the application, simply by allowing initialization for some other operation. In this case CKR_OPERATION_ACTIVE will not be returned but the previous active operation will be unusable.

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

13.2 Multiple Application Access Behavior

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

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

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

13.3 Objects, attributes, and templates

In general, a Cryptoki function which requires a template for an object needs the template to specify—either explicitly or implicitly—any attributes that are not specified elsewhere. If a template specifies a particular attribute more than once, the function can return CKR_TEMPLATE_INVALID or it can choose a particular value of the attribute from among those specified and use that value. In any event, object attributes are always single-valued.
13.4 Signing with recovery

Signing with recovery is a general alternative to ordinary digital signatures (“signing
with appendix”) which is supported by certain mechanisms. Recall that for ordinary
digital signatures, a signature of a message is computed as some function of the message
and the signer’s private key; this signature can then be used (together with the message
and the signer’s public key) as input to the verification process, which yields a simple
“signature valid/signature invalid” decision.

Signing with recovery also creates a signature from a message and the signer’s private
key. However, to verify this signature, no message is required as input. Only the
signature and the signer’s public key are input to the verification process, and the
verification process outputs either “signature invalid” or—if the signature is valid—the
original message.

Consider a simple example with the CKM_RSA_X_509 mechanism. Here, a message is
a byte string which we will consider to be a number modulo \( n \) (the signer’s RSA
modulus). When this mechanism is used for ordinary digital signatures (signatures with
appendix), a signature is computed by raising the message to the signer’s private
exponent modulo \( n \). To verify this signature, a verifier raises the signature to the signer’s
public exponent modulo \( n \), and accepts the signature as valid if and only if the result
matches the original message.

If CKM_RSA_X_509 is used to create signatures with recovery, the signatures are
produced in exactly the same fashion. For this particular mechanism, \( \text{any} \) number
modulo \( n \) is a valid signature. To recover the message from a signature, the signature is
raised to the signer’s public exponent modulo \( n \).
A Manifest constants

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

```c
#define CK_INVALID_HANDLE      0
#define CKN_SURRENDER 0
#define CK_UNAVAILABLE_INFORMATION     (~0UL)
#define CK_EFFECTIVELY_INFINITE        0
#define CKF_DONT_BLOCK 1
#define CKF_ARRAY_ATTRIBUTE    0x40000000
#define CKU_SO                 0
#define CKU_USER               1
#define CKU_CONTEXT_SPECIFIC   2
#define CKS_RO_PUBLIC_SESSION  0
#define CKS_RO_USER_FUNCTIONS  1
#define CKS_RW_PUBLIC_SESSION  2
#define CKS_RW_USER_FUNCTIONS  3
#define CKS_RW_SO_FUNCTIONS    4
#define CKO_DATA               0x00000000
#define CKO_CERTIFICATE        0x00000001
#define CKO_PUBLIC_KEY         0x00000002
#define CKO_PRIVATE_KEY        0x00000003
#define CKO_SECRET_KEY         0x00000004
#define CKO_HW_FEATURE         0x00000005
#define CKO_DOMAIN_PARAMETERS  0x00000006
#define CKO_MECHANISM          0x00000007
#define CKO_VENDOR_DEFINED     0x80000000
#define CKH_MONOTONIC_COUNTER  0x00000001
#define CKH_CLOCK              0x00000002
#define CKH_USER_INTERFACE     0x00000003
#define CKH_VENDOR_DEFINED     0x80000000
#define CKK_RSA             0x00000000
#define CKK_DSA             0x00000001
#define CKK_DH              0x00000002
#define CKK_ECDSA           0x00000003
#define CKK_EC              0x00000003
#define CKK_X9_42_DH        0x00000004
#define CKK_KEA             0x00000005
#define CKK_GENERIC_SECRET  0x00000006
#define CKK_RC2             0x00000007
#define CKK_RC4             0x00000008
#define CKK_DES             0x00000009
#define CKK_DES2            0x0000000A
#define CKK_DES3            0x0000000B
#define CKK_CAST            0x0000000C
#define CKK_CAST3           0x0000000D
#define CKK_CAST5           0x0000000E
#define CKK_CAST128         0x0000000F
#define CKK_RC5             0x00000010
```

June 2004

Copyright © 2004 RSA Security Inc.
```c
#define CKK_IDEA            0x0000001A
#define CKK_SKIPJACK        0x0000001B
#define CKK_BATON           0x0000001C
#define CKK_JUNIPER         0x0000001D
#define CKK_CDMF            0x0000001E
#define CKK_AES             0x0000001F
#define CKK_BLOWFISH        0x00000020
#define CKK_TWOFISH         0x00000021
#define CKK_VENDOR_DEFINED  0x80000000
#define CKC_X_509           0x00000000
#define CKC_X_509_ATTR_CERT 0x00000001
#define CKC_WTLS            0x00000002
#define CKC_VENDOR_DEFINED  0x80000000
#define CKA_CLASS                       0x00000000
#define CKA_TOKEN                       0x00000001
#define CKA_PRIVATE                     0x00000002
#define CKA_LABEL                       0x00000003
#define CKA_APPLICATION                 0x00000010
#define CKA_VALUE                       0x00000011
#define CKA_OBJECT_ID                   0x00000012
#define CKA_CERTIFICATE_TYPE            0x00000080
#define CKA_ISSUER                      0x00000081
#define CKA_SERIAL_NUMBER               0x00000082
#define CKA_AC_ISSUER                   0x00000083
#define CKA_OWNER                       0x00000084
#define CKA_ATTR_TYPES                  0x00000085
#define CKA_TRUSTED                     0x00000086
#define CKA_CERTIFICATE_CATEGORY        0x00000087
#define CKA_JAVA_MIDP_SECURITY_DOMAIN   0x00000088
#define CKA_URL                         0x00000089
#define CKA_HASH_OF_SUBJECT_PUBLIC_KEY  0x0000008A
#define CKA_HASH_OF_ISSUER_PUBLIC_KEY   0x0000008B
#define CKA_CHECK_VALUE                 0x00000090
#define CKA_KEY_TYPE                    0x00000100
#define CKA_SUBJECT                     0x00000101
#define CKA_ID                          0x00000102
#define CKA_SENSITIVE                   0x00000103
#define CKA_ENCRYPT                     0x00000104
#define CKA_DECRYPT                     0x00000105
#define CKA_WRAP                        0x00000106
#define CKA_UNWRAP                      0x00000107
#define CKA_SIGN                        0x00000108
#define CKA_SIGN_RECOVER                0x00000109
#define CKA_VERIFY                      0x0000010A
#define CKA_VERIFY_RECOVER              0x0000010B
#define CKA_DERIVE                      0x0000010C
#define CKA_START_DATE                  0x00000110
#define CKA_END_DATE                    0x00000111
#define CKA_MODULUS                     0x00000120
#define CKA_MODULUS_BITS                0x00000121
#define CKA_PUBLIC_EXPONENT             0x00000122
#define CKA_PRIVATE_EXPONENT            0x00000123
#define CKA_PRIME_1                     0x00000124
#define CKA_PRIME_2                     0x00000125
#define CKA_EXPONENT_1                  0x00000126
#define CKA_EXPONENT_2                  0x00000127
#define CKA_COEFFICIENT                 0x00000128
#define CKA_PRIME                       0x00000130
```
A. MANIFEST CONSTANTS

#define CKA_SUBPRIME 0x00000131
#define CKA_BASE 0x00000132
#define CKA_PRIME_BITS 0x00000133
#define CKA_SUBPRIME_BITS 0x00000134
#define CKA_VALUE_BITS 0x00000160
#define CKA_VALUE_LEN 0x00000161
#define CKA_EXTRACTABLE 0x00000162
#define CKA_LOCAL 0x00000163
#define CKA_NEVER_EXTRACTABLE 0x00000164
#define CKA_ALWAYS_SENSITIVE 0x00000165
#define CKA_KEY_GEN_MECHANISM 0x00000166
#define CKA_MODIFIABLE 0x00000170
#define CKA_ECDSA_PARAMS 0x00000180
#define CKA_EC_PARAMS 0x00000180
#define CKA_EC_POINT 0x00000181
#define CKA_SECONDARY_AUTH 0x00000200 /* Deprecated */
#define CKA_AUTH_PIN_FLAGS 0x00000201 /* Deprecated */
#define CKA_ALWAYS_AUTHENTICATE 0x00000202
#define CKA_WRAP_WITH_TRUSTED 0x00000210
#define CKA_WRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE|0x00000211)
#define CKA_UNWRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE|0x00000212)
#define CKA_HW_FEATURE_TYPE 0x00000300
#define CKA_RESET_ON_INIT 0x00000301
#define CKA_HAS_RESET 0x00000302
#define CKA_PIXEL_X 0x00000400
#define CKA_PIXEL_Y 0x00000401
#define CKA_RESOLUTION 0x00000402
#define CKA_CHAR_ROWS 0x00000403
#define CKA_CHAR_COLUMNS 0x00000404
#define CKA_COLOR 0x00000405
#define CKA_BITS_PER_PIXEL 0x00000406
#define CKA_CHAR_SETS 0x00000407
#define CKA_ENCODING_METHODS 0x00000481
#define CKA_MIME_TYPES 0x00000482
#define CKA_MECHANISM_TYPE 0x00000500
#define CKA_REQUIRED_CMS_ATTRIBUTES 0x00000501
#define CKA_DEFAULT_CMS_ATTRIBUTES 0x00000502
#define CKA_SUPPORTED_CMS_ATTRIBUTES 0x00000503
#define CKA_ALLOWED_MECHANISMS (CKF_ARRAY_ATTRIBUTE|0x00000600)
#define CKA_VENDOR_DEFINED 0x80000000

#define CKM_RSA_PKCS_KEY_PAIR_GEN 0x00000000
#define CKM_RSA_PKCS 0x00000001
#define CKM_RSA_9796 0x00000002
#define CKM_RSA_X_509 0x00000003
#define CKM_MD2_RSA_PKCS 0x00000004
#define CKM_MD5_RSA_PKCS 0x00000005
#define CKM_SHA1_RSA_PKCS 0x00000006
#define CKM_RIPEMD128_RSA_PKCS 0x00000007
#define CKM_RIPEMD160_RSA_PKCS 0x00000008
#define CKM_RSA_PKCS_OAEP 0x00000009
#define CKM_RSA_X9_31_KEY_PAIR_GEN 0x0000000A
#define CKM_RSA_X9_31 0x0000000B
#define CKM_SHA1_RSA_X9_31 0x0000000C
#define CKM_RSA_PKCS_PSS 0x0000000D
#define CKM_SHA1_RSA_PKCS_PSS 0x0000000E
#define CKM_DSA_KEY_PAIR_GEN 0x00000010
#define CKM_DSA 0x00000011
#define CKM_DSA_SHA1 0x00000012
#define CKM_SHA256_HMAC_GENERAL 0x00000252
#define CKM_SHA384 0x00000260
#define CKM_SHA384_HMAC 0x00000261
#define CKM_SHA384_HMAC_GENERAL 0x00000262
#define CKM_SHA512 0x00000270
#define CKM_SHA512_HMAC 0x00000271
#define CKM_SHA512_HMAC_GENERAL 0x00000272
#define CKM_CAST_KEY_GEN 0x00000300
#define CKM_CAST_ECB 0x00000301
#define CKM_CAST_CBC 0x00000302
#define CKM_CAST_MAC 0x00000303
#define CKM_CAST_MAC_GENERAL 0x00000304
#define CKM_CAST_CBC_PAD 0x00000305
#define CKM_CAST3_KEY_GEN 0x00000310
#define CKM_CAST3_ECB 0x00000311
#define CKM_CAST3_CBC 0x00000312
#define CKM_CAST3_MAC 0x00000313
#define CKM_CAST3_MAC_GENERAL 0x00000314
#define CKM_CAST3_CBC_PAD 0x00000315
#define CKM_CAST5_KEY_GEN 0x00000320
#define CKM_CAST5_ECB 0x00000321
#define CKM_CAST5_CBC 0x00000322
#define CKM_CAST5_MAC 0x00000323
#define CKM_CAST5_MAC_GENERAL 0x00000324
#define CKM_CAST5_CBC_PAD 0x00000325
#define CKM_CAST128_KEY_GEN 0x00000320
#define CKM_CAST128_ECB 0x00000321
#define CKM_CAST128_CBC 0x00000322
#define CKM_CAST128_MAC 0x00000323
#define CKM_CAST128_MAC_GENERAL 0x00000324
#define CKM_CAST128_CBC_PAD 0x00000325
#define CKM_IDEA_KEY_GEN 0x00000340
#define CKM_IDEA_ECB 0x00000341
#define CKM_IDEA_CBC 0x00000342
#define CKM_IDEA_MAC 0x00000343
#define CKM_IDEA_MAC_GENERAL 0x00000344
#define CKM_IDEA_CBC_PAD 0x00000345
#define CKM_GENERIC_SECRET_KEY_GEN 0x00000350
#define CKM_CONCATENATE_BASE_AND_KEY 0x00000360
#define CKM_CONCATENATE_BASE_AND_DATA 0x00000362
#define CKM_CONCATENATE_DATA_AND_BASE 0x00000363
#define CKM_XOR_BASE_AND_DATA 0x00000364
#define CKM_EXTRACT_KEY_FROM_KEY 0x00000365
#define CKM_SSL3_PRE_MASTER_KEY_GEN 0x00000370
#define CKM_SSL3_MASTER_KEY_DERIVE 0x00000371
#define CKM_SSL3_KEY_AND_MAC_DERIVE 0x00000372
#define CKM_SSL3_MASTER_KEY_DERIVE_DH 0x00000373
#define CKM_TLS_PRE_MASTER_KEY_GEN 0x00000374
#define CKM_TLS_MASTER_KEY_DERIVE 0x00000375
#define CKM_TLS_KEY_AND_MAC_DERIVE 0x00000376
#define CKM_TLS_MASTER_KEY_DERIVE_DH 0x00000377
#define CKM_TLS_PRF 0x00000378
#define CKM_SSL3_MD5_MAC 0x00000380
#define CKM_SSL3_SHA1_MAC 0x00000381
A. MANIFEST CONSTANTS

#define CKM_JUNIPER_ECB128 0x00001061
#define CKM_JUNIPER_CBC128 0x00001062
#define CKM_JUNIPER_COUNTER 0x00001063
#define CKM_JUNIPER_SHUFFLE 0x00001064
#define CKM_JUNIPER_WRAP 0x00001065
#define CKM_FASTHASH 0x00001070
#define CKM_AES_KEY_GEN 0x00001080
#define CKM_AES_ECB 0x00001081
#define CKM_AES_CBC 0x00001082
#define CKM_AES_MAC 0x00001083
#define CKM_AES_MAC_GENERAL 0x00001084
#define CKM_BLOWFISH_KEY_GEN 0x00001090
#define CKM_BLOWFISH_CBC 0x00001091
#define CKM_TWOFISH_KEY_GEN 0x00001092
#define CKM_TWOFISH_CBC 0x00001093
#define CKM_DES_ECB_ENCRYPT_DATA 0x00001100
#define CKM_DES_CBC_ENCRYPT_DATA 0x00001101
#define CKM_DES3_ECB_ENCRYPT_DATA 0x00001102
#define CKM_DES3_CBC_ENCRYPT_DATA 0x00001103
#define CKM_AES_ECB_ENCRYPT_DATA 0x00001104
#define CKM_AES_CBC_ENCRYPT_DATA 0x00001105
#define CKM_DSA_PARAMETER_GEN 0x00002000
#define CKM_DH_PKCS_PARAMETER_GEN 0x00002001
#define CKM_X9_42_DH_PARAMETER_GEN 0x00002002
#define CKM_VENDOR_DEFINED 0x80000000

#define CKR_OK 0x00000000
#define CKR_CANCEL 0x00000001
#define CKR_HOST_MEMORY 0x00000002
#define CKR_SLOT_ID_INVALID 0x00000003
#define CKR_GENERAL_ERROR 0x00000005
#define CKR_FUNCTION_FAILED 0x00000006
#define CKR_ARGUMENTS_BAD 0x00000007
#define CKR_NO_EVENT 0x00000008
#define CKR_NEED_TO_CREATE_THREADS 0x00000009
#define CKR_CANT_LOCK 0x0000000A
#define CKR_ATTRIBUTE_READ_ONLY 0x00000010
#define CKR_ATTRIBUTE_SENSITIVE 0x00000011
#define CKR_ATTRIBUTE_TYPE_INVALID 0x00000012
#define CKR_ATTRIBUTE_VALUE_INVALID 0x00000013
#define CKR_DATA_INVALID 0x00000020
#define CKR_DATA_LEN_RANGE 0x00000021
#define CKR_FUNCTION_CANCELED 0x00000050
#define CKR_FUNCTION_NOT_PARALLEL 0x00000051
#define CKR_FUNCTION_NOT_SUPPORTED 0x00000054
#define CKR_KEY_HANDLE_INVALID 0x00000060
#define CKR_KEY_SIZE_RANGE 0x00000062
#define CKR_KEY_TYPE_INCONSISTENT 0x00000063
#define CKR_KEY_NOT_NEEDED 0x00000064
#define CKR_KEY_CHANGED 0x00000065
#define CKR_KEY_NEEDED 0x00000066
#define CKR_KEY_INDIGESTIBLE 0x00000067
#define CKR_KEY_FUNCTION_NOT_PERMITTED 0x00000068
#define CKR_KEY_NOT_WRAPPABLE 0x00000069
#define CKR_KEY_UNEXTRACTABLE        0x0000006A
#define CKR_MECHANISM_INVALID       0x00000070
#define CKR_MECHANISM_PARAM_INVALID 0x00000071
#define CKR_OBJECT_HANDLE_INVALID  0x00000082
#define CKR_OPERATION_ACTIVE       0x00000090
#define CKR_OPERATION_NOT_INITIALIZED 0x00000091
#define CKR_PIN_INCORRECT          0x000000A0
#define CKR_PIN_INVALID            0x000000A1
#define CKR_PIN_LEN_RANGE          0x000000A2
#define CKR_PIN_EXPIRED            0x000000A3
#define CKR_PIN_LOCKED             0x000000A4
#define CKR_SESSION_CLOSED         0x000000B0
#define CKR_SESSION_COUNT          0x000000B1
#define CKR_SESSION_HANDLE_INVALID 0x000000B3
#define CKR_SESSION_PARALLEL_NOT_SUPPORTED 0x000000B4
#define CKR_SESSION_READ_ONLY      0x000000B5
#define CKR_SESSION_EXISTS         0x000000B6
#define CKR_SESSION_READ_ONLY_EXISTS 0x000000B7
#define CKR_SESSION_READ_WRITE_SO_EXISTS 0x000000B8
#define CKR_SIGNATURE_INVALID      0x000000C0
#define CKR_SIGNATURE_LEN_RANGE    0x000000C1
#define CKR_TEMPLATE_INCOMPLETE    0x000000D0
#define CKR_TEMPLATE_INCONSISTENT  0x000000D1
#define CKR_TOKEN_NOT_PRESENT      0x000000E0
#define CKR_TOKEN_NOT_RECOGNIZED   0x000000E1
#define CKR_TOKEN_WRITE_PROTECTED  0x000000E2
#define CKR_UNWRAPPING_KEY_HANDLE_INVALID 0x000000F0
#define CKR_UNWRAPPING_KEY_SIZE_RANGE 0x000000F1
#define CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT 0x000000F2
#define CKR_USER_ALREADY_LOGGED_IN  0x00000100
#define CKR_USER_NOT_LOGGED_IN     0x00000101
#define CKR_USER_PIN_NOT_INITIALIZED 0x00000102
#define CKR_USER_TYPE_INVALID      0x00000103
#define CKR_USER_ANOTHER_ALREADY_LOGGED_IN 0x00000104
#define CKR_USER_TOO_MANY_TYPES   0x00000105
#define CKR_WRAPPED_KEY_INVALID    0x00000110
#define CKR_WRAPPED_KEY_LEN_RANGE  0x00000112
#define CKR_WRAPPING_KEY_HANDLE_INVALID 0x00000113
#define CKR_WRAPPING_KEY_SIZE_RANGE 0x00000114
#define CKR_WRAPPING_KEY_TYPE_INCONSISTENT 0x00000115
#define CKR_RANDOM_SEED_NOT_SUPPORTED 0x00000120
#define CKR_RANDOM_NO_RNG          0x00000121
#define CKR_DOMAIN_PARAMS_INVALID  0x00000130
#define CKR_BUFFER_TOO_SMALL      0x00000150
#define CKR_SAVED_STATE_INVALID   0x00000160
#define CKR_INFORMATION_SENSITIVE 0x00000170
#define CKR_STATE_UNSAVEABLE      0x00000180
#define CKR_CRYPTOKI_NOT_INITIALIZED 0x00000190
#define CKR_CRYPTOKI_ALREADY_INITIALIZED 0x00000191
#define CKR_MUTEX_BAD             0x000001A0
#define CKR_MUTEX_NOT_LOCKED      0x000001A1
#define CKR_FUNCTION_REJECTED    0x00000200
#define CKR_VENDOR_DEFINED        0x80000000

B Token profiles

This appendix describes “profiles,” i.e., sets of mechanisms, which a token should support for various common types of application. It is expected that these sets would be
standardized as parts of the various applications, for instance within a list of requirements on the module that provides cryptographic services to the application (which may be a Cryptoki token in some cases). Thus, these profiles are intended for reference only at this point, and are not part of this standard.

The following table summarizes the mechanisms relevant to two common types of applications:

Table B-1, Mechanisms and profiles

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Government Authentication-only</th>
<th>Cellular Digital Packet Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKM_DSA_KEY_PAIR_GEN</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DSA</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CKM_DH_PKCS_KEY_PAIR_GEN</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_DH_PKCS_DERIVE</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC4_KEY_GEN</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_RC4</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CKM_SHA_1</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

**B.1 Government authentication-only**

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

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

**B.2 Cellular Digital Packet Data**

Cellular Digital Packet Data (CDPD) is a set of protocols for wireless communication. The basic set of mechanisms to support CDPD applications includes the following:

- Diffie-Hellman key generation (256-1024 bits)
- Diffie-Hellman key derivation (256-1024 bits)
- RC4 key generation (40-128 bits)
- RC4 (40-128 bits)

(The initial CDPD security specification limits the size of the Diffie-Hellman key to 256 bits, but it has been recommended that the size be increased to at least 512 bits.)
B.3 Other profiles

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

This appendix compares Cryptoki with the following cryptographic APIs:

- X/Open GCS-API - Generic Cryptographic Service API, Draft 2, February 14, 1995

C.1 FORTEZZA CIPG, Rev. 1.52

This document defines an API to the FORTEZZA PCMCIA Crypto Card. It is at a level similar to Cryptoki. The following table lists the FORTEZZA CIPG functions, together with the equivalent Cryptoki functions:

Table C-1, FORTEZZA CIPG vs. Cryptoki

<table>
<thead>
<tr>
<th>FORTEZZA CIPG</th>
<th>Equivalent Cryptoki</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI_ChangePIN</td>
<td>C_InitPIN, C_SetPIN</td>
</tr>
<tr>
<td>CI_CheckPIN</td>
<td>C_Login</td>
</tr>
<tr>
<td>CI_Close</td>
<td>C_CloseSession</td>
</tr>
<tr>
<td>CI_Decrypt</td>
<td>C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal</td>
</tr>
<tr>
<td>CI_DeleteCertificate</td>
<td>C_DestroyObject</td>
</tr>
<tr>
<td>CI_DeleteKey</td>
<td>C_DestroyObject</td>
</tr>
<tr>
<td>CI_Encrypt</td>
<td>C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal</td>
</tr>
<tr>
<td>CI_ExtractX</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>CI_GenerateIV</td>
<td>C_GenerateRandom</td>
</tr>
<tr>
<td>CI_GenerateMEK</td>
<td>C_GenerateKey</td>
</tr>
<tr>
<td>CI_GenerateRa</td>
<td>C_GenerateRandom</td>
</tr>
<tr>
<td>CI_GenerateRandom</td>
<td>C_GenerateRandom</td>
</tr>
<tr>
<td>CI_GenerateTEK</td>
<td>C_GenerateKey</td>
</tr>
<tr>
<td>CI_GenerateX</td>
<td>C_GenerateKeyPair</td>
</tr>
<tr>
<td>CI_GetCertificate</td>
<td>C_FindObjects</td>
</tr>
<tr>
<td>CI_Configuration</td>
<td>C_GetTokenInfo</td>
</tr>
<tr>
<td>CI_GetHash</td>
<td>C_DigestInit, C_Digest, C_DigestUpdate, and C_DigestFinal</td>
</tr>
<tr>
<td>CI_GetIV</td>
<td>No equivalent</td>
</tr>
<tr>
<td>CI_GetPersonalityList</td>
<td>C_FindObjects</td>
</tr>
<tr>
<td>CI_GetState</td>
<td>C_GetSessionInfo</td>
</tr>
<tr>
<td>CI_GetStatus</td>
<td>C_GetTokenInfo</td>
</tr>
<tr>
<td>FORTEZZA CIPG</td>
<td>Equivalent Cryptoki</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CI_GetTime</td>
<td>C_GetTokenInfo or</td>
</tr>
<tr>
<td></td>
<td>C_GetAttributeValue(clock object) [preferred]</td>
</tr>
<tr>
<td>CI_Hash</td>
<td>C_DigestInit, C_Digest, C_DigestUpdate, and</td>
</tr>
<tr>
<td></td>
<td>C_DigestFinal</td>
</tr>
<tr>
<td>CI_Initialize</td>
<td>C_Initialize</td>
</tr>
<tr>
<td>CI_InitializeHash</td>
<td>C_DigestInit</td>
</tr>
<tr>
<td>CI_InstallX</td>
<td>C_UnwrapKey</td>
</tr>
<tr>
<td>CI_LoadCertificate</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>CI_LoadDSAParameters</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>CI_LoadInitValues</td>
<td>C_SeedRandom</td>
</tr>
<tr>
<td>CI_LoadIV</td>
<td>C_EncryptInit, C_DecryptInit</td>
</tr>
<tr>
<td>CI_LoadK</td>
<td>C_SignInit</td>
</tr>
<tr>
<td>CI_LoadPublicKeyParameters</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>CI_LoadPIN</td>
<td>C_SetPIN</td>
</tr>
<tr>
<td>CI_LoadX</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>CI_Lock</td>
<td>Implicit in session management</td>
</tr>
<tr>
<td>CI_Open</td>
<td>C_OpenSession</td>
</tr>
<tr>
<td>CI_RelayX</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>CI_Reset</td>
<td>C_CloseAllSessions</td>
</tr>
<tr>
<td>CI_Restore</td>
<td>Implicit in session management</td>
</tr>
<tr>
<td>CI_Save</td>
<td>Implicit in session management</td>
</tr>
<tr>
<td>CI_Select</td>
<td>C_OpenSession</td>
</tr>
<tr>
<td>CI_SetConfiguration</td>
<td>No equivalent</td>
</tr>
<tr>
<td>CI_SetKey</td>
<td>C_EncryptInit, C_DecryptInit</td>
</tr>
<tr>
<td>CI_SetMode</td>
<td>C_EncryptInit, C_DecryptInit</td>
</tr>
<tr>
<td>CI_SetPersonality</td>
<td>C_CreateObject</td>
</tr>
<tr>
<td>CI_SetTime</td>
<td>No equivalent</td>
</tr>
<tr>
<td>CI_Sign</td>
<td>C_SignInit, C_Sign</td>
</tr>
<tr>
<td>CI_Terminate</td>
<td>C_CloseAllSessions</td>
</tr>
<tr>
<td>CI_Timestamp</td>
<td>C_SignInit, C_Sign</td>
</tr>
<tr>
<td>CI_Unlock</td>
<td>Implicit in session management</td>
</tr>
<tr>
<td>CI_UnwrapKey</td>
<td>C_UnwrapKey</td>
</tr>
<tr>
<td>CI.VerifySignature</td>
<td>C_VerifyInit, C_Verify</td>
</tr>
<tr>
<td>CI.VerifyTimestamp</td>
<td>C_VerifyInit, C_Verify</td>
</tr>
<tr>
<td>CI.WrapKey</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>CI_Zeroize</td>
<td>C_InitToken</td>
</tr>
</tbody>
</table>
C. COMPARISON OF CRYPTOKI AND OTHER APIs

C.2 GCS-API

This proposed standard defines an API to high-level security services such as authentication of identities and data-origin, non-repudiation, and separation and protection. It is at a higher level than Cryptoki. The following table lists the GCS-API functions with the Cryptoki functions used to implement the functions. Note that full support of GCS-API is left for future versions of Cryptoki.

<table>
<thead>
<tr>
<th>GCS-API</th>
<th>Cryptoki implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieve_CC</td>
<td></td>
</tr>
<tr>
<td>release_CC</td>
<td></td>
</tr>
<tr>
<td>generate_hash</td>
<td>C_DigestInit, C_Digest</td>
</tr>
<tr>
<td>generate_random_number</td>
<td>C_GenerateRandom</td>
</tr>
<tr>
<td>generate_checkvalue</td>
<td>C_SignInit, C_Sign, C_SignUpdate, C_SignFinal</td>
</tr>
<tr>
<td>verify_checkvalue</td>
<td>C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal</td>
</tr>
<tr>
<td>data_encipher</td>
<td>C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal</td>
</tr>
<tr>
<td>data_decipher</td>
<td>C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal</td>
</tr>
<tr>
<td>create_CC</td>
<td></td>
</tr>
<tr>
<td>derive_key</td>
<td>C_DeriveKey</td>
</tr>
<tr>
<td>generate_key</td>
<td>C_GenerateKey</td>
</tr>
<tr>
<td>store_CC</td>
<td></td>
</tr>
<tr>
<td>delete_CC</td>
<td></td>
</tr>
<tr>
<td>replicate_CC</td>
<td></td>
</tr>
<tr>
<td>export_key</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>import_key</td>
<td>C_UnwrapKey</td>
</tr>
<tr>
<td>archive_CC</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>restore_CC</td>
<td>C_UnwrapKey</td>
</tr>
<tr>
<td>set_key_state</td>
<td></td>
</tr>
<tr>
<td>generate_key_pattern</td>
<td></td>
</tr>
<tr>
<td>verify_key_pattern</td>
<td></td>
</tr>
<tr>
<td>derive_clear_key</td>
<td>C_DeriveKey</td>
</tr>
<tr>
<td>generate_clear_key</td>
<td>C_GenerateKey</td>
</tr>
<tr>
<td>load_key_parts</td>
<td></td>
</tr>
<tr>
<td>clear_key_encipher</td>
<td>C_WrapKey</td>
</tr>
<tr>
<td>clear_key_decipher</td>
<td>C_UnwrapKey</td>
</tr>
<tr>
<td>GCS-API</td>
<td>Cryptoki implementation</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>change_key_context</td>
<td></td>
</tr>
<tr>
<td>load_initial_key</td>
<td></td>
</tr>
<tr>
<td>generate_initial_key</td>
<td></td>
</tr>
<tr>
<td>set_current_master_key</td>
<td></td>
</tr>
<tr>
<td>protect_under_new_master_key</td>
<td></td>
</tr>
<tr>
<td>protect_under_current_master_key</td>
<td></td>
</tr>
<tr>
<td>initialise_random_number_generator</td>
<td>C_SeedRandom</td>
</tr>
<tr>
<td>install_algorithm</td>
<td></td>
</tr>
<tr>
<td>de_install_algorithm</td>
<td></td>
</tr>
<tr>
<td>disable_algorithm</td>
<td></td>
</tr>
<tr>
<td>enable_algorithm</td>
<td></td>
</tr>
<tr>
<td>set_defaults</td>
<td></td>
</tr>
</tbody>
</table>
D Intellectual property considerations

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

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

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

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

RSA Security Inc. makes no other representations regarding intellectual property claims by other parties. Such determination is the responsibility of the user.
E  Method for Exposing Multiple-PINs on a Token Through Cryptoki (deprecated)

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

This is the initial version of PKCS #11 v2.20.