GlobalPlatform Technology
TEE Internal Core API Specification
Version 1.1.2.50 (Target v1.2)

Public Review
June 2018
Document Reference: GPD_SPE_010
THIS SPECIFICATION OR OTHER WORK PRODUCT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE COMPANY, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER DIRECTLY OR INDIRECTLY ARISING FROM THE IMPLEMENTATION OF THIS SPECIFICATION OR OTHER WORK PRODUCT.
Contents

1 Introduction ..........................................................................................................................  14

1.1 Audience .........................................................................................................................  14

1.2 IPR Disclaimer ..................................................................................................................  14

1.3 References .......................................................................................................................  15

1.4 Terminology and Definitions ............................................................................................  16

1.5 Abbreviations and Notations ............................................................................................  20

1.6 Revision History ..............................................................................................................  22

2 Overview of the TEE Internal Core API Specification .......................................................  24

2.1 Trusted Applications .........................................................................................................  25

2.1.1 TA Interface ..................................................................................................................  26

2.1.2 Instances, Sessions, Tasks, and Commands .................................................................  27

2.1.3 Sequential Execution of Entry Points ..........................................................................  27

2.1.4 Cancellations ...............................................................................................................  27

2.1.5 Unexpected Client Termination ....................................................................................  28

2.1.6 Instance Types ............................................................................................................  28

2.1.7 Configuration, Development, and Management .........................................................  28

2.2 TEE Internal Core APIs ..................................................................................................  29

2.2.1 Trusted Core Framework API .....................................................................................  29

2.2.2 Trusted Storage API for Data and Keys .......................................................................  29

2.2.3 Cryptographic Operations API ...................................................................................  30

2.2.4 Time API ....................................................................................................................  30

2.2.5 TEE Arithmetical API ..................................................................................................  30

2.2.6 Peripheral and Event API ...........................................................................................  31

2.3 Error Handling ................................................................................................................  31

2.3.1 Normal Errors ............................................................................................................  31

2.3.2 Programmer Errors .....................................................................................................  31

2.3.3 Panics ........................................................................................................................  32

2.4 Opaque Handles ..............................................................................................................  34

2.5 Properties .......................................................................................................................  35

2.6 Peripheral Support ..........................................................................................................  35

3 Common Definitions ............................................................................................................  36

3.1 Header File .......................................................................................................................  36

3.1.1 API Version ...............................................................................................................  36

3.1.2 Target and Version Optimization ..............................................................................  37

3.1.3 Peripherals Support ....................................................................................................  37

3.2 Data Types ......................................................................................................................  38

3.2.1 Basic Types ...............................................................................................................  38

3.2.2 Bit Numbering ..........................................................................................................  38

3.2.3 TEE_Result, TEEC_Result .......................................................................................  39

3.2.4 TEE_UUID, TEEC_UUID .........................................................................................  40

3.3 Constants .......................................................................................................................  41

3.3.1 Return Code Ranges and Format ...............................................................................  41

3.3.2 Return Codes .............................................................................................................  42

3.4 Parameter Annotations ...................................................................................................  44

3.4.1 [in], [out], and [inout] ..............................................................................................  44

3.4.2 [outopt] ....................................................................................................................  44

3.4.3 [inbuf] .......................................................................................................................  45

3.4.4 [outbuf] ....................................................................................................................  45

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
4.1 Data Types ................................................................. 50
  4.1.1 TEE_Identity .................................................. 50
  4.1.2 TEE_Param .................................................... 50
  4.1.3 TEE_TASessionHandle ................................. 50
  4.1.4 TEE_PropSetHandle ...................................... 51

4.2 Constants .................................................................. 52
  4.2.1 Parameter Types ........................................... 52
  4.2.2 Login Types .................................................. 52
  4.2.3 Origin Codes ................................................ 53
  4.2.4 Property Set Pseudo-Handles ....................... 53
  4.2.5 Memory Access Rights .................................. 53

4.3 TA Interface ........................................................... 55
  4.3.1 TA_CreateEntryPoint .................................. 58
  4.3.2 TA_DestroyEntryPoint ................................. 58
  4.3.3 TA_OpenSessionEntryPoint ......................... 59
  4.3.4 TA_CloseSessionEntryPoint ......................... 61
  4.3.5 TA_InvokeCommandEntryPoint ..................... 62
  4.3.6 Operation Parameters in the TA Interface .......... 63
    4.3.6.1 Content of paramTypes Argument ........... 63
    4.3.6.2 Initial Content of params Argument ........ 64
    4.3.6.3 Behavior of the Framework when the Trusted Application Returns .......... 65
    4.3.6.4 Memory Reference and Memory Synchronization .......... 66

4.4 Property Access Functions ...................................... 67
  4.4.1 TEE_GetPropertyAsString ............................ 69
  4.4.2 TEE_GetPropertyAsBool ............................... 70
  4.4.3 TEE_GetPropertyAsUnn ................................. 71
    4.4.3.1 TEE_GetPropertyAsU32 ........................ 71
    4.4.3.2 TEE_GetPropertyAsU64 ........................ 72
  4.4.4 TEE_GetPropertyAsBinaryBlock ................. 73
  4.4.5 TEE_GetPropertyAsUUID .............................. 74
  4.4.6 TEE_GetPropertyAsIdentity ......................... 75
  4.4.7 TEE_AllocatePropertyEnumerator ................. 76
  4.4.8 TEE_FreePropertyEnumerator ...................... 77
  4.4.9 TEE_StartPropertyEnumerator ..................... 77
  4.4.10 TEE_ResetPropertyEnumerator .................... 78
  4.4.11 TEE_GetPropertyName ............................... 79
  4.4.12 TEE_GetNextProperty ............................... 80

4.5 Trusted Application Configuration Properties .......... 81

4.6 Client Properties .................................................. 84

4.7 Implementation Properties .................................... 86
  4.7.1 Specification Version Number Property .................. 92

4.8 Panics .................................................................. 93
  4.8.1 TEE_Panic ................................................... 93

4.9 Internal Client API .................................................. 94
  4.9.1 TEE_OpenTASession .................................... 94

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this
information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly
prohibited.
5 Trusted Storage API for Data and Keys .......................................................... 117

5.1 Summary of Features and Design ............................................................. 117

5.2 Trusted Storage and Rollback Detection ................................................. 119

5.3 Data Types ............................................................................................... 120

5.3.1 TEE_Attribute ..................................................................................... 120

5.3.2 TEE_ObjectInfo .................................................................................. 120

5.3.3 TEE_Whence ....................................................................................... 121

5.3.4 TEE_ObjectHandle ............................................................................. 121

5.3.5 TEE_ObjectEnumHandle .................................................................... 121

5.4 Constants ................................................................................................. 122

5.4.1 Constants Used in Trusted Storage API for Data and Keys ................. 122

5.4.2 Constants Used in Cryptographic Operations API ............................. 123

5.5 Generic Object Functions ........................................................................ 124

5.5.1 TEE_GetObjectInfo1 ........................................................................... 124

5.5.2 TEE_RestrictObjectUsage1 ................................................................. 126

5.5.3 TEE_GetObjectBufferAttribute ......................................................... 127

5.5.4 TEE_GetObjectValueAttribute .......................................................... 129

5.5.5 TEE_CloseObject ................................................................................ 130

5.6 Transient Object Functions ..................................................................... 131

5.6.1 TEE_AllocateTransientObject ......................................................... 131

5.6.2 TEE_FreeTransientObject ................................................................. 135

5.6.3 TEE_ResetTransientObject ............................................................... 135

5.6.4 TEE_PopulateTransientObject ......................................................... 136

5.6.5 TEE_InitRefAttribute, TEE_InitValueAttribute ............................. 141

5.6.6 TEE_CopyObjectAttributes1 ............................................................... 143

5.6.7 TEE_GenerateKey ............................................................................. 145

5.7 Persistent Object Functions ..................................................................... 149

5.7.1 TEE_OpenPersistentObject ............................................................... 149

5.7.2 TEE_CreatePersistentObject ............................................................ 151

5.7.3 Persistent Object Sharing Rules ......................................................... 154

5.7.4 TEE_CloseAndDeletePersistentObject1 .......................................... 156

5.7.5 TEE_RenamePersistentObject .......................................................... 157

5.8 Persistent Object Enumeration Functions ............................................. 158

5.8.1 TEE_AllocatePersistentObjectEnumerator ...................................... 158

5.8.2 TEE_FreePersistentObjectEnumerator ............................................ 158
6.10.1 Cryptographic Operations API .......................................................................................................................... 168
6.1 Data Types ..................................................................................................................................................................... 170
6.1.1 TEE_OperationMode .................................................................................................................................................. 170
6.1.2 TEE_OperationInfo .................................................................................................................................................... 171
6.1.3 TEE_OperationInfoMultiple .................................................................................................................................. 171
6.1.4 TEE_OperationHandle ............................................................................................................................................. 172
6.2 Generic Operation Functions ........................................................................................................................................ 173
6.2.1 TEE_AllocateOperation ........................................................................................................................................... 173
6.2.2 TEE_FreeOperation ................................................................................................................................................... 177
6.2.3 TEE_GetOperationInfo ............................................................................................................................................. 178
6.2.4 TEE_GetOperationInfoMultiple .............................................................................................................................. 179
6.2.5 TEE_ResetOperation .................................................................................................................................................. 181
6.2.6 TEE_SetOperationKey ............................................................................................................................................ 182
6.2.7 TEE_SetOperationKey2 .......................................................................................................................................... 184
6.2.8 TEE_CopyOperation ................................................................................................................................................ 186
6.2.9 TEE_IsAlgorithmSupported .................................................................................................................................. 187
6.3 Message Digest Functions ............................................................................................................................................ 188
6.3.1 TEE_DigestUpdate .................................................................................................................................................... 188
6.3.2 TEE_DigestDoFinal .................................................................................................................................................... 189
6.4 Symmetric Cipher Functions ...................................................................................................................................... 190
6.4.1 TEE_CipherInit ......................................................................................................................................................... 190
6.4.2 TEE_CipherUpdate .................................................................................................................................................. 192
6.4.3 TEE_CipherDoFinal ................................................................................................................................................ 193
6.5 MAC Functions ............................................................................................................................................................ 194
6.5.1 TEE_MACInit ........................................................................................................................................................... 194
6.5.2 TEE_MACUpdate ...................................................................................................................................................... 195
6.5.3 TEE_MACComputeFinal ......................................................................................................................................... 196
6.5.4 TEE_MACCompareFinal ..................................................................................................................................... 197
6.6 Authenticated Encryption Functions .......................................................................................................................... 198
6.6.1 TEE_AEInit ............................................................................................................................................................... 198
6.6.2 TEE_AEUpdateAAD ................................................................................................................................................. 200
6.6.3 TEE_AEUpdate ....................................................................................................................................................... 201
6.6.4 TEE_AESCryptFinal ................................................................................................................................................. 202
6.6.5 TEE_AEDecryptFinal ............................................................................................................................................ 203
6.7 Asymmetric Functions ................................................................................................................................................ 204
6.7.1 TEE_AsymmetricEncrypt, TEE_AsymmetricDecrypt .................................................................................................. 204
6.7.2 TEE_AsymmetricSignDigest .................................................................................................................................. 206
6.7.3 TEE_AsymmetricVerifyDigest .................................................................................................................................. 209
6.8 Key Derivation Functions ............................................................................................................................................ 212
6.8.1 TEE_DeriveKey ......................................................................................................................................................... 212
6.9 Random Data Generation Function ........................................................................................................................... 215
6.9.1 TEE_GenerateRandom ........................................................................................................................................... 215
6.10 Cryptographic Algorithms Specification .................................................................................................................. 216
6.10.1 List of Algorithm Identifiers .................................................................................................................................. 216
7 Time API .............................................................................................................................. 226
7.1 Data Types .......................................................................................................................... 226
    7.1.1 TEE_Time ......................................................................................................................... 226
7.2 Time Functions ..................................................................................................................... 227
    7.2.1 TEE_GetSystemTime ......................................................................................................... 227
    7.2.2 TEE_Wait ......................................................................................................................... 228
    7.2.3 TEE_GetTAPersistentTime .............................................................................................. 229
    7.2.4 TEE_SetTAPersistentTime .............................................................................................. 231
    7.2.5 TEE_GetREETime ............................................................................................................ 232
8 TEE Arithmetical API ........................................................................................................ 233
8.1 Introduction ......................................................................................................................... 233
8.2 Error Handling and Parameter Checking ............................................................................ 233
8.3 Data Types .......................................................................................................................... 234
    8.3.1 TEE_BigInt ...................................................................................................................... 234
    8.3.2 TEE_BigIntFMMContext ................................................................................................. 235
    8.3.3 TEE_BigIntFMM ............................................................................................................... 235
8.4 Memory Allocation and Size of Objects .............................................................................. 236
    8.4.1 TEE_BigIntSizeInU32 ..................................................................................................... 236
    8.4.2 TEE_BigIntFMMContextSizeInU32 ............................................................................... 237
    8.4.3 TEE_BigIntFMMSizeInU32 ............................................................................................. 238
8.5 Initialization Functions ....................................................................................................... 239
    8.5.1 TEE_BigIntInit .................................................................................................................. 239
    8.5.2 TEE_BigIntInitFMMContext ......................................................................................... 240
    8.5.3 TEE_BigIntInitFMM ....................................................................................................... 241
8.6 Converter Functions ........................................................................................................... 242
    8.6.1 TEE_BigIntConvertFromOctetString ............................................................................. 242
    8.6.2 TEE_BigIntConvertToOctetString ............................................................................... 243
    8.6.3 TEE_BigIntConvertFromS32 ......................................................................................... 244
    8.6.4 TEE_BigIntConvertToS32 ............................................................................................. 245
8.7 Logical Operations ............................................................................................................. 246
    8.7.1 TEE_BigIntCmp ................................................................................................................ 246
    8.7.2 TEE_BigIntCmpS32 ........................................................................................................ 246
    8.7.3 TEE_BigIntShiftRight ................................................................................................. 247
    8.7.4 TEE_BigIntGetBit .......................................................................................................... 248
    8.7.5 TEE_BigIntGetBitCount .............................................................................................. 248
    8.7.6 TEE_BigIntSetBit .......................................................................................................... 249
    8.7.7 TEE_BigIntAssign ......................................................................................................... 250
    8.7.8 TEE_BigIntAbs .............................................................................................................. 251
8.8 Basic Arithmetic Operations .............................................................................................. 252
    8.8.1 TEE_BigIntAdd ............................................................................................................... 252
    8.8.2 TEE_BigIntSub ............................................................................................................... 253
    8.8.3 TEE_BigIntNeg ............................................................................................................... 254
    8.8.4 TEE_BigIntMul ............................................................................................................... 255
    8.8.5 TEE_BigIntSquare ....................................................................................................... 256
    8.8.6 TEE_BigIntDiv ............................................................................................................... 257
8.9 Modular Arithmetic Operations ......................................................................................... 259
    8.9.1 TEE_BigIntMod .............................................................................................................. 259
    8.9.2 TEE_BigIntAddMod ....................................................................................................... 260
8.9.3 TEE_BigIntSubMod .............................................. 261
8.9.4 TEE_BigIntMulMod ........................................... 262
8.9.5 TEE_BigIntSquareMod ........................................ 263
8.9.6 TEE_BigIntInvMod ............................................... 264
8.9.7 TEE_BigIntExpMod .............................................. 265
8.10 Other Arithmetic Operations .................................. 266
8.10.1 TEE_BigIntRelativePrime ................................. 266
8.10.2 TEE_BigIntComputeExtendedGcd ...................... 267
8.10.3 TEE_BigIntIsProbablePrime .............................. 268
8.11 Fast Modular Multiplication Operations .................. 269
8.11.1 TEE_BigIntConvertToFMM ................................. 269
8.11.2 TEE_BigIntConvertFromFMM ............................... 270
8.11.3 TEE_BigIntComputeFMM .................................... 271

9 Peripheral and Event APIs ............................................ 272
9.1 Introduction .......................................................... 272
9.1.1 Peripherals .......................................................... 272
9.1.1.1 Access to Peripherals from a TA ...................... 273
9.1.1.1.1 Multiple Access to Peripherals (informative) ...... 273
9.1.2 Event Loop .......................................................... 274
9.1.3 Peripheral State ................................................... 274
9.1.4 Overview of Peripheral and Event APIs ................ 274
9.2 Constants .............................................................. 277
9.2.1 Handles ............................................................. 277
9.2.2 Maximum Sizes .................................................. 277
9.2.3 TEE_EVENT_TYPE ............................................. 277
9.2.4 TEE_PERIPHERAL_TYPE ................................. 279
9.2.5 TEE_PERIPHERAL_FLAGS ................................. 280
9.2.6 TEE_PeripheralStateId Values ......................... 281
9.3 Peripheral State Table ............................................ 282
9.3.1 Peripheral Name .................................................. 282
9.3.2 Firmware Information ........................................ 282
9.3.3 Manufacturer ..................................................... 283
9.3.4 Flags ................................................................. 283
9.3.5 Exclusive Access ............................................... 283
9.4 Operating System Pseudo-peripheral ..................... 284
9.4.1 State Table .......................................................... 284
9.4.2 Events ............................................................... 284
9.5 Session Pseudo-peripheral .................................... 285
9.5.1 State Table .......................................................... 285
9.5.2 Events ............................................................... 285
9.6 Data Structures ...................................................... 286
9.6.1 TEE_Peripheral .................................................. 286
9.6.2 TEE_PeripheralDescriptor ................................. 287
9.6.3 TEE_PeripheralHandle ....................................... 287
9.6.4 TEE_PeripheralId ................................................ 288
9.6.5 TEE_PeripheralState .......................................... 288
9.6.6 TEE_PeripheralStateId ....................................... 289
9.6.7 TEE_PeripheralValueType ................................. 289
9.6.8 TEE_Event .......................................................... 290
9.6.9 Generic Payloads ............................................... 291
9.6.9.1 TEE_Event_AccessChange ............................. 291
9.6.9.2 TEE_Event_ClientCancel ................................. 291

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this
information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly
prohibited.
9.6.9.3 TEE_Event_Timer................................................................. 291
9.6.10 TEE_EventQueueHandle...................................................... 292
9.6.11 TEE_EventSourceHandle.................................................... 292
9.7 Peripheral API Functions ......................................................... 293
  9.7.1 TEE_Peripheral_Close.......................................................... 293
  9.7.2 TEE_Peripheral_CloseMultiple.......................................... 294
  9.7.3 TEE_Peripheral_GetPeripherals.......................................... 295
  9.7.4 TEE_Peripheral_GetState................................................... 296
  9.7.5 TEE_Peripheral_GetStateTable........................................... 297
  9.7.6 TEE_Peripheral_Open....................................................... 298
  9.7.7 TEE_Peripheral_OpenMultiple........................................... 299
  9.7.8 TEE_Peripheral_Read....................................................... 301
  9.7.9 TEE_Peripheral_SetState.................................................. 302
  9.7.10 TEE_Peripheral_Write..................................................... 304
9.8 Event API Functions.................................................................. 305
  9.8.1 TEE_Event_AddSources..................................................... 305
  9.8.2 TEE_Event_CancelSources................................................. 306
  9.8.3 TEE_Event_CloseQueue..................................................... 307
  9.8.4 TEE_Event_DropSources.................................................. 308
  9.8.5 TEE_Event_ListSources.................................................... 309
  9.8.6 TEE_Event_OpenQueue..................................................... 310
  9.8.7 TEE_Event_TimerCreate.................................................. 312
  9.8.8 TEE_Event_Wait.............................................................. 313
Annex A Panicked Function Identification ....................................... 315
Annex B Deprecated Functions, Identifiers, Properties, and Values........ 321
  B.1 Deprecated Functions ....................................................... 321
    B.1.1 TEE_GetObjectInfo – Deprecated.................................. 321
    B.1.2 TEE_RestrictObjectUsage – Deprecated.......................... 322
    B.1.3 TEE_CopyObjectAttributes – Deprecated........................ 324
    B.1.4 TEE_CloseAndDeletePersistentObject – Deprecated........ 325
    B.1.5 TEE_BigIntInitFMMContext - deprecated.......................... 325
  B.2 Deprecated Identifiers ...................................................... 327
  B.3 Deprecated Properties ..................................................... 329
Annex C Normative References for Algorithms ................................. 330
Annex D Peripheral API Usage (Informative)..................................... 334
Figures

Figure 2-1: Trusted Application Interactions with the Trusted OS ........................................................................................ 26
Figure 7-1: Persistent Time Status State Machine ............................................................................................................... 229
Figure 9-1: Example of Multiple Access to Bus-oriented Peripheral (Informative) ............................................................. 273
Figure 9-2: Peripheral API Overview .................................................................................................................................... 275
Figure 9-3: Event API Overview ............................................................................................................................................. 276
Tables

Table 1-1: Normative References .................................................................................................................... 15
Table 1-2: Informative References .................................................................................................................. 16
Table 1-3: Terminology and Definitions ........................................................................................................... 17
Table 1-4: Abbreviations .................................................................................................................................. 20
Table 1-5: Revision History ............................................................................................................................. 22
Table 2-1: Handle Types ................................................................................................................................. 34
Table 3-1: UUID Usage Reservations ............................................................................................................. 40
Table 3-2: Return Code Formats and Ranges ................................................................................................ 41
Table 3-3: API Return Codes .......................................................................................................................... 42
Table 4-1: Parameter Type Constants ............................................................................................................ 52
Table 4-2: Login Type Constants .................................................................................................................... 52
Table 4-3: Origin Code Constants ................................................................................................................... 53
Table 4-4: Property Set Pseudo-Handle Constants ........................................................................................ 53
Table 4-5: Memory Access Rights Constants ................................................................................................ 53
Table 4-6: TA Interface Functions ................................................................................................................... 55
Table 4-7: Effect of Client Operation on TA Interface ..................................................................................... 56
Table 4-8: Content of params[i] when Trusted Application Entry Point Is Called......................................... 64
Table 4-9: Interpretation of params[i] when Trusted Application Entry Point Returns .................................. 65
Table 4-10: Property Sets ................................................................................................................................ 67
Table 4-11: Trusted Application Standard Configuration Properties ............................................................... 81
Table 4-12: Standard Client Properties ........................................................................................................... 84
Table 4-13: Client Identities ............................................................................................................................. 84
Table 4-14: Implementation Properties ........................................................................................................... 86
Table 4-14b: Specification Version Number Property – 32-bit Integer Structure ............................................ 92
Table 4-15: Interpretation of params[i] on Entry to Internal Client API ............................................................. 99
Table 4-16: Effects of Internal Client API on params[i] .................................................................................. 99
Table 4-17: Valid Hint Values ......................................................................................................................... 109
Table 5-1: Values of gpd.tee.trustedStorage.rollbackDetection.protectionLevel ........................................ 119
Table 5-1b: TEE_Whence Constants .............................................................................................................. 121
Table 5-2: Object Storage Constants ............................................................................................................. 122
Table 5-3: Data Flag Constants ....................................................................................................................... 122
Table 5-4: Usage Constants ............................................................................................................................ 122
Table 5-4b: Miscellaneous Constants [formerly Table 5-8] ............................................................................ 123
Table 5-5: Handle Flag Constants ................................................................................................................... 123
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-6</td>
<td>Operation Constants</td>
</tr>
<tr>
<td>5-7</td>
<td>Operation States</td>
</tr>
<tr>
<td>5-8</td>
<td>[moved – now Table 5-4b]</td>
</tr>
<tr>
<td>5-9</td>
<td>TEE_AllocateTransientObject Object Types and Key Sizes</td>
</tr>
<tr>
<td>5-10</td>
<td>TEE_PopulateTransientObject Supported Attributes</td>
</tr>
<tr>
<td>5-11</td>
<td>TEE_CopyObjectAttributes1 Parameter Types</td>
</tr>
<tr>
<td>5-12</td>
<td>TEE_GenerateKey Parameters</td>
</tr>
<tr>
<td>5-13</td>
<td>Effect of TEE_DATA_FLAG_OVERWRITE on Behavior of TEE_CreatePersistentObject</td>
</tr>
<tr>
<td>5-14</td>
<td>Examples of TEE_OpenPersistentObject Sharing Rules</td>
</tr>
<tr>
<td>6-1</td>
<td>Supported Cryptographic Algorithms</td>
</tr>
<tr>
<td>6-2</td>
<td>Optional Cryptographic Algorithms</td>
</tr>
<tr>
<td>6-3</td>
<td>Possible TEE_OperationMode Values</td>
</tr>
<tr>
<td>6-4</td>
<td>TEE_AllocateOperation Allowed Modes</td>
</tr>
<tr>
<td>6-5</td>
<td>Public Key Allowed Modes</td>
</tr>
<tr>
<td>6-6</td>
<td>Key-Pair Parts for Operation Modes</td>
</tr>
<tr>
<td>6-6b</td>
<td>Symmetric Encrypt/Decrypt Operation Parameters</td>
</tr>
<tr>
<td>6-7</td>
<td>Asymmetric Encrypt/Decrypt Operation Parameters</td>
</tr>
<tr>
<td>6-8</td>
<td>Asymmetric Sign Operation Parameters</td>
</tr>
<tr>
<td>6-9</td>
<td>Asymmetric Verify Operation Parameters</td>
</tr>
<tr>
<td>6-10</td>
<td>Asymmetric Derivation Operation Parameters</td>
</tr>
<tr>
<td>6-11</td>
<td>List of Algorithm Identifiers</td>
</tr>
<tr>
<td>6-12</td>
<td>Structure of Algorithm Identifier or Object Type Identifier</td>
</tr>
<tr>
<td>6-12b</td>
<td>Algorithm Subtype Identifier</td>
</tr>
<tr>
<td>6-13</td>
<td>List of Object Types</td>
</tr>
<tr>
<td>6-14</td>
<td>List of Supported Cryptographic Elements</td>
</tr>
<tr>
<td>6-15</td>
<td>Object or Operation Attributes</td>
</tr>
<tr>
<td>6-16</td>
<td>Attribute Format Definitions</td>
</tr>
<tr>
<td>6-17</td>
<td>Partial Structure of Attribute Identifier</td>
</tr>
<tr>
<td>6-18</td>
<td>Attribute Identifier Flags</td>
</tr>
<tr>
<td>7-1</td>
<td>Values of the gpd.tee.systemTime.protectionLevel Property</td>
</tr>
<tr>
<td>7-2</td>
<td>Values of the gpd.tee.TAPersistentTime.protectionLevel Property</td>
</tr>
<tr>
<td>9-1</td>
<td>Maximum Sizes of Structure Payloads</td>
</tr>
<tr>
<td>9-2</td>
<td>TEE_EVENT_TYPE Values</td>
</tr>
<tr>
<td>9-3</td>
<td>TEE_PERIPHERAL_TYPE Values</td>
</tr>
<tr>
<td>9-4</td>
<td>TEE_PERIPHERAL_FLAGS Values</td>
</tr>
<tr>
<td>9-5</td>
<td>TEE_PeripheralStateId Values</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>9-6</td>
<td>TEE_PERIPHERAL_STATE_NAME Values</td>
</tr>
<tr>
<td>9-7</td>
<td>TEE_PERIPHERAL_STATE_FW_INFO Values</td>
</tr>
<tr>
<td>9-8</td>
<td>TEE_PERIPHERAL_STATE_MANUFACTURER Values</td>
</tr>
<tr>
<td>9-9</td>
<td>TEE_PERIPHERAL_STATE_FLAGS Values</td>
</tr>
<tr>
<td>9-10</td>
<td>TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS Values</td>
</tr>
<tr>
<td>9-11</td>
<td>TEE_PERIPHERAL_OS State Table Values</td>
</tr>
<tr>
<td>9-12</td>
<td>TEE_PERIPHERAL_SESSION State Table Values</td>
</tr>
<tr>
<td>9-13</td>
<td>TEE_PeripheralValueType Values</td>
</tr>
<tr>
<td>A-1</td>
<td>Function Identification Values</td>
</tr>
<tr>
<td>B-1</td>
<td>Deprecated Object Identifier</td>
</tr>
<tr>
<td>B-2</td>
<td>Deprecated Algorithm Identifiers</td>
</tr>
<tr>
<td>B-3</td>
<td>Deprecated Properties</td>
</tr>
<tr>
<td>C-1</td>
<td>Normative References for Algorithms</td>
</tr>
</tbody>
</table>
1 Introduction

This specification defines a set of C APIs for the development of Trusted Applications (TAs) running inside a Trusted Execution Environment (TEE). For the purposes of this document a TEE is expected to meet the requirements defined in the GlobalPlatform TEE System Architecture ([Sys Arch]) specification, i.e. it is accessible from a Rich Execution Environment (REE) through the GlobalPlatform TEE Client API (described in the GlobalPlatform TEE Client API Specification [Client API]) but is specifically protected against malicious attacks and only runs code trusted in integrity and authenticity.

The APIs defined in this document target the C language and provide the following set of functionalities to TA developers:

- Basic OS-like functionalities, such as memory management, timer, and access to configuration properties
- Communication means with Client Applications (CAs) running in the Rich Execution Environment
- Trusted Storage facilities
- Cryptographic facilities
- Time management facilities

The scope of this document is the development of Trusted Applications in the C language and their interactions with the TEE Client API. It does not cover other possible language bindings or the run-time installation and management of Trusted Applications.

1.1 Audience

This document is suitable for software developers implementing Trusted Applications running inside the TEE which need to expose an externally visible interface to Client Applications and to use resources made available through the TEE Internal Core API, such as cryptographic capabilities and Trusted Storage.

This document is also intended for implementers of the TEE itself, its Trusted OS, Trusted Core Framework, the TEE APIs, and the communications infrastructure required to access Trusted Applications.

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit [https://www.globalplatform.org/specificationsipdisclaimers.asp](https://www.globalplatform.org/specificationsipdisclaimers.asp). GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.
### 1.3 References

See also Annex C: Normative References for Algorithms.

<table>
<thead>
<tr>
<th>Standard / Specification</th>
<th>Description</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPD_SPE_007</td>
<td>GlobalPlatform Technology TEE Client API Specification</td>
<td>[Client API]</td>
</tr>
<tr>
<td>GPD_SPE_009</td>
<td>GlobalPlatform Technology TEE System Architecture</td>
<td>[Sys Arch]</td>
</tr>
<tr>
<td>GPD_SPE_025</td>
<td>GlobalPlatform Technology TEE TA Debug Specification</td>
<td>[TEE TA Debug]</td>
</tr>
<tr>
<td>GPD_SPE_120</td>
<td>GlobalPlatform Technology TEE Management Framework</td>
<td>[TEE Mgmt Fmwk]</td>
</tr>
<tr>
<td>GPD_SPE_042</td>
<td>GlobalPlatform Technology TEE TUI Extension: Biometrics API</td>
<td>[TEE TUI Bio]</td>
</tr>
<tr>
<td>GPD_SPE_055</td>
<td>GlobalPlatform Technology TEE Trusted User Interface Low-level API</td>
<td>[TEE TUI Low]</td>
</tr>
<tr>
<td>GPD_SPE_021</td>
<td>GlobalPlatform Technology TEE Protection Profile</td>
<td>[TEE PP]</td>
</tr>
<tr>
<td>ISO/IEC 9899:1999</td>
<td>Programming languages – C</td>
<td>[C99]</td>
</tr>
<tr>
<td>NIST Recommended Elliptic Curves</td>
<td>Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography</td>
<td>[NIST Re Cur]</td>
</tr>
<tr>
<td>NIST SP800-56B</td>
<td>Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography</td>
<td>[NIST SP800-56B]</td>
</tr>
<tr>
<td>RFC 2045</td>
<td>Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies</td>
<td>[RFC 2045]</td>
</tr>
<tr>
<td>RFC 2119</td>
<td>Key words for use in RFCs to Indicate Requirement Levels</td>
<td>[RFC 2119]</td>
</tr>
<tr>
<td>RFC 4122</td>
<td>A Universally Unique IDentifier (UUID) URN Namespace</td>
<td>[RFC 4122]</td>
</tr>
<tr>
<td>RFC 7748</td>
<td>Elliptic Curves for Security</td>
<td>[X25519]</td>
</tr>
<tr>
<td>RFC 8032</td>
<td>Edwards-Curve Digital Signature Algorithm</td>
<td>[Ed25519]</td>
</tr>
<tr>
<td>SM2</td>
<td>Organization of State Commercial Administration of China, &quot;Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves&quot;, December 2010</td>
<td>[SM2]</td>
</tr>
</tbody>
</table>
### Table 1-2: Informative References

<table>
<thead>
<tr>
<th>Standard / Specification</th>
<th>Description</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP_GUI_001</td>
<td>GlobalPlatform Document Management Guide</td>
<td>[Doc Mgmt]</td>
</tr>
<tr>
<td>ISO/IEC 14888-3</td>
<td>Information technology – Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms (English Language reference for SM2)</td>
<td>[ISO 14888-3]</td>
</tr>
<tr>
<td>ISO/IEC 18033-3</td>
<td>Information technology – Security techniques – Encryption algorithms – Part 3: Block ciphers (English Language reference for SM4)</td>
<td>[ISO 18033-3]</td>
</tr>
</tbody>
</table>

### 1.4 Terminology and Definitions

The following meanings apply to SHALL, SHALL NOT, MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY in this document (refer to [RFC 2119]):

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation Flag</td>
<td>An indicator that a Client has requested cancellation of an operation.</td>
</tr>
<tr>
<td>Client</td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td>• a Client Application using the TEE Client API</td>
</tr>
<tr>
<td></td>
<td>• a Trusted Application acting as a client of another Trusted Application,</td>
</tr>
<tr>
<td></td>
<td>using the Internal Client API</td>
</tr>
<tr>
<td>Client Application (CA)</td>
<td>An application running outside of the Trusted Execution Environment making</td>
</tr>
<tr>
<td></td>
<td>use of the TEE Client API to access facilities provided by Trusted</td>
</tr>
<tr>
<td></td>
<td>Applications inside the Trusted Execution Environment. Contrast Trusted</td>
</tr>
<tr>
<td></td>
<td>Application (TA).</td>
</tr>
<tr>
<td>Client Properties</td>
<td>A set of properties associated with the Client of a Trusted Application.</td>
</tr>
<tr>
<td>Command</td>
<td>A message (including a Command Identifier and four Operation Parameters)</td>
</tr>
<tr>
<td></td>
<td>send by a Client to a Trusted Application to initiate an operation.</td>
</tr>
<tr>
<td>Command Identifier</td>
<td>A 32-bit integer identifying a Command.</td>
</tr>
<tr>
<td>Cryptographic Key Object</td>
<td>An object containing key material.</td>
</tr>
<tr>
<td>Cryptographic Key-Pair Object</td>
<td>An object containing material associated with both keys of a key-pair.</td>
</tr>
<tr>
<td>Cryptographic Operation</td>
<td>An opaque reference that identifies a particular cryptographic operation.</td>
</tr>
<tr>
<td>Handle</td>
<td></td>
</tr>
<tr>
<td>Cryptographic Operation Key</td>
<td>The key to be used for a particular operation.</td>
</tr>
<tr>
<td>Data Object</td>
<td>An object containing a data stream but no key material.</td>
</tr>
<tr>
<td>Data Stream</td>
<td>Data associated with a persistent object (excluding Object Attributes and</td>
</tr>
<tr>
<td></td>
<td>metadata).</td>
</tr>
<tr>
<td>Event API</td>
<td>An API that supports the event loop. Includes the following functions,</td>
</tr>
<tr>
<td></td>
<td>among others:</td>
</tr>
<tr>
<td></td>
<td>• TEE_Event_AddSources</td>
</tr>
<tr>
<td></td>
<td>• TEE_Event_OpenQueue</td>
</tr>
<tr>
<td></td>
<td>• TEE_Event_Wait</td>
</tr>
<tr>
<td>Event loop</td>
<td>A mechanism by which a TA can enquire for and then process messages from</td>
</tr>
<tr>
<td></td>
<td>types of peripherals including pseudo-peripherals.</td>
</tr>
<tr>
<td>Function Number</td>
<td>Identifies a function within a specification. With the Specification Number,</td>
</tr>
<tr>
<td></td>
<td>forms a unique identifier for a function. May be displayed when a panic</td>
</tr>
<tr>
<td></td>
<td>occurs or in debug messages where supported.</td>
</tr>
<tr>
<td>Implementation</td>
<td>A particular implementation of the Trusted OS.</td>
</tr>
<tr>
<td>Initialized</td>
<td>Describes a transient object whose attributes have been populated.</td>
</tr>
<tr>
<td>Instance</td>
<td>A particular execution of a Trusted Application, having physical memory</td>
</tr>
<tr>
<td></td>
<td>space that is separated from the physical memory space of all other TA</td>
</tr>
<tr>
<td></td>
<td>instances.</td>
</tr>
<tr>
<td>Key Size</td>
<td>The key size associated with a Cryptographic Object; values are limited</td>
</tr>
<tr>
<td></td>
<td>by the key algorithm used.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Key Usage Flags</td>
<td>Indicators of the operations permitted with a Cryptographic Object.</td>
</tr>
<tr>
<td>Memory Reference Parameter</td>
<td>An Operation Parameter that carries a pointer to a client-owned memory buffer. Contrast Value Parameter.</td>
</tr>
<tr>
<td>Metadata</td>
<td>Additional data associated with a Cryptographic Object: Key Size and Key Usage Flags.</td>
</tr>
<tr>
<td>Multi Instance Trusted Application</td>
<td>Denotes a Trusted Application for which each session opened by a client is directed to a separate TA instance.</td>
</tr>
<tr>
<td>Object Attribute</td>
<td>Small amounts of data used to store key material in a structured way.</td>
</tr>
<tr>
<td>Object Handle</td>
<td>An opaque reference that identifies a particular object.</td>
</tr>
<tr>
<td>Object Identifier</td>
<td>A variable-length binary buffer identifying a persistent object.</td>
</tr>
<tr>
<td>Operation Parameter</td>
<td>One of four data items passed in a Command, which can contain integer values or references to client-owned shared memory blocks.</td>
</tr>
<tr>
<td>Panic</td>
<td>An exception that kills a whole TA instance. See section 2.3.3 for full definition.</td>
</tr>
<tr>
<td>Parameter Annotation</td>
<td>Denotes the pattern of usage of a function parameter or pair of function parameters.</td>
</tr>
<tr>
<td>Peripheral API</td>
<td>A low-level API that enables a Trusted Application to interact with peripherals via the Trusted OS. Includes the following functions, among others: TEE_Peripheral_GetPeripherals TEE_Peripheral_GetStateTable TEE_Peripheral_Open The Peripheral API was initially defined in [TEE TUI Low].</td>
</tr>
<tr>
<td>Persistent Object</td>
<td>An object identified by an Object Identifier and including a Data Stream. Contrast Transient Object.</td>
</tr>
<tr>
<td>Property</td>
<td>An immutable value identified by a name.</td>
</tr>
<tr>
<td>Property Set</td>
<td>Any of the following:</td>
</tr>
<tr>
<td></td>
<td>- The configuration properties of a Trusted Application</td>
</tr>
<tr>
<td></td>
<td>- Properties associated with a Client Application by the Rich Execution Environment</td>
</tr>
<tr>
<td></td>
<td>- Properties describing characteristics of a Trusted OS and/or TEE Implementation</td>
</tr>
<tr>
<td>REE Time</td>
<td>A time value that is as trusted as the REE.</td>
</tr>
<tr>
<td>Rich Execution Environment (REE)</td>
<td>An environment that is provided and governed by a Rich OS, potentially in conjunction with other supporting operating systems and hypervisors; it is outside of the TEE. This environment and applications running on it are considered untrusted. Contrast Trusted Execution Environment (TEE).</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Rich OS</td>
<td>Typically, an OS providing a much wider variety of features than are provided by the OS running inside the TEE. It is very open in its ability to accept applications. It will have been developed with functionality and performance as key goals, rather than security. Due to its size and needs, the Rich OS will run in an execution environment outside of the TEE hardware (often called an REE – Rich Execution Environment) with much lower physical security boundaries. From the TEE viewpoint, everything in the REE is considered untrusted, though from the Rich OS point of view there may be internal trust structures. Contrast Trusted OS.</td>
</tr>
<tr>
<td>Session</td>
<td>Logically connects multiple commands invoked on a Trusted Application.</td>
</tr>
<tr>
<td>Single Instance Trusted Application</td>
<td>Denotes a Trusted Application for which all sessions opened by clients are directed to a single TA instance.</td>
</tr>
<tr>
<td>Specification Number</td>
<td>Identifies the specification within which a function is defined. May be displayed when a panic occurs or in debug messages where supported.</td>
</tr>
<tr>
<td>Storage Identifier</td>
<td>A 32-bit identifier for a Trusted Storage Space that can be accessed by a Trusted Application.</td>
</tr>
<tr>
<td>System Time</td>
<td>A time value that can be used to compute time differences and operation deadlines.</td>
</tr>
<tr>
<td>TA Persistent Time</td>
<td>A time value set by the Trusted Application that persists across platform reboots and whose level of trust can be queried.</td>
</tr>
<tr>
<td>Task</td>
<td>The entity that executes any code executed in a Trusted Application.</td>
</tr>
<tr>
<td>TEE Implementation</td>
<td>A specific embodiment of a TEE – i.e. a Trusted OS executing on a particular hardware platform.</td>
</tr>
<tr>
<td>Transient Object</td>
<td>An object containing attributes but no data stream, which is reclaimed when closed or when the TA instance is destroyed.</td>
</tr>
<tr>
<td></td>
<td>Contrast Persistent Object.</td>
</tr>
<tr>
<td>Trusted Application (TA)</td>
<td>An application running inside the Trusted Execution Environment that provides security related functionality to Client Applications outside of the TEE or to other Trusted Applications inside the Trusted Execution Environment.</td>
</tr>
<tr>
<td></td>
<td>Contrast Client Application (CA).</td>
</tr>
<tr>
<td>Trusted Application Configuration Properties</td>
<td>A set of properties associated with the installation of a Trusted Application.</td>
</tr>
<tr>
<td>Trusted Core Framework or “Framework”</td>
<td>The part of the Trusted OS responsible for implementing the Trusted Core Framework API(^1) that provides OS-like facilities to Trusted Applications and a way for the Trusted OS to interact with the Trusted Applications.</td>
</tr>
</tbody>
</table>

\(^1\) The Trusted Core Framework API is described in Chapter 4.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Execution Environment (TEE)</td>
<td>An execution environment that runs alongside but isolated from a REE. A TEE has security capabilities and meets certain security-related requirements: It protects TEE assets from general software attacks, defines rigid safeguards as to data and functions that a program can access, and resists a set of defined threats. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. It incorporates a Trusted OS and may include additional firmware as indicated by the gpd.tee.trustedos.* and gpd.tee.firmware.* properties. Contrast Rich Execution Environment (REE).</td>
</tr>
<tr>
<td>Trusted OS</td>
<td>An operating system running in the TEE providing the TEE Internal Core API to Trusted Applications.</td>
</tr>
<tr>
<td>Trusted Storage Spaces</td>
<td>Storage that is protected either by the hardware of the TEE or cryptographically by keys held in the TEE. Data held in such storage is either private to the Trusted Application that created it or is shared according to the rules of a Security Domain hierarchy. See [TMF].</td>
</tr>
<tr>
<td>Uninitialized</td>
<td>Describes a transient object allocated with a certain object type and maximum size but with no attributes.</td>
</tr>
<tr>
<td>Universally Unique Identifier (UUID)</td>
<td>An identifier as specified in RFC 4122 ([RFC 4122]).</td>
</tr>
<tr>
<td>Value Parameter</td>
<td>An Operation Parameter that carries two 32-bit integers. Contrast Memory Reference Parameter.</td>
</tr>
</tbody>
</table>

### 1.5 Abbreviations and Notations

Table 1-4: Abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAD</td>
<td>Additional Authenticated Data</td>
</tr>
<tr>
<td>AE</td>
<td>Authenticated Encryption</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CA</td>
<td>Client Application</td>
</tr>
<tr>
<td>CMAC</td>
<td>Cipher-based MAC</td>
</tr>
<tr>
<td>CRT</td>
<td>Chinese Remainder Theorem</td>
</tr>
<tr>
<td>CTS</td>
<td>CipherText Stealing</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DH</td>
<td>Diffie-Hellman</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>ECDH</td>
<td>Elliptic Curve Diffie-Hellman</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curve Digital Signature Algorithm</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>FMM</td>
<td>Fast Modular Multiplication</td>
</tr>
<tr>
<td>gcd</td>
<td>Greatest Common Divisor</td>
</tr>
<tr>
<td>HMAC</td>
<td>Hash-based Message Authentication Code</td>
</tr>
<tr>
<td>IEEIE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IPR</td>
<td>Intellectual Property Rights</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization Vector</td>
</tr>
<tr>
<td>LS</td>
<td>Liaison Statement</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>MD5</td>
<td>Message Digest 5</td>
</tr>
<tr>
<td>MGF</td>
<td>Mask Generating Function</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OAEP</td>
<td>Optimal Asymmetric Encryption Padding</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PKCS</td>
<td>Public Key Cryptography Standards</td>
</tr>
<tr>
<td>PSS</td>
<td>Probabilistic Signature Scheme</td>
</tr>
<tr>
<td>REE</td>
<td>Rich Execution Environment</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comments; may denote a memorandum published by the IETF</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir, Adleman asymmetric algorithm</td>
</tr>
<tr>
<td>SDO</td>
<td>Standards Defining Organization</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>TA</td>
<td>Trusted Application</td>
</tr>
<tr>
<td>TEE</td>
<td>Trusted Execution Environment</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>UTF</td>
<td>Unicode Transformation Format</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally Unique Identifier</td>
</tr>
<tr>
<td>XTS</td>
<td>XEX-based Tweaked Codebook mode with ciphertext stealing (CTS)</td>
</tr>
</tbody>
</table>
1.6 Revision History

GlobalPlatform technical documents numbered n.0 are major releases. Those numbered n.1, n.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered n.n.1, n.n.2, etc., are maintenance releases that incorporate errata and precisions; all non-trivial changes are indicated, often with revision marks.

Table 1-5: Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2011</td>
<td>1.0</td>
<td>Initial Public Release, as “TEE Internal API Specification”.</td>
</tr>
<tr>
<td>June 2014</td>
<td>1.1</td>
<td>Public Release, as “TEE Internal Core API Specification”.</td>
</tr>
<tr>
<td>June 2016</td>
<td>1.1.1</td>
<td>Public Release, showing all non-trivial changes since v1.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Significant changes include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Many parameters were defined as size_t in v1.0 then changed to uint32_t in v1.1, and have now been reverted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Improved clarity of specification with regard to TEE_GenerateKey parameter checking. Reverted over-prescriptive requirements for parameter vetting, re-enabling practical prime checking.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarification of invalid storage ID handling with regard to TEE_CreatePersistentObject and TEE_OpenPersistentObject.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified which algorithms may use an IV.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the availability of TEE_GetPropertyAsBinaryBlock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified mismatches between Table 6-12 and elsewhere.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Deprecated incorrectly defined algorithm identifiers and defined a distinct set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Corrected an error in TEE_BigIntComputeExtendedGcd range validation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified operation of TEEC_OpenSession with NULL TEEC_Operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified relationship of specification with FIPS 186-2 and FIPS 186-4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified uniqueness of gpd.tee.deviceID in case of multiple TEEs on a device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Corrected details of when TEE_HANDLE_FLAG_INITIALIZED is set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the security of the location of operation parameters that the TA is acting on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the handling and validation of storage identifiers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the protection level relationships with anti-rollback, and the way anti-rollback violation is signaled to a TA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the data retention requirement for an unused “b” attribute value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the acceptable bit size for some security operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Relaxed attribute restrictions such that TEE_PopulateTransientObject and TEE_GenerateKey are aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Clarified the handling of ACCESS_WRITE_META.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| November 2016 | 1.1.2    | - New section 3.1.1 – Added #define TEE_CORE_API specific to API specification version.  
- Section 4.7 – Clarified existing gpd.tee.apiversion, and noted that it is deprecated.  
- Section 4.7 – Added more precise gpd.tee.internalCore.version.  
- New section 4.7.1 – Defined structure of integer version field structure as used in other GlobalPlatform specs. |
| August 2017 | 1.1.1.17  | Committee Review toward v1.2  
- Introduced:  
  o Curve 25519 & BSI related curves and algorithms support  
  o Chinese Algorithms  
  o Peripheral API and Event API  
  o TEE_IsAlgorithmSupported to interrogate available ECC algorithms  
  o TEE_BigIntAbs, TEE_BigIntExpMod, TEE_BigIntSetBit, TEE_BigIntSet bignum functions  
  o Memory allocation options with No Share and No Fill hints  
- Clarified principles behind the choice of Panic vs. Error  
- Improved version control allowing TA builder to potentially request an API version  
- Improved support for 32-bit or 64-bit TA operation  
- Clarified functionality:  
  o Cryptographic operation states with regard to reset  
  o Use of identical keys in TEE_SetOperationKey2  
  o State transitions in TEE_AEUpdateAAD and associated functionality |
| April 2018   | 1.1.1.44  | Member Review                                                               |
| June 2018    | 1.1.2.50  | Public Review                                                               |
| TBD          | 1.2       | Public Release                                                              |
2 Overview of the TEE Internal Core API Specification

This specification defines a set of C APIs for the development of Trusted Applications (TAs) running inside a Trusted Execution Environment (TEE). For the purposes of this document a TEE is expected to meet the requirements defined in [Sys Arch], i.e. it is accessible from a Rich Execution Environment (REE) through the GlobalPlatform TEE Client API [Client API] but is specifically protected against malicious attacks and runs only code trusted in integrity and authenticity.

A TEE provides the Trusted Applications an execution environment with defined security boundaries, a set of security enabling capabilities, and means to communicate with Client Applications running in the Rich Execution Environment. This document specifies how to use these capabilities and communication means for Trusted Applications developed using the C programming language. It does not cover how Trusted Applications are installed or managed (described in TEE Management Framework – [TEE Mgmt Fmwk]) and does not cover other language bindings.

Sections below provide an overview of the TEE Internal Core API specification.

- Section 2.1 describes Trusted Applications and their operations and interactions with other TEE components.
- Section 2.2 gives an overview of the TEE Internal Core APIs that provide core secure services to the Trusted Applications.
- Section 2.3 describes error handling, including how errors are handled by TEE internal specifications, whether detected during TA execution or in a panic situation.
- Section 2.4 describes different opaque handle types used in the specification. These opaque handles refer to objects created by the API implementation for a TA instance.
- Section 2.5 describes TEE properties that refer to configuration parameters, permissions, or implementation characteristics.
2.1 Trusted Applications

A Trusted Application (TA) is a program that runs in a Trusted Execution Environment (TEE) and exposes security services to its Clients. A Trusted Application is command-oriented. Clients access a Trusted Application by opening a session with the Trusted Application and invoking commands within the session. When a Trusted Application receives a command, it parses the messages associated with the command, performs any required processing, and then sends a response back to the client.

A Client typically runs in the Rich Execution Environment and communicates with a Trusted Application using the TEE Client API [Client API]. It is then called a "Client Application". It is also possible for a Trusted Application to act as a client of another Trusted Application, using the Internal Client API (see section 4.9). The term "Client" covers both cases.
### 2.1.1 TA Interface

Each Trusted Application exposes an interface (the TA interface) composed of a set of entry point functions that the Trusted Core Framework implementation calls to inform the TA about life cycle changes and to relay communication between Clients and the TA. Once the Trusted Core Framework has called one of the TA entry points, the TA can make use of the TEE Internal Core API to access the facilities of the Trusted OS, as illustrated in Figure 2-1. For more information on the TA interface, see section 4.3.

Each Trusted Application is identified by a **Universally Unique Identifier** (UUID) as specified in [RFC 4122]. Each Trusted Application also comes with a set of Trusted Application Configuration Properties. These properties are used to configure the Trusted OS facilities exposed to the Trusted Application. Properties can also be used by the Trusted Application itself as a means of configuration.

**Figure 2-1: Trusted Application Interactions with the Trusted OS**

![Diagram showing interactions between Trusted Application, TA Interface, Trusted OS, andHW/Peripherals]
2.1.2 Instances, Sessions, Tasks, and Commands

When a Client creates a session with a Trusted Application, it connects to an Instance of that Trusted Application. A Trusted Application instance has physical memory space which is separated from the physical memory space of all other Trusted Application instances. The Trusted Application instance memory space holds the Trusted Application instance heap and writable global and static data.

All code executed in a Trusted Application is said to be executed by Tasks. A Task keeps a record of its execution history (typically realized with a stack) and current execution state. This record is collectively called a Task context. A Task SHALL be created each time the Trusted OS calls an entry point of the Trusted Application. Once the entry point has returned, an implementation may recycle a Task to call another entry point but this SHALL appear like a completely new Task was created to call the new entry point.

A Session is used to logically connect multiple commands invoked in a Trusted Application. Each session has its own state, which typically contains the session context and the context(s) of the Task(s) executing the session.

A Command is issued within the context of a session and contains a Command Identifier, which is a 32-bit integer, and four Operation Parameters, which can contain integer values or references to client-owned shared memory blocks.

It is up to the Trusted Application implementer to define the combinations of commands and their parameters that are supported by the Trusted Application. This is outside the scope of this specification.

2.1.3 Sequential Execution of Entry Points

All entry point calls within a given Trusted Application instance are called in sequence, i.e. no more than one entry point is executed at any point in time. The Trusted Core Framework implementation SHALL guarantee that a commenced entry point call is completed before any new entry point call is allowed to begin execution.

If there is more than one entry point call to complete at any point in time, all but one call SHALL be queued by the Framework. The order in which the Framework queues and picks enqueued calls for execution is implementation-defined.

It is not possible to execute multiple concurrent commands within a session. The TEE guarantees that a pending command has completed before a new command is executed.

Since all entry points of a given Trusted Application instance are called in sequence, there is no need to use any dedicated synchronization mechanisms to maintain consistency of any Trusted Application instance memory. The sequential execution of entry points inherently guarantees this consistency.

2.1.4 Cancellations

Clients can request the cancellation of open-session and invoke-command operations at any time.

If an operation is requested to be cancelled and has not reached the Trusted Application yet but has been queued, then the operation is simply retired from the queue.

If the operation has already been transmitted to the Trusted Application, then the task running the operation is put in the cancelled state. This has an effect on a few “cancellable” functions, such as TEE_Wait, but this effect may also be masked by the Trusted Application if it does not want to be affected by client cancellations. See section 4.10 for more details on how a Trusted Application can handle cancellation requests and mask their effect.
2.1.5 Unexpected Client Termination

When the client of a Trusted Application dies or exits abruptly and when it can be properly detected, then this
shall appear to the Trusted Application as if the client requests cancellation of all pending operations and
gracefully closes all its client sessions. It shall be indistinguishable from a clean session closing.

More precisely, the REE SHOULD detect when a Client Application dies or exits. When this happens, the REE
shall initiate a termination process that shall result in the following sequence of events for all Trusted
Application instances that are serving a session with the terminating client:

- If an operation is pending in the closing session, it shall appear as if the client had requested its
cancellation.
- When no operation remains pending in the session, the session shall be closed.

If a TA client is a TA itself, this sequence of events shall happen when the client TA panics or exits due to
the termination of its own Client Application.\(^2\)

2.1.6 Instance Types

At least two Trusted Application instance types shall be supported: Multi Instance and Single Instance.
Whether a Trusted Application is Multi Instance or Single Instance is part of its configuration properties and
shall be enforced by the Trusted OS. See section 4.5 for more information on configuration properties.

- For a **Multi Instance Trusted Application**, each session opened by a client is directed to a separate
  Trusted Application instance, created on demand when the session is opened and destroyed when the
  session closes. By definition, every instance of such a Trusted Application accepts and handles one
  and only one session at a given time.
- For a **Single Instance Trusted Application**, all sessions opened by the clients are directed to a
  single Trusted Application instance. From the Trusted Application point of view, all sessions share the
  same Trusted Application instance memory space, which means for example that memory
dynamically allocated for one session is accessible in all other sessions. It is also configurable
  whether a Single Instance Trusted Application accepts multiple concurrent sessions or not.

2.1.7 Configuration, Development, and Management

Trusted Applications as discussed in this document are developed using the C language. The way Trusted
Applications are compiled and linked is implementation-dependent.

The TEE Management Framework [TEE Mgmt Fmwk] defines a mechanism by which Trusted Applications
can be configured and installed in a TEE. The scope of this specification does not include configuration,
installation, de-installation, signing, verification, or any other life cycle or deployment aspects.

\(^2\) Panics are discussed in section 2.3.3.
2.2 TEE Internal Core APIs

The TEE Internal Core APIs provide specified functionality that MUST be available on a GlobalPlatform TEE implementation alongside optional functionality that MAY be available in a GlobalPlatform TEE implementation. The Trusted OS implements TEE Internal Core APIs that are used by Trusted Applications to develop secure tasks. These APIs provide building blocks to TAs by offering them a set of core services.

A guiding principle for the TEE Internal Core APIs is that it should be possible for a TA implementer to write source code which is portable to different TEE implementations. In particular, the TEE Internal Core APIs are designed to be used portably on TEE implementations which might have very different CPU architectures running the Trusted OS.

The TEE Internal Core APIs are further classified into six broad categories described below.

2.2.1 Trusted Core Framework API

This specification defines an API that provides OS functionality – integration, scheduling, communication, memory management, and system information retrieval interfaces – and channels communications from Client Applications or other Trusted Applications to the Trusted Application.

2.2.2 Trusted Storage API for Data and Keys

This specification defines an API that defines Trusted Storage for keys or general purpose data. This API provides access to the following facilities:

- Trusted Storage for general purpose data and key material with guarantees on the confidentiality and integrity of the data stored and atomicity of the operations that modify the storage
  - The Trusted Storage may be backed by non-secure resources as long as suitable cryptographic protection is applied, which SHALL be as strong as the means used to protect the TEE code and data itself.
  - The Trusted Storage SHALL be bound to a particular device, which means that it SHALL be accessible or modifiable only by authorized TAs running in the same TEE and on the same device as when the data was created.
  - See [Sys Arch] section 2.2 for more details on the security requirements for the Trusted Storage.
- Ability to hide sensitive key material from the TA itself
- Association of data and key: Any key object can be associated with a data stream and pure data objects contain only the data stream and no key material.
- Separation of storage among different TAs:
  - Each TA has access to its own storage space that is shared among all the instances of that TA but separated from the other TAs.
2.2.3 Cryptographic Operations API

This specification defines an API that provides the following cryptographic facilities:

- Generation and derivation of keys and key-pairs
- Support for the following types of cryptographic algorithms:
  - Digests
  - Symmetric Ciphers
  - Message Authentication Codes (MAC)
  - Authenticated Encryption algorithms such as AES-CCM and AES-GCM
  - Asymmetric Encryption and Signature
  - Key Exchange algorithms
- Pre-allocation of cryptographic operations and key containers so that resources can be allocated ahead of time and reused for multiple operations and with multiple keys over time

2.2.4 Time API

This specification defines an API to access three sources of time:

- The **System Time** has an arbitrary non-persistent origin. It may use a secure dedicated hardware timer or be based on the REE timers.
- The **TA Persistent Time** is real-time and persistent but its origin is individually controlled by each TA. This allows each TA to independently synchronize its time with the external source of trusted time of its choice. The TEE itself is not required to have a defined trusted source of time.
- The **REE Time** is real-time but SHOULD NOT be more trusted than the REE and the user.

The level of trust that a Trusted Application can put in System Time and its TA Persistent Time is implementation-defined as a given Implementation may not include fully trustable hardware sources of time and hence may have to rely on untrusted real-time clocks and timers managed by the Rich Execution Environment. However, when a more trustable source of time is available, it is expected that it will be exposed to Trusted Applications through this Time API. Note that a Trusted Application can programmatically determine the level of protection of time sources by querying implementation properties `gpd.tee.systemTime.protectionLevel` and `gpd.tee.TAPersistentTime.protectionLevel`.

2.2.5 TEE Arithmetical API

The TEE Arithmetical API is a low-level API that complements the Cryptographic API when a Trusted Application needs to implement asymmetric algorithms, modes, or paddings not supported by the Cryptographic API.

The API provides arithmetical functions to work on big numbers and prime field elements. It provides operations including regular arithmetic, modular arithmetic, primality test, and fast modular multiplication that can be based on the Montgomery reduction or a similar technique.
2.2.6 Peripheral and Event API

The Peripheral and Event API is a low-level API that enables a Trusted Application to interact with peripherals via the Trusted OS.

The Peripheral and Event API offers mechanisms to:

- Discover and identify the peripherals available to a Trusted Application.
- Determine the level of trust associated with data coming to and from the peripheral.
- Configure peripherals.
- Open and close connections between the Trusted Application and peripherals.
- Interact with peripherals using polling mechanism.
- Receive input from peripherals and other event sources using an asynchronous event mechanism.

2.3 Error Handling

2.3.1 Normal Errors

The TEE Internal Core API functions usually return a return code of type `TEE_Result` to indicate errors to the caller. This is used to denote "normal" run-time errors that the TA code is expected to catch and handle, such as out-of-memory conditions or short buffers.

Routines defined in this specification SHOULD only return the return codes defined in their definition in this specification. Where return codes are defined they SHOULD only be returned with the meaning defined by this specification: Errors which are detected for which no return code has been defined SHALL cause the routine to panic.

2.3.2 Programmer Errors

There are a number of conditions in this specification that can only occur as a result of Programmer Error, i.e. they are triggered by incorrect use of the API by a Trusted Application, such as wrong parameters, wrong state, invalid pointers, etc., rather than by run-time errors such as out-of-memory conditions.

Some Programmer Errors are explicitly tagged as "Panic Reasons" and SHALL be reliably detected by an Implementation. These errors make it impossible to produce the result of the function and require that the API panic the calling TA instance, which kills the instance. If such a Panic Reason occurs, it SHALL NOT go undetected and, e.g. produce incorrect results or corrupt TA data.

However, it is accepted that some Programmer Errors cannot be realistically detected at all times and that precise behavior cannot be specified without putting too much of a burden on the implementation. In case of such a Programmer Error, an Implementation is therefore not required to gracefully handle the error or even to behave consistently, but the Implementation SHOULD still make a best effort to detect the error and panic the calling TA. In any case, a Trusted Application SHALL NOT be able to use a Programmer Error on purpose to circumvent the security boundaries enforced by an Implementation.

In general, incorrect handles—i.e. handles not returned by the API, already closed, with the wrong owner, type, or state—are definite Panic Reasons while incorrect pointers are imprecise Programmer Errors.

Any routine defined by this specification MAY generate a panic if it detects a relevant hardware failure or is passed invalid arguments that could have been detected by the programmer, even if no panics are listed for that routine.
2.3.3 Panics

The GP TA interface assumes that parameters have been validated prior to calling. While some platforms might return errors for invalid parameters, security vulnerabilities are often created by incorrect error handling. Thus, rather than returning errors, the general design of the GP interfaces invokes a Panic in the TA.

To avoid TA Panics, the TA implementer SHALL handle potential fault conditions before calling the Trusted OS. This approach reduces the likelihood of a TA implementer introducing security vulnerabilities.

A Panic is an instance-wide uncatchable exception that kills a whole TA instance.

1. A Panic SHALL be raised when the Implementation detects an avoidable Programmer Error and there is no specifically defined error code which covers the problem;
2. A Panic SHALL be raised when the Trusted Application itself requests a panic by calling the function TEE_Panic.
3. A Panic MAY be raised if the TA’s action results in detection of a fault in the TEE itself (e.g. a corrupted TEE library) which renders the called services temporarily or permanently unavailable.
4. A Trusted OS MAY raise a TA Panic under implementation-defined circumstances.

In earlier versions of this and other GlobalPlatform TEE specifications, function definitions frequently contain the “catch all” statement that a TA may Panic if an error occurs which is not one of those specified for an API which has been called by the TA.

With the introduction of the Peripheral API, and in particular the Event API it should be noted that:

- A function SHALL NOT cause a Panic if the error detected during the call is not specifically defined for or occurring within that function.
- A function SHALL NOT cause a Panic due to an error detected during an asynchronous operation.
- It is the responsibility of the Trusted OS to cause a Panic based on the criteria of a specific function/operation.
- An asynchronous operation SHALL cause a Panic in the background of any function if the Panic conditions of that asynchronous operation is met.
- In all cases, any reported specification number and function number SHALL be for the operation or function that caused the detected the Panic state and SHALL NOT be for any other operation or function that is occurring at the same time.

When a Panic occurs, the Trusted Core Framework kills the panicking TA instance and does the following:

- It discards all client entry point calls queued on the TA instance and closes all sessions opened by Clients.
- It closes all resources that the TA instance opened, including all handles and all memory, and destroys the instance. Note that multiple instances can reference a common resource, for example an object. If an instance sharing a resource is destroyed, the Framework does not destroy the shared resource immediately, but will wait until no other instances reference the resource before reclaiming it.
After a Panic, no TA function of the instance is ever called again, not even `TA_DestroyEntryPoint`.

From the client's point of view, when a Trusted Application panics, the client commands SHALL return the error `TEE_ERROR_TARGET_DEAD` with an origin value of `TEE_ORIGIN_TEE` until the session is closed. (For details about return origins, see the function `TEE_InvokeTACommand` in section 4.9.3 or the function `TEEC_InvokeCommand` in [Client API] section 4.5.9.)

When a Panic occurs, an Implementation in a non-production environment, such as in a development or pre-production state, is encouraged to issue precise diagnostic information using the mechanisms defined in the GlobalPlatform TEE TA Debug Specification ([TEE TA Debug]) or an implementation-specific alternative to help the developer understand the Programmer Error. Diagnostic information SHOULD NOT be exposed outside of a secure development environment.

The debug API defined mechanism [TEE TA Debug] passes a panic code among the information it returns. This SHALL either be the panic code passed to `TEE_Panic` or any standard or implementation-specific error code which best indicates the reason for the panic.
2.4 Opaque Handles

This specification makes use of handles that opaquely refer to objects created by the API Implementation for a particular TA instance. A handle is only valid in the context of the TA instance that creates it and SHALL always be associated with a type.

The special value `TEE_HANDLE_NULL`, which SHALL always be 0, is used to denote the absence of a handle. It is typically used when an error occurs or sometimes to trigger a special behavior in some function. For example, the function `TEE_SetOperationKey` clears the operation key if passed `TEE_HANDLE_NULL`. In general, the "close"-like functions do nothing if they are passed the `NULL` handle.

Other than the particular case of `TEE_HANDLE_NULL`, this specification does not define any constraint on the actual value of a handle.

Passing an invalid handle, i.e. a handle not returned by the API, already closed, or of the wrong type, is always a Programmer Error, except sometimes for the specific value `TEE_HANDLE_NULL`. When a handle is dereferenced by the API, the Implementation SHALL always check its validity and panic the TA instance if it is not valid.

This specification defines a C type for each high-level type of handle. The following types are defined:

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Handle Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TEE_TASessionHandle</code></td>
<td>Handle on sessions opened by a TA on another TA</td>
</tr>
<tr>
<td><code>TEE_PropSetHandle</code></td>
<td>Handle on a property set or a property enumerator</td>
</tr>
<tr>
<td><code>TEE_ObjectHandle</code></td>
<td>Handle on a cryptographic object</td>
</tr>
<tr>
<td><code>TEE_ObjectEnumHandle</code></td>
<td>Handle on a persistent object enumerator</td>
</tr>
<tr>
<td><code>TEE_OperationHandle</code></td>
<td>Handle on a cryptographic operation</td>
</tr>
</tbody>
</table>

These C types are defined as pointers on undefined structures. For example, `TEE_TASessionHandle` is defined as `struct __TEE_TASessionHandle*`. This is just a means to leverage the C language type-system to help separate different handle types. It does not mean that an Implementation has to define the structure, and handles do not need to represent addresses.
2.5 Properties

This specification makes use of Properties to represent configuration parameters, permissions, or implementation characteristics.

A property is an immutable value identified by a name, which is a Unicode string. The property value can be retrieved in a variety of formats: Unicode string, binary block, 32-bit integer, Boolean, and Identity.

Property names and values are intended to be rather small with a few hundreds of characters at most, although the specification defines no limit on the size of names or values.

In this specification, Unicode strings are always encoded in zero-terminated UTF-8, which means that a Unicode string cannot contain the U+0000 code point.

The value of a property is immutable: A Trusted Application can only retrieve it and cannot modify it. The value is set and controlled by the Implementation and SHALL be trustable by the Trusted Applications.

The following Property Sets are exposed in the API:

- Each Trusted Application can access its own configuration properties. Some of these parameters affect the behavior of the Trusted OS itself. Others can be used to configure the behavior of the TAs that this TA connects to.

- A TA instance can access a set of properties for each of its Clients. When the Client is a Trusted Application, the property set contains the configuration properties of that Trusted Application. Otherwise, it contains properties set by the Rich Execution Environment.

- Finally, a TA can access properties describing characteristics of the TEE Implementation itself.

Property names are case-sensitive and have a hierarchical structure with levels in the hierarchy separated by the dot character ".". Property names SHOULD use the reverse domain name convention to minimize the risk of collisions between properties defined by different organizations, although this cannot really be enforced by an Implementation. For example, the ACME company SHOULD use the "com.acme." prefix and properties standardized at ISO will use the "org.iso." namespace.

This specification reserves the "gpd." namespace and defines the meaning of a few properties in this namespace. Any Implementation SHALL refuse to define properties in this namespace unless they are defined in the GlobalPlatform specifications.

2.6 Peripheral Support

This specification defines support for managing peripherals. There are functions for communicating directly, in a low-level manner, with peripherals and support for an event loop which can receive events from peripherals such as touch screens and biometric authenticators.

In this specification, the Peripheral API and Event API are optional. Implementation of other GlobalPlatform specifications may make the presence of the Peripheral API and Event API mandatory. As an example, at the time of writing the GlobalPlatform TEE TUI Extension: Biometrics API ([TEE TUI Bio]) and GlobalPlatform TEE Trusted User Interface Low-level API [TEE TUI Low] specifications require support of the Peripheral and Event APIs.
3  Common Definitions

This chapter specifies the header file, common data types, constants, and parameter annotations used throughout the specification.

3.1  Header File

The header file for the TEE Internal Core API SHALL have the name “tee_internal_api.h”.

```c
#include "tee_internal_api.h"
```

3.1.1  API Version

The header file SHALL contain version specific definitions from which TA compilation options can be selected.

```c
#define TEE_CORE_API_MAJOR_VERSION ([Major version number])
#define TEE_CORE_API_MINOR_VERSION ([Minor version number])
#define TEE_CORE_API_MAINTENANCE_VERSION ([Maintenance version number])
#define TEE_CORE_API_VERSION (TEE_CORE_API_MAJOR_VERSION << 24) +
(TEE_CORE_API_MINOR_VERSION << 16) +
(TEE_CORE_API_MAINTENANCE_VERSION << 8)
```

The document version-numbering format is $X.Y[.z]$, where:

- Major Version ($X$) is a positive integer identifying the major release.
- Minor Version ($Y$) is a positive integer identifying the minor release.
- The optional Maintenance Version ($z$) is a positive integer identifying the maintenance release.

- $\text{TEE_CORE_API\_MAJOR\_VERSION}$ indicates the major version number of the TEE Internal Core API. It SHALL be set to the major version number of this specification.
- $\text{TEE_CORE_API\_MINOR\_VERSION}$ indicates the minor version number of the TEE Internal Core API. It SHALL be set to the minor version number of this specification. If the minor version is zero, then one zero shall be present.
- $\text{TEE_CORE_API\_MAINTENANCE\_VERSION}$ indicates the maintenance version number of the TEE Internal Core API. It SHALL be set to the maintenance version number of this specification. If the maintenance version is zero, then one zero shall be present.

The definitions of “Major Version”, “Minor Version”, and “Maintenance Version” in the version number of this specification are determined as defined in the GlobalPlatform Document Management Guide ([Doc Mgmt]). In particular, the value of $\text{TEE_CORE_API\_MAINTENANCE\_VERSION}$ SHALL be zero if it is not already defined as part of the version number of this document. The “Draft Revision” number SHALL NOT be provided as an API version indication.

A compound value SHALL also be defined. If the Maintenance version number is 0, the compound value SHALL be defined as:

```c
#define TEE_CORE_API_\[Major version number\]_\[Minor version number\]
```

If the Maintenance version number is not zero, the compound value SHALL be defined as:

```c
#define TEE_CORE_API_\[Major version number\]_\[Minor version number\]_\[Maintenance version number\]
```
Some examples of version definitions:

For GlobalPlatform TEE Internal Core API Specification v1.3, these would be:

```
#define TEE_CORE_API_MAJOR_VERSION (1)
#define TEE_CORE_API_MINOR_VERSION (3)
#define TEE_CORE_API_MAINTENANCE_VERSION (0)
#define TEE_CORE_API_1_3
```

And the value of TEE_CORE_API_VERSION would be 0x01030000.

For a maintenance release of the specification as v2.14.7, these would be:

```
#define TEE_CORE_API_MAJOR_VERSION (2)
#define TEE_CORE_API_MINOR_VERSION (14)
#define TEE_CORE_API_MAINTENANCE_VERSION (7)
#define TEE_CORE_API_2_14_7
```

And the value of TEE_CORE_API_VERSION would be 0x020E0700.

3.1.2 Target and Version Optimization

This specification supports definitions that TA vendors can use to specialize behavior at compile time to provide version and target-specific optimizations.

This version of the specification is designed so that it can be used in conjunction with mechanisms to:

- Provide information about the target platform and Trusted OS
- Configure the compile and link environment to the configuration best suited to a Trusted Application

The detail of these mechanisms and their output is out of the scope of this document, but it is intended that the output could be generated automatically from build system metadata and included by `tee_internal_api.h`.

The file prefix “gpd_ta_build_” is reserved for files generated by the build system, possibly derived from metadata.

The model for TA construction supported by this specification assumes that a TA will be built to comply to a specific target and set of API versions which is fixed at compile time. A Trusted OS MAY support more than one set of target and API versions at run-time by mechanisms which are outside of the scope of this specification.

3.1.3 Peripherals Support

Since: TEE Internal Core API v1.2

A Trusted OS supporting the optional Peripheral API SHALL define the following sentinel:

```
#define TEE_CORE_API_EVENT
```
3.2 Data Types

In general, comparison of values of given data types is only valid within the scope of a TA instance. Even in the same Trusted OS, other TA instances may have different endianness and word length. It is up to the TA implementer to make sure their TA to TA protocols take this into account.

3.2.1 Basic Types

This specification makes use of the integer and Boolean C types as defined in the C99 standard (ISO/IEC 9899:1999 – [C99]). In the event of any difference between the definitions in this specification and those in [C99], C99 shall prevail.

The following basic types are used:

- `size_t`: The unsigned integer type of the result of the sizeof operator.
- `intptr_t`: A signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to void* in a given TA instance, and the result will compare equal to the original pointer.
- `uintptr_t`: An unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to void* in a given TA instance, and the result will compare equal to the original pointer.
- `uint64_t`: Unsigned 64-bit integer
- `uint32_t`: Unsigned 32-bit integer
- `int64_t`: Signed 64-bit integer
- `int32_t`: Signed 32-bit integer
- `uint16_t`: Unsigned 16-bit integer
- `int16_t`: Signed 16-bit integer
- `uint8_t`: Unsigned 8-bit integer
- `int8_t`: Signed 8-bit integer
- `bool`: Boolean type with the values `true` and `false`
- `char`: Character; used to denote a byte in a zero-terminated string encoded in UTF-8

3.2.2 Bit Numbering

In this specification, bits in integers are numbered from 0 (least-significant bit) to n (most-significant bit), where n + 1 bits are used to represent the integer, e.g. for a 2048-bit TEE_BigInt, the bits would be numbered 0 to 2047 and for a 32-bit `uint32_t` they would be numbered from 0 to 31.
3.2.3 TEE_Result, TEEC_Result

Since: TEE Internal API v1.0

```c
typedef uint32_t TEE_Result;
```

`TEE_Result` is the type used for return codes from the APIs.

For compatibility with [Client API], the following alias of this type is also defined:

Since: TEE Internal API v1.0

```c
typedef TEE_Result TEEC_Result;
```
3.2.4  TEE_UUID, TEEC_UUID

Since: TEE Internal API v1.0

```c
typedef struct
{
  uint32_t timeLow;
  uint16_t timeMid;
  uint16_t timeHiAndVersion;
  uint8_t  clockSeqAndNode[8];
} TEE_UUID;
```

TEE_UUID is the Universally Unique Resource Identifier type as defined in [RFC 4122]. This type is used to identify Trusted Applications and clients.

UUIDs can be directly hard-coded in the Trusted Application code. For example, the UUID 79B77788-9789-4a7a-A2BE-B60155EEF5F3 can be hard-coded using the following code:

```c
static const TEE_UUID myUUID =
{
  0x79b77788, 0x9789, 0x4a7a,
  { 0xa2, 0xbe, 0xb6, 0x1, 0x55, 0xee, 0xf5, 0xf3 }
};
```

For compatibility with [Client API], the following alias of this type is also defined:

Note: The TEE_UUID structure is sensitive to differences in the endianness of the Client API and the TA. It is the responsibility of the Trusted OS to ensure that any endianness difference between client and TA is managed internally when those structures are passed through one of the defined APIs. The definition below assumes that the endianness of both Client API and TA are the same, and needs to be changed appropriately if this is not the case.

Since: TEE Internal API v1.0

```c
typedef TEE_UUID TEEC_UUID;
```

Universally Unique Resource Identifiers come in a number of different versions. The following reservations of usage are made:

Since: TEE Internal Core API v1.1, based on [TEE Mgmt Fmwk] v1.0

<table>
<thead>
<tr>
<th>Version</th>
<th>Reservation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUID v5</td>
<td>When the TEE Management Framework [TEE Mgmt Fmwk] is supported by a TEE, then TA and Security Domain (SD) UUIDs using version 5 SHALL conform to the extended v5 requirements found in that specification.</td>
</tr>
</tbody>
</table>
3.3 Constants

3.3.1 Return Code Ranges and Format

The format of return codes and the reserved ranges are defined in Table 3-2.

Table 3-2: Return Code Formats and Ranges

<table>
<thead>
<tr>
<th>Range</th>
<th>Value</th>
<th>Format Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>Reserved for use in GlobalPlatform</td>
<td>0x00000001 – 0x6FFFFFFF</td>
<td>The return code may identify the specification, as discussed following the</td>
</tr>
<tr>
<td>specifications, providing non-error</td>
<td></td>
<td>table.</td>
</tr>
<tr>
<td>information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for implementation-specific</td>
<td>0x70000000 – 0x7FFFFFFF</td>
<td></td>
</tr>
<tr>
<td>return code providing non-error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for implementation-specific</td>
<td>0x80000000 – 0x8FFFFFFF</td>
<td></td>
</tr>
<tr>
<td>errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for future use in GlobalPlatform</td>
<td>0x90000000 – 0xEFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>specifications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for GlobalPlatform TEE API</td>
<td>0xF0000000 – 0xFFFEFFFF</td>
<td>The return code may identify the specification, as discussed following the</td>
</tr>
<tr>
<td>defined errors</td>
<td></td>
<td>table.</td>
</tr>
<tr>
<td>Client API defined Errors (TEEC_*)</td>
<td>0xFFFF0000 – 0xFFFFFFFF</td>
<td>Note that some return codes from this and other specifications have</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incorrectly been defined in this range and are therefore grandfathered in.</td>
</tr>
</tbody>
</table>

An error code is a return code that denotes some failure: These are the return codes above 0x7FFFFFFF.

Return codes in specified ranges in Table 3-2 MAY include the specification number as a 3 digit BCD (Binary Coded Decimal) value in nibbles 7 through 5 (where the high nibble is considered nibble 8).

For example, GPD_SPE_123 may define return codes as follows:

- Specification unique non-error return codes may be numbered 0x01230000 to 0x0123FFFF.
- Specification unique error codes may be numbered 0xF1230000 to 0xF123FFFF.
### 3.3.2 Return Codes

Table 3-3 lists return codes that are used throughout the APIs.

<table>
<thead>
<tr>
<th>Constant Names and Aliases</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>TEEC_SUCCESS 0x00000000</td>
</tr>
<tr>
<td>TEE_ERROR_CORRUPT_OBJECT</td>
<td>0xF0100001</td>
</tr>
<tr>
<td>TEE_ERROR_CORRUPT_OBJECT_2</td>
<td>0xF0100002</td>
</tr>
<tr>
<td>TEE_ERROR_STORAGE_NOT_AVAILABLE</td>
<td>0xF0100003</td>
</tr>
<tr>
<td>TEE_ERROR_STORAGE_NOT_AVAILABLE_2</td>
<td>0xF0100004</td>
</tr>
<tr>
<td>TEE_ERROR_OLD_VERSION</td>
<td>0xF0100005</td>
</tr>
<tr>
<td>TEE_ERROR_GENERIC</td>
<td>TEEC_ERROR_GENERIC 0xFFFF0000</td>
</tr>
<tr>
<td>TEE_ERROR_ACCESS_DENIED</td>
<td>TEEC_ERROR_ACCESS_DENIED 0xFFFF0001</td>
</tr>
<tr>
<td>TEE_ERROR_CANCEL</td>
<td>TEEC_ERROR_CANCEL 0xFFFF0002</td>
</tr>
<tr>
<td>TEE_ERROR_ACCESS_CONFLICT</td>
<td>TEEC_ERROR_ACCESS_CONFLICT 0xFFFF0003</td>
</tr>
<tr>
<td>TEE_ERROR_EXCESS_DATA</td>
<td>TEEC_ERROR_EXCESS_DATA 0xFFFF0004</td>
</tr>
<tr>
<td>TEE_ERROR_BAD_FORMAT</td>
<td>TEEC_ERROR_BAD_FORMAT 0xFFFF0005</td>
</tr>
<tr>
<td>TEE_ERROR_BAD_PARAMETERS</td>
<td>TEEC_ERROR_BAD_PARAMETERS 0xFFFF0006</td>
</tr>
<tr>
<td>TEE_ERROR_BAD_STATE</td>
<td>TEEC_ERROR_BAD_STATE 0xFFFF0007</td>
</tr>
<tr>
<td>TEE_ERROR_ITEM_NOT_FOUND</td>
<td>TEEC_ERROR_ITEM_NOT_FOUND 0xFFFF0008</td>
</tr>
<tr>
<td>TEE_ERROR_NOT_IMPLEMENTED</td>
<td>TEEC_ERROR_NOT_IMPLEMENTED 0xFFFF0009</td>
</tr>
<tr>
<td>TEE_ERROR_NOT_SUPPORTED</td>
<td>TEEC_ERROR_NOT_SUPPORTED 0xFFFF000A</td>
</tr>
<tr>
<td>TEE_ERROR_NO_DATA</td>
<td>TEEC_ERROR_NO_DATA 0xFFFF000B</td>
</tr>
<tr>
<td>TEE_ERROR_OUT_OF_MEMORY</td>
<td>TEEC_ERROR_OUT_OF_MEMORY 0xFFFF000C</td>
</tr>
<tr>
<td>TEE_ERROR_BUSY</td>
<td>TEEC_ERROR_BUSY 0xFFFF000D</td>
</tr>
<tr>
<td>TEE_ERROR_COMMUNICATION</td>
<td>TEEC_ERROR_COMMUNICATION 0xFFFF000E</td>
</tr>
<tr>
<td>TEE_ERROR_SECURITY</td>
<td>TEEC_ERROR_SECURITY 0xFFFF000F</td>
</tr>
<tr>
<td>TEE_ERROR_SHORT_BUFFER</td>
<td>TEEC_ERROR_SHORT_BUFFER 0xFFFF0010</td>
</tr>
<tr>
<td>TEE_ERROR_EXTERNAL_CANCEL</td>
<td>TEEC_ERROR_EXTERNAL_CANCEL 0xFFFF0011</td>
</tr>
<tr>
<td>TEE_ERROR_TIMEOUT</td>
<td>0xFFFF3001</td>
</tr>
<tr>
<td>TEE_ERROR_OVERFLOW</td>
<td>0xFFFF300F</td>
</tr>
<tr>
<td>TEE_ERROR_TARGET_DEAD</td>
<td>TEEC_ERROR_TARGET_DEAD 0xFFFF3024</td>
</tr>
<tr>
<td>TEE_ERROR_STORAGE_NO_SPACE</td>
<td>0xFFFF3041</td>
</tr>
<tr>
<td>TEE_ERROR_MAC_INVALID</td>
<td>0xFFFF3071</td>
</tr>
<tr>
<td>TEE_ERROR_SIGNATURE_INVALID</td>
<td>0xFFFF3072</td>
</tr>
<tr>
<td>TEE_ERROR_TIME_NOT_SET</td>
<td>0xFFFF5000</td>
</tr>
<tr>
<td>Constant Names and Aliases</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>TEE_ERROR_TIME_NEEDS_RESET</td>
<td>0xFFFF5001</td>
</tr>
</tbody>
</table>
3.4 Parameter Annotations

This specification uses a set of patterns on the function parameters. Instead of repeating this pattern again on each occurrence, these patterns are referred to with Parameter Annotations. It is expected that this will also help with systematically translating the APIs into languages other than the C language.

The following sub-sections list all the parameter annotations used in the specification.

Note that these annotations cannot be expressed in the C language. However, the [in], [inbuf], [instring], [instringopt], and [ctx] annotations can make use of the const C keyword. This keyword is omitted in the specification of the functions to avoid mixing the formal annotations and a less expressive C keyword. However, the C header file of a compliant Implementation SHOULD use the const keyword when these annotations appear.

3.4.1 [in], [out], and [inout]

The annotation [in] applies to a parameter that has a pointer type on a structure, a base type, or more generally a buffer of a size known in the context of the API call. If the size needs to be clarified, the syntax [in(size)] is used.

When the [in] annotation is present on a parameter, it means that the API Implementation uses the pointer only for reading and does not accept shared memory.

When a Trusted Application calls an API function that defines a parameter annotated with [in], the parameter SHALL be entirely readable by the Trusted Application and SHALL be entirely owned by the calling Trusted Application instance, as defined in section 4.11.1. In particular, this means that the parameter SHALL NOT reside in a block of shared memory owned by a client of the Trusted Application. The Implementation SHALL check these conditions and if they are not satisfied, the API call SHALL panic the calling Trusted Application instance.

The annotations [out] and [inout] are equivalent to [in] except that they indicate write access and read-and-write access respectively.

Note that, as described in section 4.11.1, the NULL pointer SHALL never be accessible to a Trusted Application. This means that a Trusted Application SHALL NOT pass the NULL pointer in an [in] parameter, except perhaps if the buffer size is zero.

See the function TEE_CheckMemoryAccessRights in section 4.11.1 for more details about shared memory and the NULL pointer. See the function TEE_Panic in section 4.8.1 for information about Panics.

3.4.2 [outopt]

The [outopt] annotation is equivalent to [out] except that the caller can set the parameter to NULL, in which case the result SHALL be discarded.
3.4.3 [inbuf]

The [inbuf] annotation applies to a pair of parameters, the first of which is of pointer type, such as a `void*`, and the second of which is of type `size_t`. It means that the parameters describe an input data buffer. The entire buffer SHALL be readable by the Trusted Application and there is no restriction on the owner of the buffer: It can reside in shared memory or in private memory.

The Implementation SHALL check that the buffer is entirely readable and SHALL panic the calling Trusted Application instance if that is not the case.

Because the NULL pointer is never readable, a Trusted Application cannot pass NULL in the first (pointer) parameter unless the second (size_t) parameter is set to 0.

3.4.4 [outbuf]

The [outbuf] annotation applies to a pair of parameters, the first of which is of pointer type, such as a `void*`, and the second of which is of type `size_t*`, herein referenced with the names buffer and size. It is used by API functions to return an output data buffer. The data buffer SHALL be allocated by the calling Trusted Application and passed in the buffer parameter. Because the size of the output buffer cannot generally be determined in advance, the following convention is used:

- On entry, *size contains the number of bytes actually allocated in buffer. The buffer with this number of bytes SHALL be entirely writable by the Trusted Application, otherwise the Implementation SHALL panic the calling Trusted Application instance. In any case, the implementation SHALL NOT write beyond this limit.
- On return:
  - If the output fits in the output buffer, then the Implementation SHALL write the output in buffer and SHALL update *size with the actual size of the output in bytes.
  - If the output does not fit in the output buffer, then the implementation SHALL update *size with the required number of bytes and SHALL return TEE_ERROR_SHORT_BUFFER. It is implementation-dependent whether the output buffer is left untouched or contains part of the output. In any case, the TA SHOULD consider that its content is undefined after the function returns.

When the function returns TEE_ERROR_SHORT_BUFFER, it SHALL NOT have performed the actual requested operation. It SHALL just return the size of the output data.

Note that if the caller sets *size to 0, the function will always return TEE_ERROR_SHORT_BUFFER unless the actual output data is empty. In this case, the parameter buffer can take any value, e.g. NULL, as it will not be accessed by the Implementation. If *size is set to a non-zero value on entry, then buffer cannot be NULL because the buffer starting from the NULL address is never writable.

There is no restriction on the owner of the buffer: It can reside in shared memory or in private memory.

The parameter size SHALL be considered as [inout]. That is, size SHALL be readable and writable by the Trusted Application. The parameter size SHALL NOT be NULL and SHALL NOT reside in shared memory. The Implementation SHALL check these conditions and panic the calling Trusted Application instance if they are not satisfied.
3.4.5  [outbufopt]

The [outbufopt] annotation is equivalent to [outbuf] but if the parameter size is set to NULL, then the function SHALL behave as if the output buffer was not large enough to hold the entire output data and the output data SHALL be discarded. In this case, the parameter buffer is ignored, but SHOULD normally be set to NULL, too.

Note the difference between passing a size pointer set to NULL and passing a size that points to 0.

Assuming the function does not fail for any other reasons:

- If size is set to NULL, the function performs the operation, returns TEE_SUCCESS, and the output data is discarded.
- If size points to 0, the function does not perform the operation. It just updates *size with the output size and returns TEE_ERROR_SHORT_BUFFER.

3.4.6  [instring] and [instringopt]

The [instring] annotation applies to a single [in] parameter, which SHALL contain a zero-terminated string of char characters. Because the buffer is [in], it cannot reside in shared memory.

The [instringopt] annotation is equivalent to [instring] but the parameter can be set to NULL to denote the absence of a string.

3.4.7  [outstring] and [outstringopt]

The [outstring] annotation is equivalent to [outbuf], but the output data is specifically a zero-terminated string of char characters. The size of the buffer SHALL account for the zero terminator. The buffer may reside in shared memory.

The [outstringopt] annotation is equivalent to [outstring] but with [outbufopt] instead of [outbuf]. which means that size can be set to NULL to discard the output.

3.4.8  [ctx]

The [ctx] annotation applies to a void* parameter. It means that the parameter is not accessed by the implementation, but will merely be stored to be provided to the Trusted Application later. Although a Trusted Application typically uses such parameters to store pointers to allocated structures, they can contain any value.
3.5 Backward Compatibility

It is an explicit principle of the design of the TEE Internal Core API that backward compatibility is supported between specification versions with the same major version number. It is, in addition, a principle of the design of this specification that the API should not depend on details of the implementation platform.

There are cases where previous versions of the TEE Internal Core API contain API definitions which depend on memory accesses being expressible using 32-bit representations for pointers and buffer sizes. In TEE Internal Core API v1.2 and later we resolve this issue in a way which is backward compatible with idiomatic C99 code, but which may cause issues with code which has been written making explicit assumptions about C language type coercions to 32-bit integers.

From TEE Internal Core API v1.2 onward, definitions are available which allow a TA or its build environment to define the API version it requires. A Trusted OS or the corresponding TA build system can use these to select how TEE Internal Core API features are presented to the TA.

3.5.1 Version Compatibility Definitions

A TA can set the definitions in this section to non-zero values if it was written in a way that requires strict compatibility with a specific version of this specification. These definitions could, for example, be set in the TA source code, or they could be set by the build system provided by the Trusted OS, based on metadata that is out of scope of this specification.

This mechanism can be used where a TA depends for correct operation on the older definition. TA authors are warned that older versions are updated to clarify intended behavior rather than to change it, and there may be inconsistent behavior between different Trusted OS platforms where these definitions are used.

This mechanism resolves all necessary version information when a TA is compiled to run on a given Trusted OS.

Since: TEE Internal Core API v1.2

#define TEE_CORE_API_REQUIRED_MAJOR_VERSION (major)
#define TEE_CORE_API_REQUIRED_MINOR_VERSION (minor)
#define TEE_CORE_API_REQUIRED_MAINTENANCE_VERSION (maintenance)

The following rules govern the use of TEE_CORE_API_REQUIRED_MAJOR_VERSION, TEE_CORE_API_REQUIRED_MINOR_VERSION, and TEE_CORE_API_REQUIRED_MAINTENANCE_VERSION by TA implementers:

- If TEE_CORE_API_REQUIRED_MAINTENANCE_VERSION is defined by a TA, then TEE_CORE_API_REQUIRED_MAJOR_VERSION and TEE_CORE_API_REQUIRED_MINOR_VERSION SHALL also be defined by the TA.

- If TEE_CORE_API_REQUIRED_MINOR_VERSION is defined by a TA, then TEE_CORE_API_REQUIRED_MAJOR_VERSION SHALL also be defined by the TA.

If the TA violates any rule above, TA compilation SHALL stop with an error indicating the reason.

TEE_CORE_API_REQUIRED_MAJOR_VERSION is used by a TA to indicate that it requires strict compatibility with a specific major version of this specification in order to operate correctly. If this value is set to 0 or is unset, it indicates that the latest major version of this specification SHALL be used.

TEE_CORE_API_REQUIRED_MINOR_VERSION is used by a TA to indicate that it requires strict compatibility with a specific minor version of this specification in order to operate correctly. If this value is unset, it indicates that the latest minor version of this specification associated with the determined
TEE_CORE_API_REQUIRED_MAINTENANCE_VERSION is used by a TA to indicate that it requires strict compatibility with a specific major version of this specification in order to operate correctly. If this value is unset, it indicates that the latest maintenance version of this specification associated with TEE_CORE_API_REQUIRED_MAJOR_VERSION and TEE_CORE_API_REQUIRED_MINOR_VERSION SHALL be used.

If none of the definitions above is set, a Trusted OS or its build system SHALL select the most recent version of this specification that it supports, as defined in section 3.1.1.

If the Trusted OS is unable to provide an implementation matching the request from the TA, compilation of the TA against that Trusted OS or its build system SHALL fail with an error indicating that the Trusted OS is incompatible with the TA. This ensures that TAs originally developed against previous versions of this specification can be compiled with identical behavior, or will fail to compile.

If the above definitions are set, a Trusted OS SHALL behave exactly according to the definitions for the indicated version of the specification, with only the definitions in that version of the specification being exported to a TA by the trusted OS or its build system. In particular an implementation SHALL NOT enable APIs which were first defined in a later version of this specification than the version requested by the TA.

If the above definitions are set to 0 or are not set, then the Trusted OS SHALL behave according to this version of the specification.

To assist TA developers wishing to make use of backward-compatible behavior, each API in this document is marked with the version of this specification in which it was last modified. Where strict backward compatibility is not maintained, information has been provided to explain any changed behavior.

As an example, consider a TA which requires strict compatibility with TEE Internal Core API v1.1:

```c
#define TEE_CORE_API_REQUIRED_MAJOR_VERSION (1)
#define TEE_CORE_API_REQUIRED_MINOR_VERSION (1)
#define TEE_CORE_API_REQUIRED_MAINTENANCE_VERSION (0)
```

Due to the semantics of the C preprocessor, the above definitions SHALL be defined before the main body of definitions in “tee_internal_api.h” is processed. The mechanism by which this occurs is out of scope of this document.
4 Trusted Core Framework API

This chapter defines the Trusted Core Framework API, defining OS-like APIs and infrastructure. It contains the following sections:

- Section 4.1, Data Types
- Section 4.2, Constants
  Common definitions used throughout the chapter.
- Section 4.3, TA Interface
  Defines the entry points that each TA SHALL define.
- Section 4.4, Property Access Functions
  Defines the generic functions to access properties. These functions can be used to access TA Configuration Properties, Client Properties, and Implementation Properties.
- Section 4.5, Trusted Application Configuration Properties
  Defines the standard Trusted Application Configuration Properties.
- Section 4.6, Client Properties
  Defines the standard Client Properties.
- Section 4.7, Implementation Properties
  Defines the standard Implementation Properties.
- Section 4.8, Panics
  Defines the function TEE_Panic.
- Section 4.9, Internal Client API
  Defines the Internal Client API that allows a Trusted Application to act as a Client of another Trusted Application.
- Section 4.10, Cancellation Functions
  Defines how a Trusted Application can handle client cancellation requests, acknowledge them, and mask or unmask the propagated effects of cancellation requests on cancellable functions.
- Section 4.11, Memory Management Functions
  Defines how to check the access rights to memory buffers, how to access global variables, how to allocate memory (similar to malloc), and a few utility functions to fill or copy memory blocks.
4.1 Data Types

4.1.1 TEE_Identity

Since: TEE Internal API v1.0

```c
typedef struct {
    uint32_t login;
    TEE_UUID uuid;
} TEE_Identity;
```

The TEE_Identity structure defines the full identity of a Client:

- login is one of the TEE_LOGIN_XXX constants. (See section 4.2.2.)
- uuid contains the client UUID or Nil (as defined in [RFC 4122]) if not applicable.

4.1.2 TEE_Param

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
typedef union {
    struct {
        void* buffer; size_t size;
    } memref;
    struct {
        uint32_t a;
        uint32_t b;
    } value;
} TEE_Param;
```

This union describes one parameter passed by the Trusted Core Framework to the entry points TA_OpenSessionEntryPoint or TA_InvokeCommandEntryPoint or by the TA to the functions TEE_OpenTASession or TEE_InvokeTACCommand.

Which of the field value or memref to select is determined by the parameter type specified in the argument paramTypes passed to the entry point. See section 4.3.6.1 and section 4.9.4 for more details on how this type is used.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the size.

4.1.3 TEE_TASessionHandle

Since: TEE Internal API v1.0

```c
typedef struct __TEE_TASessionHandle* TEE_TASessionHandle;
```

TEE_TASessionHandle is an opaque handle on a TA Session. These handles are returned by the function TEE_OpenTASession specified in section 4.9.1.
4.1.4 TEE_PropSetHandle

Since: TEE Internal API v1.0

```c
typedef struct __TEE_PropSetHandle* TEE_PropSetHandle;
```

TEE_PropSetHandle is an opaque handle on a property set or enumerator. These handles either are returned by the function `TEE_AllocatePropertyEnumerator` specified in section 4.4.7 or are one of the pseudo-handles defined in section 4.2.4.

Since: TEE Internal Core API v1.2

TEE_PropSetHandle values use interfaces that are shared between defined constants and real opaque handles.

The Trusted OS SHALL take precautions that it will never generate a real opaque handle of type TEE_PropSetHandle using constant values defined in section 4.2.4, and that when acting upon a TEE_PropSetHandle it will, where appropriate, filter for these constant values first.
4.2 Constants

4.2.1 Parameter Types

Table 4-1: Parameter Type Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Equivalent on Client API</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PARAM_TYPE_NONE</td>
<td>TEEC_NONE</td>
<td>0</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INPUT</td>
<td>TEEC_VALUE_INPUT</td>
<td>1</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_OUTPUT</td>
<td>TEEC_VALUE_OUTPUT</td>
<td>2</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INOUT</td>
<td>TEEC_VALUE_INOUT</td>
<td>3</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INPUT</td>
<td>TEEC_MEMREF_TEMP_INPUT</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>or TEEC_MEMREF_PARTIAL_INPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_OUTPUT</td>
<td>TEEC_MEMREF_TEMP_OUTPUT</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>or TEEC_MEMREF_PARTIAL_OUTPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INOUT</td>
<td>TEEC_MEMREF_TEMP_INOUT</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>or TEEC_MEMREF_PARTIAL_INOUT</td>
<td></td>
</tr>
</tbody>
</table>

4.2.2 Login Types

Table 4-2: Login Type Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Equivalent on Client API</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_LOGIN_PUBLIC</td>
<td>TEEC_LOGIN_PUBLIC</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_LOGIN_USER</td>
<td>TEEC_LOGIN_USER</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_LOGIN_GROUP</td>
<td>TEEC_LOGIN_GROUP</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_LOGIN_APPLICATION</td>
<td>TEEC_LOGIN_APPLICATION</td>
<td>0x00000004</td>
</tr>
<tr>
<td>TEE_LOGIN_APPLICATION_USER</td>
<td>TEEC_LOGIN_APPLICATION_USER</td>
<td>0x00000005</td>
</tr>
<tr>
<td>TEE_LOGIN_APPLICATION_GROUP</td>
<td>TEEC_LOGIN_APPLICATION_GROUP</td>
<td>0x00000006</td>
</tr>
<tr>
<td>Reserved for future GlobalPlatform defined login types</td>
<td>0x00000007 – 0x7FFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Reserved for implementation-specific login types</td>
<td>0x80000000 – 0xEFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>TEE_LOGIN_TRUSTED_APP</td>
<td></td>
<td>0xF0000000</td>
</tr>
<tr>
<td>Reserved for future GlobalPlatform defined login types</td>
<td>0xF0000001 – 0xFFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>
4.2.3 Origin Codes

Table 4-3: Origin Code Constants

<table>
<thead>
<tr>
<th>Constant Names</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ORIGIN_API</td>
<td>TEEC_ORIGIN_API</td>
</tr>
<tr>
<td></td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_ORIGIN_COMMS</td>
<td>TEEC_ORIGIN_COMMS</td>
</tr>
<tr>
<td></td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_ORIGIN_TEE</td>
<td>TEEC_ORIGIN_TEE</td>
</tr>
<tr>
<td></td>
<td>0x00000003</td>
</tr>
<tr>
<td>TEE_ORIGIN_TRUSTED_APP</td>
<td>TEEC_ORIGIN_TRUSTED_APP</td>
</tr>
<tr>
<td></td>
<td>0x00000004</td>
</tr>
<tr>
<td>Reserved for future GlobalPlatform</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x00000005 – 0xEFFFFFFF</td>
</tr>
<tr>
<td>Reserved for implementation-specific origin values</td>
<td>0xF0000000 – 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

Note: Other specifications can define additional origin code constants, so TA implementers SHOULD ensure that they include default handling for other values.

4.2.4 Property Set Pseudo-Handles

Table 4-4: Property Set Pseudo-Handle Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved for use by allocated property set pseudo-</td>
<td></td>
</tr>
<tr>
<td>handles.</td>
<td>ALL 32-bit address</td>
</tr>
<tr>
<td></td>
<td>boundary aligned values</td>
</tr>
<tr>
<td></td>
<td>are reserved for use</td>
</tr>
<tr>
<td></td>
<td>as non-constant values</td>
</tr>
<tr>
<td></td>
<td>allocated by the API as</td>
</tr>
<tr>
<td></td>
<td>opaque handles. i.e.</td>
</tr>
<tr>
<td></td>
<td>any value with the</td>
</tr>
<tr>
<td></td>
<td>least significant 2</td>
</tr>
<tr>
<td></td>
<td>address bits zero</td>
</tr>
<tr>
<td>Reserved</td>
<td>Non 32-bit boundary</td>
</tr>
<tr>
<td></td>
<td>aligned values In the</td>
</tr>
<tr>
<td></td>
<td>range 0x00000000 – 0xEFFFFFFF</td>
</tr>
<tr>
<td>Reserved for implementation-specific property</td>
<td>Non 32-bit boundary</td>
</tr>
<tr>
<td>sets</td>
<td>aligned values in the</td>
</tr>
<tr>
<td></td>
<td>range: 0xF0000000 – 0xFFFEFFFF</td>
</tr>
<tr>
<td>Reserved for future GlobalPlatform use</td>
<td>Non 32-bit boundary</td>
</tr>
<tr>
<td></td>
<td>aligned values in the</td>
</tr>
<tr>
<td></td>
<td>range: 0xFFFF0000 – 0xFFFFFFFC</td>
</tr>
<tr>
<td>TEE_PROPSET_TEE_IMPLEMENTATION</td>
<td>(TEE_PropSetHandle)0xFFFFFD</td>
</tr>
<tr>
<td>TEE_PROPSET_CURRENT_CLIENT</td>
<td>(TEE_PropSetHandle)0xFFFFFE</td>
</tr>
<tr>
<td>TEE_PROPSET_CURRENT_TA</td>
<td>(TEE_PropSetHandle)0xFFFFFFF</td>
</tr>
</tbody>
</table>

4.2.5 Memory Access Rights

Table 4-5: Memory Access Rights Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_MEMORY_ACCESS_READ</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_MEMORY_ACCESS_WRITE</td>
<td>0x00000002</td>
</tr>
<tr>
<td>Constant Name</td>
<td>Constant Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>TEE_MEMORY_ACCESS_ANY_OWNER</td>
<td>0x00000004</td>
</tr>
</tbody>
</table>
4.3 TA Interface

Each Trusted Application SHALL provide the Implementation with a number of functions, collectively called the “TA interface”. These functions are the entry points called by the Trusted Core Framework to create the instance, notify the instance that a new client is connecting, notify the instance when the client invokes a command, etc. These entry points cannot be registered dynamically by the Trusted Application code: They SHALL be bound to the framework before the Trusted Application code is started.

Table 4-6 lists the functions in the TA interface.

<table>
<thead>
<tr>
<th>TA Interface Function (Entry Point)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA_CreateEntryPoint</td>
<td>This is the Trusted Application constructor. It is called once and only once in the lifetime of the Trusted Application instance. If this function fails, the instance is not created.</td>
</tr>
<tr>
<td>TA_DestroyEntryPoint</td>
<td>This is the Trusted Application destructor. The Trusted Core Framework calls this function just before the Trusted Application instance is terminated. The Framework SHALL guarantee that no sessions are open when this function is called. When TA_DestroyEntryPoint returns, the Framework SHALL collect all resources claimed by the Trusted Application instance.</td>
</tr>
<tr>
<td>TA_OpenSessionEntryPoint</td>
<td>This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened. In this function, the Trusted Application can attach an opaque void* context to the session. This context is recalled in all subsequent TA calls within the session.</td>
</tr>
<tr>
<td>TA_CloseSessionEntryPoint</td>
<td>This function is called when the client closes a session and disconnects from the Trusted Application instance. The Implementation guarantees that there are no active commands in the session being closed. The session context reference is given back to the Trusted Application by the Framework. It is the responsibility of the Trusted Application to deallocate the session context if memory has been allocated for it.</td>
</tr>
<tr>
<td>TA_InvokeCommandEntryPoint</td>
<td>This function is called whenever a client invokes a Trusted Application command. The Framework gives back the session context reference to the Trusted Application in this function call.</td>
</tr>
</tbody>
</table>
Table 4-7 summarizes client operations and the resulting Trusted Application effect.

<table>
<thead>
<tr>
<th>Client Operation</th>
<th>Trusted Application Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEEC_OpenSession or TEE_OpenTASession</td>
<td>If a new Trusted Application instance is needed to handle the session, TA_CreateEntryPoint is called. Then, TA_OpenSessionEntryPoint is called.</td>
</tr>
<tr>
<td>TEEC_InvokeCommand or TEE_InvokeTACommand</td>
<td>TA_InvokeCommandEntryPoint is called.</td>
</tr>
<tr>
<td>TEEC_CloseSession or TEE_CloseTASession</td>
<td>TA_CloseSessionEntryPoint is called. For a multi-instance TA or for a single-instance, non keep-alive TA, if the session closed was the last session on the instance, then TA_DestroyEntryPoint is called. Otherwise, the instance is kept until the TEE shuts down.</td>
</tr>
<tr>
<td>TEEC_RequestCancellation or The function TEE_OpenTASession or TEE_InvokeTACommand is cancelled.</td>
<td>See section 4.10 for details on the effect of cancellation requests.</td>
</tr>
<tr>
<td>Client terminates unexpectedly</td>
<td>From the point of view of the TA instance, the behavior SHALL be identical to the situation where the client does not terminate unexpectedly but, for all opened sessions: • requests the cancellation of all pending operations in that session, • waits for the completion of all these operations in that session, • and finally closes that session. Note that there is no way for the TA to distinguish between the client gracefully cancelling all its operations and closing all its sessions and the Implementation taking over when the client dies unexpectedly.</td>
</tr>
</tbody>
</table>

Interface Operation Parameters

When a Client opens a session on a Trusted Application or invokes a command, it can send Operation Parameters to the Trusted Application. The parameters encode the data associated with the operation. Up to four parameters can be sent in an operation. If these are insufficient, then one of the parameters may be used to carry further parameter data via a Memory Reference.

Each parameter can be individually typed by the Client as a Value Parameter, carrying two 32-bit integers, or a Memory Reference Parameter, carrying a pointer to a client-owned memory buffer. Each parameter is also tagged with a direction of data flow (input, output, or both input and output). For output Memory References, there is a built-in mechanism for the Trusted Applications to report the necessary size of the buffer in case of a too-short buffer. See section 4.3.6 for more information about the handling of parameters in the TA interface.
Note that Memory Reference Parameters typically point to memory owned by the client and shared with the Trusted Application for the duration of the operation. This is especially useful in the case of REE Clients to minimize the number of memory copies and the data footprint in case a Trusted Application needs to deal with large data buffers, for example to process a multimedia stream protected by DRM.

Security Considerations

The fact that Memory References may use memory directly shared with the client implies that the Trusted Application needs to be especially careful when handling such data: Even if the client is not allowed to access the shared memory buffer during an operation on this buffer, the Trusted OS usually cannot enforce this restriction. A badly-designed or rogue client may well change the content of the shared memory buffer at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application needs to be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application SHOULD always read data only once from a shared buffer and then validate it. It SHALL NOT assume that data written to the buffer can be read unchanged later on.

Error Handling

All TA interface functions except TA_DestroyEntryPoint and TA_CloseSessionEntryPoint return a return code of type TEE_Result. The behavior of the Framework when an entry point returns an error depends on the entry point called:

- If TA_CreateEntryPoint returns an error, the Trusted Application instance is not created.
- If TA_OpenSessionEntryPoint returns an error code, the client connection is rejected.
  Additionally, the error code is propagated to the client as described below.
- If TA_InvokeCommandEntryPoint returns an error code, this error code is propagated to the client.
- TA_CloseSessionEntryPoint and TA_DestroyEntryPoint cannot return an error. TA_OpenSessionEntryPoint and TA_InvokeCommandEntryPoint return codes are propagated to the client via the TEE Client API (see [Client API]) or the Internal Client API (see section 4.9) with the origin set to TEEC_ORIGIN_TRUSTED_APP.

Client Properties

When a Client connects to a Trusted Application, the Framework associates the session with Client Properties. Trusted Applications can retrieve the identity and properties of their client by calling one of the property access functions with the TEE_PROPSET_CURRENT_CLIENT. The standard Client Properties are fully specified in section 4.6.

The TA_EXPORT keyword

Depending on the compiler used and the targeted platform, a TA entry point may need to be decorated with an annotation such as __declspec(dllexport) or similar. This annotation SHALL be defined in the TEE Internal Core API header file as TA_EXPORT and placed between the entry point return type and function name as shown in the specification of each entry point.
### 4.3.1 TA_CreateEntryPoint

**Since:** TEE Internal API v1.0

```c
TEE_Result TA_EXPORT TA_CreateEntryPoint( void );
```

**Description**

The function `TA_CreateEntryPoint` is the Trusted Application's constructor, which the Framework calls when it creates a new instance of the Trusted Application.

To register instance data, the implementation of this constructor can use either global variables or the function `TEE_SetInstanceData` (described in section 4.11.2).

**Specification Number:** 10  **Function Number:** 0x102

**Return Code**

- **TEE_SUCCESS:** If the instance is successfully created, the function SHALL return `TEE_SUCCESS`.
- Any other value: If any other code is returned, then the instance is not created, and no other entry points of this instance will be called. The Framework SHALL reclaim all resources and dereference all objects related to the creation of the instance.

  If this entry point was called as a result of a client opening a session, the return code is returned to the client and the session is not opened.

**Panic Reasons**

- If the Implementation detects any error which cannot be represented by any defined or implementation defined error code.

### 4.3.2 TA_DestroyEntryPoint

**Since:** TEE Internal API v1.0

```c
void TA_EXPORT TA_DestroyEntryPoint( void );
```

**Description**

The function `TA_DestroyEntryPoint` is the Trusted Application's destructor, which the Framework calls when the instance is being destroyed.

When the function `TA_DestroyEntryPoint` is called, the Framework guarantees that no client session is currently open. Once the call to `TA_DestroyEntryPoint` has been completed, no other entry point of this instance will ever be called.

Note that when this function is called, all resources opened by the instance are still available. It is only after the function returns that the Implementation SHALL start automatically reclaiming resources left open.

After this function returns, the Implementation SHALL consider the instance destroyed and SHALL reclaim all resources left open by the instance.

**Specification Number:** 10  **Function Number:** 0x103

**Panic Reasons**

- If the Implementation detects any error.
4.3.3 TA_OpenSessionEntryPoint

Since: TEE Internal API v1.0

```
TEE_Result TAEXPORT TA_OpenSessionEntryPoint(
uint32_t   paramTypes,
[inout] TEE_Param  params[4],
[out][ctx] void**     sessionContext );
```

Description

The Framework calls the function TA_OpenSessionEntryPoint when a client requests to open a session
with the Trusted Application. The open session request may result in a new Trusted Application instance being
created as defined by the gpd.ta.singleInstance property described in section 4.5.

The client can specify parameters in an open operation which are passed to the Trusted Application instance
in the arguments paramTypes and params. These arguments can also be used by the Trusted Application
instance to transfer response data back to the client. See section 4.3.6 for a specification of how to handle the
operation parameters.

If this function returns TEE_SUCCESS, the client is connected to a Trusted Application instance and can invoke
Trusted Application commands. When the client disconnects, the Framework will eventually call the
TA_CloseSessionEntryPoint entry point.

If the function returns any error, the Framework rejects the connection and returns the return code and the
current content of the parameters to the client. The return origin is then set to TEEC_ORIGIN_TRUSTED_APP.

The Trusted Application instance can register a session data pointer by setting *sessionContext. The
framework SHALL ensure that sessionContext is a valid address of a pointer, and that it is unique per TEE
Client session.

The value of this pointer is not interpreted by the Framework, and is simply passed back to other TA_ functions
within this session. Note that *sessionContext may be set with a pointer to a memory allocated by the
Trusted Application instance or with anything else, such as an integer, a handle, etc. The Framework will not
automatically free *sessionContext when the session is closed; the Trusted Application instance is
responsible for freeing memory if required.

During the call to TA_OpenSessionEntryPoint the client may request to cancel the operation. See
section 4.10 for more details on cancellations. If the call to TA_OpenSessionEntryPoint returns
TEE_SUCCESS, the client SHALL consider the session as successfully opened and explicitly close it if
necessary.

Parameters

- paramTypes: The types of the four parameters. See section 4.3.6.1 for more information.
- params: A pointer to an array of four parameters. See section 4.3.6.2 for more information.
- sessionContext: A pointer to a variable that can be filled by the Trusted Application instance with
  an opaque void* data pointer

Note: The params parameter is defined in the prototype as an array of length 4, implementers should be
aware that the address of the start of the array is passed to the callee.

Specification Number: 10 Function Number: 0x105

Return Value

- TEE_SUCCESS: If the session is successfully opened.
• Any other value: If the session could not be opened.
  o The return code may be one of the pre-defined codes, or may be a new return code defined by the Trusted Application implementation itself. In any case, the Implementation SHALL report the return code to the client with the origin TEEC_ORIGIN_TRUSTED_APP.

Panic Reasons

• If the Implementation detects any error which cannot be expressed by any defined or implementation defined error code.
4.3.4 TA_CloseSessionEntryPoint

Since: TEE Internal API v1.0

```c
void TA_EXPORT TA_CloseSessionEntryPoint(
    [ctx] void*    sessionContext);
```

Description

The Framework calls the function `TA_CloseSessionEntryPoint` to close a client session. The Trusted Application implementation is responsible for freeing any resources consumed by the session being closed. Note that the Trusted Application cannot refuse to close a session, but can hold the closing until it returns from `TA_CloseSessionEntryPoint`. This is why this function cannot return a return code.

Parameters

- `sessionContext`: The value of the `void*` opaque data pointer set by the Trusted Application in the function `TA_OpenSessionEntryPoint` for this session.

Specification Number: 10  Function Number: 0x101

Return Value

This function can return no success or error code.

Panic Reasons

- If the Implementation detects any error.
4.3.5 TA_InvokeCommandEntryPoint

Since: TEE Internal API v1.0

```c
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
[ctx] void* sessionContext,
uint32_t commandID,
uint32_t paramTypes,
[inout] TEE_Param params[4] );
```

**Description**

The Framework calls the function TA_InvokeCommandEntryPoint when the client invokes a command within the given session.

The Trusted Application can access the parameters sent by the client through the paramTypes and params arguments. It can also use these arguments to transfer response data back to the client. See section 4.3.6 for a specification of how to handle the operation parameters.

During the call to TA_InvokeCommandEntryPoint the client may request to cancel the operation. See section 4.10 for more details on cancellations.

A command is always invoked within the context of a client session. Thus, any client property (see section 4.6) can be accessed by the command implementation.

**Parameters**

- **sessionContext**: The value of the void* opaque data pointer set by the Trusted Application in the function TA_OpenSessionEntryPoint
- **commandID**: A Trusted Application-specific code that identifies the command to be invoked
- **paramTypes**: The types of the four parameters. See section 4.3.6.1 for more information.
- **params**: A pointer to an array of four parameters. See section 4.3.6.2 for more information.

**Note**: The params parameter is defined in the prototype as an array of length 4, implementers should be aware that the address of the start of the array is passed to the callee.

**Specification Number**: 10  **Function Number**: 0x104

**Return Value**

- **TEE_SUCCESS**: If the command is successfully executed, the function SHALL return this value.
- **Any other value**: If the invocation of the command fails for any reason
  - The return code may be one of the pre-defined codes, or may be a new return code defined by the Trusted Application implementation itself. In any case, the Implementation SHALL report the return code to the client with the origin TEEC_ORIGIN_TRUSTED_APP.

**Panic Reasons**

- If the Implementation detects any error which cannot be expressed by any defined or implementation defined error code.
4.3.6 Operation Parameters in the TA Interface

When a client opens a session or invokes a command within a session, it can transmit operation parameters to the Trusted Application instance and receive response data back from the Trusted Application instance. Arguments paramTypes and params are used to encode the operation parameters and their types which are passed to the Trusted Application instance. While executing the open session or invoke command entry points, the Trusted Application can also write in params to encode the response data.

4.3.6.1 Content of paramTypes Argument

The argument paramTypes encodes the type of each of the four parameters passed to an entry point. The content of paramTypes is implementation-dependent.

Each parameter type can take one of the TEE_PARAM_TYPE_XXX values listed in Table 4-1 on page 52. The type of each parameter determines whether the parameter is used or not, whether it is a Value or a Memory Reference, and the direction of data flow between the Client and the Trusted Application instance: Input (Client to Trusted Application instance), Output (Trusted Application instance to Client), or both Input and Output. The parameter type is set to TEE_PARAM_TYPE_NONE when no parameters are passed by the client in either TEEC_OpenSession or TEEC_InvokeCommand; this includes when the operation parameter itself is set to NULL.

The following macros are available to decode paramTypes:

```c
#define TEE_PARAM_TYPES(t0,t1,t2,t3)  
   ((t0) | ((t1) << 4) | ((t2) << 8) | ((t3) << 12))
#define TEE_PARAM_TYPE_GET(t, i) (((t) >> ((i)*4)) & 0xF)
```

The macro TEE_PARAM_TYPES can be used to construct a value that you can compare against an incoming paramTypes to check the type of all the parameters in one comparison, as in the following example:

```c
if (paramTypes !=
    TEE_PARAM_TYPES(  
        TEE_PARAM_TYPE_MEMREF_INPUT,  
        TEE_PARAM_TYPE_MEMREF_OUTPUT,  
        TEE_PARAM_TYPE_NONE,  
        TEE_PARAM_TYPE_NONE))
{  
    /* Bad parameter types */  
    return TEE_ERROR_BAD_PARAMETERS;  
}
```

The macro TEE_PARAM_TYPE_GET can be used to extract the type of a given parameter from paramTypes if you need more fine-grained type checking.
### 4.3.6.2 Initial Content of params Argument

When the Framework calls the Trusted Application entry point, it initializes the content of params[i] as described in Table 4-8.

<table>
<thead>
<tr>
<th>Value of type[i]</th>
<th>Content of params[i] when the Entry Point is Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PARAM_TYPE_NONE</td>
<td>Filled with zeroes.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_OUTPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INPUT</td>
<td>params[i].value.a and params[i].value.b contain the two integers sent by the client</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INOUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INPUT</td>
<td>params[i].memref.buffer is a pointer to memory buffer shared by the client. This can be NULL.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_OUTPUT</td>
<td>params[i].memref.size describes the size of the buffer.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INOUT</td>
<td>If buffer is NULL, size is guaranteed to be zero.</td>
</tr>
</tbody>
</table>

Note that if the Client is a Client Application that uses the TEE Client API ([Client API]), the Trusted Application cannot distinguish between a registered and a temporary Memory Reference. Both are encoded as one of the TEE_PARAM_TYPE_MEMREF_XXX types and a pointer to the data is passed to the Trusted Application.

**Security Warning:** For a Memory Reference Parameter, the buffer may concurrently exist within the client and Trusted Application instance memory spaces. It SHALL therefore be assumed that the client is able to make changes to the content of this buffer asynchronously at any moment. It is a security risk to assume otherwise.

Any Trusted Application which implements functionality that needs some guarantee that the contents of a buffer are constant SHOULD copy the contents of a shared buffer into Trusted Application instance-owned memory.

To determine whether a given buffer is a Memory Reference or a buffer owned by the Trusted Application itself, the function TEE_CheckMemoryAccessRights defined in section 4.11.1 can be used.
4.3.6.3 Behavior of the Framework when the Trusted Application Returns

When the Trusted Application entry point returns, the Framework reads the content of each `params[i]` to determine what response data to send to the client, as described in Table 4-9.

Table 4-9: Interpretation of `params[i]` when Trusted Application Entry Point Returns

<table>
<thead>
<tr>
<th>Value of <code>type[i]</code></th>
<th>Behavior of the Framework when Entry Point Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PARAM_TYPE_NONE</td>
<td>The content of params[i] is ignored.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INPUT</td>
<td>The content of params[i] is ignored.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INPUT</td>
<td>The content of params[i] is ignored.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_OUTPUT</td>
<td>params[i].value.a and params[i].value.b contain the two integers sent to the client.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INOUT</td>
<td>params[i].value.a and params[i].value.b contain the two integers sent to the client.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_OUTPUT</td>
<td>The Framework reads params[i].memref.size:</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INOUT</td>
<td>- If it is equal or less than the original value of size, it is considered as the actual size of the memory buffer.</td>
</tr>
<tr>
<td></td>
<td>In this case, the Framework assumes that the Trusted Application has not written beyond this actual size and only this actual size will be synchronized with the client.</td>
</tr>
<tr>
<td></td>
<td>- If it is greater than the original value of size, it is considered as a request for a larger buffer. In this case, the Framework assumes that the Trusted Application has not written anything in the buffer and no data will be synchronized.</td>
</tr>
</tbody>
</table>
4.3.6.4 Memory Reference and Memory Synchronization

Note that if a parameter is a Memory Reference, the memory buffer may be released or unmapped immediately after the operation completes. Also, some implementations may explicitly synchronize the contents of the memory buffer before the operation starts and after the operation completes.

As a consequence:

- The Trusted Application SHALL NOT access the memory buffer after the operation completes. In particular, it cannot be used as a long-term communication means between the client and the Trusted Application instance. A Memory Reference SHALL be accessed only during the lifetime of the operation.
- The Trusted Application SHALL NOT attempt to write into a memory buffer of type TEE_PARAM_TYPE_MEMREF_INPUT.
  - It is a Programmer Error to attempt to do this but the Implementation is not required to detect this and the access may well be just ignored.
- For a Memory Reference Parameter marked as OUTPUT or INOUT, the Trusted Application can write in the entire range described by the initial content of params[i].memref.size. However, the Implementation SHALL only guarantee that the client will observe the modifications below the final value of size and only if the final value is equal or less than the original value.

For example, assume the original value of size is 100:

  - If the Trusted Application does not modify the value of size, the complete buffer is synchronized and the client is guaranteed to observe all the changes.
  - If the Trusted Application writes 50 in size, then the client is only guaranteed to observe the changes within the range from index 0 to index 49.
  - If the Trusted Application writes 200 in size, then no data is guaranteed to be synchronized with the client. However, the client will receive the new value of size. The Trusted Application can typically use this feature to tell the client that the Memory Reference was too small and request that the client retry with a Memory Reference of at least 200 bytes.

Failure to comply with these constraints will result in undefined behavior and is a Programmer Error.
4.4 Property Access Functions

This section defines a set of functions to access individual properties in a property set, to convert them into a variety of types (printable strings, integers, Booleans, binary blocks, etc.), and to enumerate the properties in a property set. These functions can be used to access TA Configuration Properties, Client Properties, and Implementation Properties.

The property set is passed to each function in a pseudo-handle parameter. Table 4-10 lists the defined property sets.

<table>
<thead>
<tr>
<th>Pseudo-Handle</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PROPSET_CURRENT_TA</td>
<td>The configuration properties for the current Trusted Application. See section 4.5 for a definition of these properties.</td>
</tr>
<tr>
<td>TEE_PROPSET_CURRENT_CLIENT</td>
<td>The properties of the current client. This pseudo-handle is valid only in the context of the following entry points:</td>
</tr>
<tr>
<td></td>
<td>o TA_OpenSessionEntryPoint</td>
</tr>
<tr>
<td></td>
<td>o TA_InvokeCommandEntryPoint</td>
</tr>
<tr>
<td></td>
<td>o TA_CloseSessionEntryPoint</td>
</tr>
<tr>
<td></td>
<td>See section 4.6 for a definition of these properties.</td>
</tr>
<tr>
<td>TEE_PROPSET_TEE_IMPLEMENTATION</td>
<td>The properties of the Trusted OS itself. See section 4.7.</td>
</tr>
</tbody>
</table>

Properties can be retrieved and converted using TEEGetPropertyAsXXX access functions (described in the following sections).

A property may be retrieved and converted into a printable string or into the type defined for the property which will be one of the following types:

- Binary block
- 32-bit unsigned integer
- 64-bit unsigned integer
- Boolean
- UUID
- Identity (a pair composed of a login method and a UUID)

Retrieving as a String

While implementations have latitude on how they set and store properties internally, a property that is retrieved via the function TEEGetPropertyAsString SHALL always be converted into a printable string encoded in UTF-8.

To ensure consistency between the representation of a property as one of the above types and its representation as a printable string encoded in UTF-8, the following conversion rules apply:

- Binary block is converted into a string that is consistent with a Base64 encoding of the binary block as defined in RFC 2045 ([RFC 2045]) section 6.8 but with the following tolerance:
  o An Implementation is allowed not to encode the final padding ‘=’ characters.
• An implementation is allowed to insert characters that are not in the Base64 character set.

• 32-bit and 64-bit unsigned integers

are converted into strings that are consistent with the following syntax:

integer:     decimal-integer
| hexadecimal-integer
| binary-integer

decimal-integer:  [0-9,_]>({K,M}?)
hexadecimal-integer:  0[x,X][0-9,a-f,A-F,_]+
binary-integer:     0[b,B][0,1,_]+

Note that the syntax allows returning the integer either in decimal, hexadecimal, or binary format, that
the representation can mix cases and can include underscores to separate groups of digits, and finally
that the decimal representation may use ‘k’ or ‘M’ to denote multiplication by 1024 or 1048576
respectively.

For example, here are a few acceptable representations of the number 1024: “1K”, “0X400”,
“0b100_0000_0000”.

• Boolean

is converted into a string equal to “true” or “false” case-insensitive, depending on the value of the
Boolean.

• UUID

is converted into a string that is consistent with the syntax defined in [RFC 4122]. Note that this string
may mix character cases.

• Identity

is converted into a string consistent with the following syntax:

identity: integer (':' uuid)?

where:

• The integer is consistent with the integer syntax described above

• If the identity UUID is Nil, then it can be omitted from the string representation of the property

Enumerating Properties

Properties in a property set can also be enumerated. For this:

• Allocate a property enumerator using the function TEE_AllocatePropertyEnumerator.

• Start the enumeration by calling TEE_StartPropertyEnumerator, passing the pseudo-handle on
the desired property set.

• Call the functions TEE_GetProperty[AsXXX] with the enumerator handle and a NULL name.

An enumerator provides the properties in an arbitrary order. In particular, they are not required to be sorted by
name although a given implementation may ensure this.
4.4.1 TEE_GetPropertyAsString

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_GetPropertyAsString(
    [instringopt] TEE_PropSetHandle propsetOrEnumerator,
    [instring] char* name,
    [outstring] char* valueBuffer, size_t* valueBufferLen );
```

Description

The TEE_GetPropertyAsString function performs a lookup in a property set to retrieve an individual property and convert its value into a printable string.

When the lookup succeeds, the implementation SHALL convert the property into a printable string and copy the result into the buffer described by valueBuffer and valueBufferLen.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and it SHALL be encoded in UTF-8.
  - If propsetOrEnumerator is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name SHALL NOT be NULL
- valueBuffer, valueBufferLen: Output buffer for the property value

Specification Number: 10 Function Number: 0x207

Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_SHORT_BUFFER: If the value buffer is not large enough to hold the whole property value

Panic Reasons

- If the implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the valueBufferLen.
4.4.2 TEE_GetPropertyAsBool

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetPropertyAsBool(
    TEE_PropSetHandle propsetOrEnumerator,
    [instringopt] char*              name,
    [out]       bool*              value );
```

Description

The `TEE_GetPropertyAsBool` function retrieves a single property in a property set and converts its value to a Boolean.

If a property cannot be viewed as a Boolean, this function SHALL return `TEE_ERROR_BAD_FORMAT`.

Parameters

- `propsetOrEnumerator`: One of the `TEE_PROPSET_XXX` pseudo-handles or a handle on a property enumerator
- `name`: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` SHALL NOT be `NULL`.
- `value`: A pointer to the variable that will contain the value of the property on success or `false` on error.

Specification Number: 10  Function Number: 0x205

Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the property is not found or if `name` is not a valid UTF-8 encoding
- `TEE_ERROR_BAD_FORMAT`: If the property value is not defined as a Boolean

Panic Reasons

- If the implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
4.4.3 TEE_GetPropertyAsU32

4.4.3.1 TEE_GetPropertyAsU32

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetPropertyAsU32(
    TEE_PropSetHandle propsetOrEnumerator,
    [instringopt] char* name,
    [out] uint32_t* value);
```

Description

The TEE_GetPropertyAsU32 function retrieves a single property in a property set and converts its value to a 32-bit unsigned integer.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If propsetOrEnumerator is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name SHALL NOT be NULL.
- value: A pointer to the variable that will contain the value of the property on success, or zero on error.

Specification Number: 10  Function Number: 0x208

Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property value is not defined as an unsigned 32-bit integer

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
4.4.3.2 TEE_GetPropertyAsU64

Since: TEE Internal Core API v1.2

```
TEE_Result TEE_GetPropertyAsU64(
    TEE_PropSetHandle propsetOrEnumerator,
    [instringopt] char* name,
    [out] uint64_t* value );
```

Description

The `TEE_GetPropertyAsU64` function retrieves a single property in a property set and converts its value to a 64-bit unsigned integer. If the underlying value is a 32-bit integer, the Trusted OS SHALL zero extend it.

Parameters

- `propsetOrEnumerator`: One of the `TEE_PROPSET_XXX` pseudo-handles or a handle on a property enumerator
- `name`: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` SHALL NOT be `NULL`.
- `value`: A pointer to the variable that will contain the value of the property on success, or zero on error.

Specification Number: 10   Function Number: 0x20D

Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the property is not found or if `name` is not a valid UTF-8 encoding
- `TEE_ERROR_BAD_FORMAT`: If the property value is not defined as an unsigned 64-bit integer

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
4.4.4 TEE_GetPropertyAsBinaryBlock

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_GetPropertyAsBinaryBlock(
    TEE_PropSetHandle propsetOrEnumerator,
    [instringopt] char* name,
    [outbuf] void* valueBuffer, size_t* valueBufferLen );
```

Description

The function TEE_GetPropertyAsBinaryBlock retrieves an individual property and converts its value into a binary block.

If a property cannot be viewed as a binary block, this function SHALL return TEE_ERROR_BAD_FORMAT.

Parameters

- propsetOrEnumerator: One of the TEE_PROPSET_XXX pseudo-handles or a handle on a property enumerator
- name: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If propsetOrEnumerator is a property enumerator handle, name is ignored and can be NULL.
  - Otherwise, name SHALL NOT be NULL.
- valueBuffer, valueBufferLen: Output buffer for the binary block

Specification Number: 10 Function Number: 0x204

Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If the property is not found or if name is not a valid UTF-8 encoding
- TEE_ERROR_BAD_FORMAT: If the property cannot be retrieved as a binary block
- TEE_ERROR_SHORT_BUFFER: If the value buffer is not large enough to hold the whole property value

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the valueBufferLen.
4.4.5 TEE_GetPropertyAsUUID

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetPropertyAsUUID(
    [instringopt] char*              name,
    [out] TEE_UUID*          value);
```

Description

The function `TEE_GetPropertyAsUUID` retrieves an individual property and converts its value into a UUID.

If a property cannot be viewed as a UUID, this function SHALL return `TEE_ERROR_BAD_FORMAT`.

Parameters

- `propsetOrEnumerator`: One of the `TEE_PROPSET_XXX` pseudo-handles or a handle on a property enumerator
- `name`: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` SHALL NOT be `NULL`.
- `value`: A pointer filled with the UUID. SHALL NOT be `NULL`.

Specification Number: 10 Function Number: 0x209

Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the property is not found or if `name` is not a valid UTF-8 encoding
- `TEE_ERROR_BAD_FORMAT`: If the property cannot be converted into a UUID

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
### 4.4.6 TEE_GetPropertyAsIdentity

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetPropertyAsIdentity(    
    [instringopt] char* name, 
    [out] TEE_Identity* value );
```

#### Description

The function `TEE_GetPropertyAsIdentity` retrieves an individual property and converts its value into a `TEE_Identity`. If a property cannot be viewed as an identity, this function SHALL return `TEE_ERROR_BAD_FORMAT`.

#### Parameters

- `propsetOrEnumerator`: One of the `TEE_PROPSET_XXX` pseudo-handles or a handle on a property enumerator.
- `name`: A pointer to the zero-terminated string containing the name of the property to retrieve. Its content is case-sensitive and SHALL be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` SHALL NOT be `NULL`.
- `value`: A pointer filled with the identity. SHALL NOT be `NULL`.

#### Specification Number: 10    Function Number: 0x206

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the property is not found or if `name` is not a valid UTF-8 encoding.
- `TEE_ERROR_BAD_FORMAT`: If the property value cannot be converted into an Identity.

#### Panic Reasons

- If the implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
4.4.7 TEE_AllocatePropertyEnumerator

Since: TEE Internal API v1.0

```c
TEE_Result TEE_AllocatePropertyEnumerator(
    [out] TEE_PropSetHandle* enumerator );
```

Description

The function TEE_AllocatePropertyEnumerator allocates a property enumerator object. Once a handle on a property enumerator has been allocated, it can be used to enumerate properties in a property set using the function TEE_StartPropertyEnumerator.

Parameters

- enumerator: A pointer filled with an opaque handle on the property enumerator on success and with TEE_HANDLE_NULL on error

Specification Number: 10  Function Number: 0x201

Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OUT_OF_MEMORY: If there are not enough resources to allocate the property enumerator

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
### 4.4.8 TEE_FreePropertyEnumerator

**Since:** TEE Internal API v1.0

```c
void TEE_FreePropertyEnumerator(
    TEE_PropSetHandle enumerator );
```

**Description**

The function `TEE_FreePropertyEnumerator` deallocates a property enumerator object.

**Parameters**

- `enumerator`: A handle on the enumerator to free

**Specification Number:** 10  **Function Number:** 0x202

**Panic Reasons**

- If the Implementation detects any error.

### 4.4.9 TEE_StartPropertyEnumerator

**Since:** TEE Internal API v1.0

```c
void TEE_StartPropertyEnumerator(
    TEE_PropSetHandle enumerator,
    TEE_PropSetHandle propSet );
```

**Description**

The function `TEE_StartPropertyEnumerator` starts to enumerate the properties in an enumerator.

Once an enumerator is attached to a property set:

- Properties can be retrieved using one of the `TEE_GetPropertyAsXXX` functions, passing the enumerator handle as the property set and NULL as the name.
- The function `TEE_GetPropertyName` can be used to retrieve the name of the current property in the enumerator.
- The function `TEE_GetNextProperty` can be used to advance the enumeration to the next property in the property set.

**Parameters**

- `enumerator`: A handle on the enumerator
- `propSet`: A pseudo-handle on the property set to enumerate. SHALL be one of the `TEE_PROPSET_XXX` pseudo-handles.

**Specification Number:** 10  **Function Number:** 0x20C

**Panic Reasons**

- If the Implementation detects any error.
4.4.10 TEE_ResetPropertyEnumerator

Since: TEE Internal API v1.0

```c
void TEE_ResetPropertyEnumerator(
    TEE_PropSetHandle enumerator);
```

Description

The function TEE_ResetPropertyEnumerator resets a property enumerator to its state immediately after allocation. If an enumeration is currently started, it is abandoned.

Parameters

- `enumerator`: A handle on the enumerator to reset

Specification Number: 10  Function Number: 0x20B

Panic Reasons

- If the Implementation detects any error.
4.4.11  TEE_GetPropertyName

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_GetPropertyName(
    TEE_PropSetHandle enumerator,
    [outstring] void* nameBuffer, size_t* nameBufferLen );
```

Description

The function TEE_GetPropertyName gets the name of the current property in an enumerator.

The property name SHALL be the valid UTF-8 encoding of a Unicode string containing no intermediate U+0000 code points.

Parameters

- enumerator: A handle on the enumerator
- nameBuffer, nameBufferLen: The buffer filled with the name

Specification Number: 10  Function Number: 0x20A

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If there is no current property either because the enumerator has not started or because it has reached the end of the property set
- TEE_ERROR_SHORT_BUFFER: If the name buffer is not large enough to contain the property name

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the nameBufferLen.
### 4.4.12 TEE_GetNextProperty

**Since:** TEE Internal API v1.0

```c
TEE_Result TEE_GetNextProperty(
    TEE_PropSetHandle enumerator);
```

**Description**

The function `TEE_GetNextProperty` advances the enumerator to the next property.

**Parameters**

- `enumerator`: A handle on the enumerator

**Specification Number:** 10  **Function Number:** 0x203

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the enumerator has reached the end of the property set or if it has not started

**Panic Reasons**

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
4.5 Trusted Application Configuration Properties

Each Trusted Application is associated with Configuration Properties that are accessible using the generic Property Access Functions and the TEE_PROPSET_CURRENT_TA pseudo-handle. This section defines a few standard configuration properties that affect the behavior of the Implementation. Other configuration properties can be defined:

- either by the Implementation to configure implementation-defined behaviors,
- or by the Trusted Application itself for its own configuration purposes.

The way properties are actually configured and attached to a Trusted Application is beyond the scope of the specification.

Table 4-11 defines the standard configuration properties for Trusted Applications.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpd.ta.appID</td>
<td>UUID</td>
<td>Since: TEE Internal API v1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The identifier of the Trusted Application.</td>
</tr>
<tr>
<td>gpd.ta.singleInstance</td>
<td>Boolean</td>
<td>Since: TEE Internal API v1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether the Implementation SHALL create a single TA instance for all the client sessions (if true) or SHALL create a separate instance for each client session (if false).</td>
</tr>
<tr>
<td>gpd.ta.multiSession</td>
<td>Boolean</td>
<td>Since: TEE Internal API v1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whether the Trusted Application instance supports multiple sessions. This property is ignored when gpd.ta.singleInstance is set to false.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gpd.ta.instanceKeepAlive</td>
<td>Boolean</td>
<td>Whether the Trusted Application instance context SHALL be preserved when there are no sessions connected to the instance. The instance context is defined as all writable data within the memory space of the Trusted Application instance, including the instance heap. This property is meaningful only when the gpd.ta.singleInstance is set to true. When this property is set to false, then the TA instance SHALL be created when one or more sessions are opened on the TA and it SHALL be destroyed when there are no more sessions opened on the instance. When this property is set to true, then the TA instance is terminated only when the TEE shuts down, which includes when the device goes through a system-wide global power cycle. Note that the TEE SHALL NOT shut down whenever the REE does not shut down and keeps a restorable state, including when it goes through transitions into lower power states (hibernation, suspend, etc.). The exact moment when a keep-alive single instance is created is implementation-defined but it SHALL be no later than the first session opening.</td>
</tr>
<tr>
<td>gpd.ta.dataSize</td>
<td>Integer</td>
<td>Maximum estimated amount of dynamic data in bytes configured for the Trusted Application. The memory blocks allocated through TEE_Malloc are drawn from this space, as well as the task stacks. How this value precisely relates to the exact number and sizes of blocks that can be allocated is implementation-dependent.</td>
</tr>
<tr>
<td>gpd.ta.stackSize</td>
<td>Integer</td>
<td>Maximum stack size in bytes available to any task in the Trusted Application at any point in time. This corresponds to the stack size used by the TA code itself and does not include stack space possibly used by the Trusted Core Framework. For example, if this property is set to “512”, then the Framework SHALL guarantee that, at any time, the TA code can consume up to 512 bytes of stack and still be able to call any functions in the API.</td>
</tr>
<tr>
<td>gpd.ta.version</td>
<td>String</td>
<td>Version number of this Trusted Application.</td>
</tr>
<tr>
<td>gpd.ta.description</td>
<td>String</td>
<td>Optional description of the Trusted Application.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| gpd.ta.endian  | Integer| **Since**: TEE Internal Core API v1.2  
Endianness of the current TA. Legal values are:  
- The value 0 indicates little-endian TA.  
- The value 1 indicates a big-endian TA.  
- Values from 2 to 0x7FFFFFFF are reserved for future versions of this specification.  
- Values in the range 0x80000000 to 0xFFFFFFFF are implementation defined. |
4.6 Client Properties

This section defines the standard Client Properties, accessible using the generic Property Access Functions and the TEE_PROPSET_CURRENT_CLIENT pseudo-handle. Other non-standard client properties can be defined by specific implementations, but they SHALL be defined outside the “gpd.” namespace.

Note that Client Properties can be accessed only in the context of a TA entry point associated with a client, i.e. in one of the following entry point functions: TA_OpenSessionEntryPoint, TA_InvokeCommandEntryPoint, or TA_CloseSessionEntryPoint.

Table 4-12 defines the standard Client Properties.

Table 4-12: Standard Client Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpd.client.identity</td>
<td>Identity</td>
<td>Identity of the current client. This can be conveniently retrieved using the function TEEGetPropertyAsIdentity (see section 4.4.6). A Trusted Application can use the client identity to perform access control. For example, it can refuse to open a session for a client that is not identified.</td>
</tr>
<tr>
<td>gpd.client.endian</td>
<td>Integer</td>
<td>Endianness of the current client. Legal values are as defined for gpd.ta.endian.</td>
</tr>
</tbody>
</table>

As shown in Table 4-13, the client identifier and the client properties that the Trusted Application can retrieve depend on the nature of the client and the method it has used to connect. (The constant values associated with the login methods are listed in Table 4-2.)

Table 4-13: Client Identities

<table>
<thead>
<tr>
<th>Login Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_LOGIN_PUBLIC</td>
<td>The client is in the Rich Execution Environment and is neither identified nor authenticated. The client has no identity and the UUID is the Nil UUID as defined in [RFC 4122].</td>
</tr>
<tr>
<td>TEE_LOGIN_USER</td>
<td>The Client Application has been identified by the Rich Execution Environment and the client UUID reflects the actual user that runs the calling application independently of the actual application.</td>
</tr>
<tr>
<td>TEE_LOGIN_GROUP</td>
<td>The client UUID reflects a group identity that is executing the calling application. The notion of group identity and the corresponding UUID is REE-specific.</td>
</tr>
<tr>
<td>TEE_LOGIN_APPLICATION</td>
<td>The Client Application has been identified by the Rich Execution Environment independently of the identity of the user executing the application. The nature of this identification and the corresponding UUID is REE-specific.</td>
</tr>
<tr>
<td>TEE_LOGIN_APPLICATION_USER</td>
<td>The client UUID identifies both the calling application and the user that is executing it.</td>
</tr>
</tbody>
</table>
### Login Method

<table>
<thead>
<tr>
<th>Login Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_LOGIN_APPLICATION_GROUP</td>
<td>The client UUID identifies both the calling application and a group that is executing it.</td>
</tr>
<tr>
<td>TEE_LOGIN_TRUSTED_APP</td>
<td>The client is another Trusted Application. The client identity assigned to this session is the UUID of the calling Trusted Application. The client properties are all the configuration properties of the calling Trusted Application.</td>
</tr>
<tr>
<td></td>
<td>The range (0x80000000–0xEFFFFFFF) is reserved for implementation-defined login methods.</td>
</tr>
<tr>
<td></td>
<td>The meaning of the Client UUID and the associated client properties are implementation-defined. If the Trusted Application does not support the particular implementation, it SHOULD assume that the client has minimum rights, i.e. rights equivalent to the login method TEE_LOGIN_PUBLIC.</td>
</tr>
<tr>
<td>Other values are reserved for GlobalPlatform use, as described in Table 4-2.</td>
<td></td>
</tr>
</tbody>
</table>

Client properties are meant to be managed by either the Rich OS or the Trusted OS and these SHALL ensure that a Client cannot tamper with its own properties in the following sense:

- The property \(\text{gpd.client.identity}\) SHALL always be determined by the Trusted OS and the determination of whether or not it is equal to TEE_LOGIN_TRUSTED_APP SHALL be as trustworthy as the Trusted OS itself.
- When \(\text{gpd.client.identity}\) is equal to TEE_LOGIN_TRUSTED_APP then the Trusted OS SHALL ensure that the remaining properties are equal to the properties of the calling TA up to the same level of trustworthiness that the target TA places in the Trusted OS.
- When \(\text{gpd.client.identity}\) is not equal to TEE_LOGIN_TRUSTED_APP, then the Rich OS is responsible for ensuring that the Client Application cannot tamper with its own properties.

Note that if a Client wants to transmit a property that is not synthesized by the Rich OS or Trusted OS, such as a password, then it SHALL use a parameter to the session open operation or in subsequent commands.
### 4.7 Implementation Properties

The implementation properties can be retrieved by the generic Property Access Functions with the `TEE_PROPSET_TEE_IMPLEMENTATION` pseudo-handle.

Table 4-14 defines the standard implementation properties.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gpd.tee.apiversion</code></td>
<td>String</td>
<td>Since: TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1.2&lt;br&gt;A string composed of the Major and Minor version of the specification, e.g. “1.1”.&lt;br&gt;Zero values must be represented (e.g. version 3.0 is “3.0”). This string does NOT include any other parts of the version number.&lt;br&gt;(This property is deprecated in favor of <code>gpd.tee.internalCore.version</code>.)</td>
</tr>
<tr>
<td><code>gpd.tee.internalCore.version</code></td>
<td>Integer</td>
<td>Since: TEE Internal Core API v1.1.2&lt;br&gt;The TEE Internal Core API Specification version number expressed as an integer. See section 4.7.1 for details of the structure of this integer field.</td>
</tr>
<tr>
<td><code>gpd.tee.description</code></td>
<td>String</td>
<td>Since: TEE Internal API v1.0&lt;br&gt;A description of the implementation. The content of this property is implementation-dependent but typically contains a version and build number of the implementation as well as other configuration information.&lt;br&gt;Note that implementations are free to define their own non-standard identification property names, provided they are not in the &quot;gpd.&quot; namespace.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gpd.tee.deviceID</td>
<td>UUID</td>
<td><strong>Since</strong>: TEE Internal API v1.0&lt;br&gt;A device identifier that SHALL be globally unique among all GlobalPlatform TEEs whatever the manufacturer, vendor, or integration.&lt;br&gt;<strong>Since</strong>: TEE Internal Core API v1.1.1&lt;br&gt;If there are multiple GlobalPlatform TEEs on one device, each such TEE SHALL have a unique <code>gpd.tee.deviceID</code>.&lt;br&gt;&lt;strong&gt;Implementer's Note&lt;/strong&gt;&lt;br&gt;It is acceptable to derive this device identifier from statistically unique secret or public information, such as a Hardware Unique Key, die identifiers, etc. However, note that this property is intended to be public and exposed to any software running on the device, not only to Trusted Applications. The derivation SHALL therefore be carefully designed so that it does not compromise secret information.</td>
</tr>
<tr>
<td>gpd.tee.systemTime.protectionLevel</td>
<td>Integer</td>
<td><strong>Since</strong>: TEE Internal API v1.0&lt;br&gt;The protection level provided by the system time implementation. See the function <code>TEE_GetSystemTime</code> in section 7.2.1 for more details.</td>
</tr>
<tr>
<td>gpd.tee.TAPersistentTime.protectionLevel</td>
<td>Integer</td>
<td><strong>Since</strong>: TEE Internal API v1.0&lt;br&gt;The protection level provided for the TA Persistent Time. See the function <code>TEE_GetTAPersistentTime</code> in section 7.2.3 for more details.</td>
</tr>
<tr>
<td>gpd.tee.arith.maxBigIntSize</td>
<td>Integer</td>
<td><strong>Since</strong>: TEE Internal API v1.0&lt;br&gt;Maximum size in bits of the big integers for all the functions in the TEE Arithmetical API specified in Chapter 8. Beyond this limit, some of the functions MAY panic due to insufficient pre-allocated resources or hardware limitations.</td>
</tr>
<tr>
<td>gpd.tee.cryptography.ecc</td>
<td>Boolean</td>
<td><strong>Since</strong>: TEE Internal Core API v1.1; deprecated in TEE Internal Core API v1.2&lt;br&gt;If set to <code>true</code>, then the Elliptic Curve Cryptographic (ECC) algorithms shown in Table 6-2 are supported.&lt;br&gt;(This property is deprecated; however, see section 6.10.3 regarding responding when this property is queried.)</td>
</tr>
<tr>
<td>Property Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| gpd.tee.cryptography.nist    | Boolean | Since: TEE Internal Core API v1.2  
If set to `true`, then all of the cryptographic elements defined in this specification in Table 6-14 with the Source column marked NIST are supported.  
If it is set to `false` or is absent, it does not mean that none of these cryptographic elements are supported. See TEE_IsAlgorithmSupported in section 6.2.9. |
| gpd.tee.cryptography.bsi-r   |       | Since: TEE Internal Core API v1.2  
If set to `true`, then all of the cryptographic elements defined in this specification in Table 6-14 with the Source column marked BSI-R are supported.  
If it is set to `false` or is absent, it does not mean that none of these cryptographic elements are supported. See TEE_IsAlgorithmSupported in section 6.2.9. |
| gpd.tee.cryptography.bsi-t   |       | Since: TEE Internal Core API v1.2  
If set to `true`, then all of the cryptographic elements defined in this specification in Table 6-14 with the Source column marked BSI-T are supported.  
If it is set to `false` or is absent, it does not mean that none of these cryptographic elements are supported. See TEE_IsAlgorithmSupported in section 6.2.9. |
| gpd.tee.cryptography.ietf    |       | Since: TEE Internal Core API v1.2  
If set to `true`, then all of the cryptographic elements defined in this specification in Table 6-14 with the Source column marked IETF are supported.  
If it is set to `false` or is absent, it does not mean that none of these cryptographic elements are supported. See TEE_IsAlgorithmSupported in section 6.2.9. |
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpd.tee.cryptography.octa</td>
<td>Since: TEE Internal Core API v1.2 If set to true, then the cryptographic elements defined in this specification in Table 6-14 with the Source column marked OCTA are supported. In addition, all definitions related to SM3 and SM4 are also supported. If it is set to false or is absent, it does not mean that none of these cryptographic elements are supported. See TEE_IsAlgorithmSupported in section 6.2.9.</td>
<td></td>
</tr>
<tr>
<td>gpd.tee.trustedStorage.antiRollback.protectionLevel</td>
<td>Integer</td>
<td>Since: TEE Internal Core API v1.1 Indicates the level of protection from rollback of Trusted Storage supplied by the implementation: 0 (or missing): No anti rollback protection 100: Anti rollback mechanism for the Trusted Storage is enforced at the REE level. 1000: Anti rollback mechanism for the Trusted Storage is based on TEE-controlled hardware. This hardware SHALL be out of reach of software attacks from the REE. If an active TA attempts to access material held in Trusted Storage that has been rolled back, it will receive an error equivalent to a corrupted object. Users may still be able to roll back the Trusted Storage but this SHALL be detected by the Implementation</td>
</tr>
<tr>
<td>gpd.tee.trustedos.implementation.version</td>
<td>String</td>
<td>Since: TEE Internal Core API v1.1 The detailed version number of the Trusted OS. The value of this property SHALL change whenever anything changes in the code forming the Trusted OS which provides the TEE, i.e. any patch SHALL change this string.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Type</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| gpd.tee.trustedos.implementation.binversion | binary  | **Since:** TEE Internal Core API v1.1  
A binary value which is equivalent to gpd.tee.trustedos.implementation.version. May be derived from some form of certificate indicating the software has been signed, a measurement of the image, a checksum, a direct binary conversion of gpd.tee.trustedos.implementation.version, or any other binary value which the TEE manufacturer chooses to provide. The Trusted OS manufacturer's documentation SHALL state the format of this value.  
The value of this property SHALL change whenever anything changes in the code forming the Trusted OS which provides the TEE, i.e. any patch SHALL change this binary. |
| gpd.tee.trustedos.manufacturer    | String  | **Since:** TEE Internal Core API v1.1  
Name of the manufacturer of the Trusted OS.                                                                                                                                                                                                                                                                                         |
| gpd.tee.firmware.implementation.version | String  | **Since:** TEE Internal Core API v1.1  
The detailed version number of the firmware which supports the Trusted OS implementation. This includes all privileged software involved in the secure booting and support of the TEE apart from the secure OS and Trusted Applications.  
The value of this property SHALL change whenever any changes in this code, i.e. any patch SHALL change this string. The value of this property MAY be the empty string if there is no such software. |
| gpd.tee.firmware.implementation.binversion | Binary  | **Since:** TEE Internal Core API v1.1  
A binary value which is equivalent to gpd.tee.firmware.implementation.version. May be derived from some form of certificate indicating the firmware has been signed, a measurement of the image, a checksum, a direct binary conversion of gpd.tee.firmware.implementation.version, or any other binary value which the Trusted OS manufacturer chooses to provide. The Trusted OS manufacturer's documentation SHALL state the format of this value.  
The value of this property SHALL change whenever anything changes in this code, i.e. any patch SHALL change this binary. The value of this property MAY be a zero length value if there is no such firmware. |
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| gpd.tee.firmware.manufacturer             | String    | Since: TEE Internal Core API v1.1  
Name of the manufacturer of the firmware which supports the Trusted OS or the empty string if there is no such firmware. |
| gpd.tee.event.maxSources                  | Integer   | Since: TEE Internal Core API v1.2  
The maximum number of secure event sources the implementation can support.                                                              |
4.7.1 Specification Version Number Property

This specification defines a TEE property containing the version number of the specification that the implementation conforms to. The property can be retrieved using the normal Property Access Functions. The property SHALL be named "gpd.tee.internalCore.version" and SHALL be of integer type with the interpretation given below.

The specification version number property consists of four positions: major, minor, maintenance, and RFU. These four bytes are combined into a 32-bit unsigned integer as follows:

- The major version number of the specification is placed in the most significant byte.
- The minor version number of the specification is placed in the second most significant byte.
- The maintenance version number of the specification is placed in the second least significant byte. If the version is not a maintenance version, this SHALL be zero.
- The least significant byte is reserved for future use. Currently this byte SHALL be zero.

<table>
<thead>
<tr>
<th>Major version number of the specification</th>
<th>Minor version number of the specification</th>
<th>Maintenance version number of the specification</th>
<th>Reserved for future use. Currently SHALL be zero.</th>
</tr>
</thead>
</table>

So, for example:

- Specification version 1.1 will be held as 0x01010000 (16842752 in base 10).
- Specification version 1.2 will be held as 0x01020000 (16908288 in base 10).
- Specification version 1.2.3 will be held as 0x01020300 (16909056 in base 10).
- Specification version 12.13.14 will be held as 0x0C0D0E00 (202182144 in base 10).
- Specification version 212.213.214 will be held as 0xD4D5D600 (3570783744 in base 10).

This places the following requirement on the version numbering:
- No specification can have a Major or Minor or Maintenance version number greater than 255.
4.8 Panics

4.8.1 TEE_Panic

Since: TEE Internal API v1.0

```c
void TEE_Panic(TEE_Result panicCode);
```

Description

The TEE_Panic function raises a Panic in the Trusted Application instance.

When a Trusted Application calls the TEE_Panic function, the current instance SHALL be destroyed and all
the resources opened by the instance SHALL be reclaimed. All sessions opened from the panicking instance
on another TA SHALL be gracefully closed and all cryptographic objects and operations SHALL be closed
properly.

When an instance panics, its clients receive the return code TEE_ERROR_TARGET_DEAD of origin
TEE_ORIGIN_TEE until they close their session. This applies to Rich Execution Environment clients calling
through the TEE Client API (see [Client API]) and to Trusted Execution Environment clients calling through the
Internal Client API (see section 4.9).

When this routine is called, an Implementation in a non-production environment, such as in a development or
pre-production state, SHALL display the supplied panicCode using the mechanisms defined in
[TEE TA Debug] (or an implementation-specific alternative) to help the developer understand the Programmer
Error. Diagnostic information SHOULD NOT be exposed outside of a secure development environment.

Once an instance is panicked, no TA entry point is ever called again for this instance, not even
TA_DestroyEntryPoint. The caller cannot expect that the TEE_Panic function will return.

Parameters

- panicCode: An informative panic code defined by the TA. May be displayed in traces if traces are
  available.

Specification Number: 10  Function Number: 0x301
4.9 Internal Client API

This API allows a Trusted Application to act as a client to another Trusted Application.

4.9.1 TEE_OpenTASession

Since: TEE Internal API v1.2

```c
TEE_Result TEE_OpenTASession(
    [in] TEE_UUID* destination,
    uint32_t cancellationRequestTimeout,
    uint32_t paramTypes,
    [inout] TEE_Param params[4],
    [out] TEE_TASessionHandle* session,
    [out] uint32_t* returnOrigin);
```

Description

The function TEE_OpenTASession opens a new session with a Trusted Application. The destination Trusted Application is identified by its UUID passed in destination. A set of four parameters can be passed during the operation. See section 4.9.4 for a detailed specification of how these parameters are passed in the paramTypes and params arguments.

The result of this function is returned both in the return code and the return origin, stored in the variable pointed to by returnOrigin:

- If the return origin is different from TEE_ORIGIN_TRUSTED_APP, then the function has failed before it could reach the target Trusted Application. The possible return codes are listed in “Return Code” below.
- If the return origin is TEE_ORIGIN_TRUSTED_APP, then the meaning of the return value depends on the protocol exposed by the target Trusted Application. However, if TEE_SUCCESS is returned, it always means that the session was successfully opened and if the function returns a value different from TEE_SUCCESS, it means that the session opening failed.

When the session is successfully opened, i.e. when the function returns TEE_SUCCESS, a valid session handle is written into *session. Otherwise, the value TEE_HANDLE_NULL is written into *session.

Parameters

- destination: A pointer to a TEE_UUID structure containing the UUID of the destination Trusted Application
- cancellationRequestTimeout: Timeout in milliseconds or the special value TEE_TIMEOUT_INFINITE if there is no timeout. After the timeout expires, the TEE SHALL act as though a cancellation request for the operation had been sent.
- paramTypes: The types of all parameters passed in the operation. See section 4.9.4 for more details.
- params: The parameters passed in the operation. See section 4.9.4 for more details. These are updated only if the returnOrigin is TEE_ORIGIN_TRUSTED_APP.
- session: A pointer to a variable that will receive the client session handle. The pointer SHALL NOT be NULL. The value is set to TEE_HANDLE_NULL upon error.
- returnOrigin: A pointer to a variable which will contain the return origin. This field may be NULL if the return origin is not needed.
Note: The `params` parameter is defined in the prototype as an array of length 4, implementers should be aware that the address of the start of the array is passed to the callee.

**Specification Number:** 10  **Function Number:** 0x403

**Return Code**

- **TEE_SUCCESS:** In case of success; the session was successfully opened.
- Any other value: The opening failed.

If the return origin is different from `TEE_ORIGIN_TRUSTED_APP`, one of the following return codes can be returned:

- **TEE_ERROR_OUT_OF_MEMORY:** If not enough resources are available to open the session
- **TEE_ERROR_ITEM_NOT_FOUND:** If no Trusted Application matches the requested destination UUID
- **TEE_ERROR_ACCESS_DENIED:** If access to the destination Trusted Application is denied
- **TEE_ERROR_BUSY:** If the destination Trusted Application does not allow more than one session at a time and already has a session in progress
- **TEE_ERROR_TARGET_DEAD:** If the destination Trusted Application has panicked during the operation
- **TEE_ERROR_CANCEL:** If the request is cancelled by anything other than the destination Trusted Application

If the return origin is `TEE_ORIGIN_TRUSTED_APP`, the return code is defined by the protocol exposed by the destination Trusted Application.

**Panic Reasons**

- If the implementation detects any error which cannot be represented by any defined or implementation defined error code.
- If memory which was allocated with `TEE_MALLOC_NO_SHARE` is referenced by one of the parameters.

**Backward Compatibility**

The error code `TEE_CANCEL` was added in TEE Internal Core v1.2.

### 4.9.2 TEE_CloseTASession

**Since:** TEE Internal API v1.0

```c
void TEE_CloseTASession(TEE_TASessionHandle session);
```

**Description**

The function `TEE_CloseTASession` closes a client session.

**Parameters**

- **session:** An opened session handle
Specification Number: 10  Function Number: 0x401

Panic Reasons

- If the Implementation detects any error.
4.9.3 TEE_InvokeTACommand

Since: TEE Internal API v1.2

```c
TEE_Result TEE_InvokeTACommand(
    TEE_TASessionHandle session,
    uint32_t cancellationRequestTimeout,
    uint32_t commandID,
    uint32_t paramTypes,
    [inout] TEE_Param params[4],
    [out] uint32_t* returnOrigin);
```

**Description**

The function TEE_InvokeTACommand invokes a command within a session opened between the client Trusted Application instance and a destination Trusted Application instance. The parameter `session` SHALL reference a valid session handle opened by TEE_OpenTASession. Up to four parameters can be passed during the operation. See section 4.9.4 for a detailed specification of how these parameters are passed in the `paramTypes` and `params` arguments.

The result of this function is returned both in the return value and the return origin, stored in the variable pointed to by `returnOrigin`:

- If the return origin is different from TEE_ORIGIN_TRUSTED_APP, then the function has failed before it could reach the destination Trusted Application. The possible return codes are listed in “Return Code” below.
- If the return origin is TEE_ORIGIN_TRUSTED_APP, then the meaning of the return value is determined by the protocol exposed by the destination Trusted Application. It is recommended that the Trusted Application developer choose TEE_SUCCESS (0) to indicate success in their protocol, as this makes it possible to determine success or failure without looking at the return origin.

**Parameters**

- **session**: An opened session handle
- **cancellationRequestTimeout**: Timeout in milliseconds or the special value TEE_TIMEOUT_INFINITE if there is no timeout. After the timeout expires, the TEE SHALL act as though a cancellation request for the operation had been sent.
- **commandID**: The identifier of the Command to invoke. The meaning of each Command Identifier SHALL be defined in the protocol exposed by the target Trusted Application.
- **paramTypes**: The types of all parameters passed in the operation. See section 4.9.4 for more details.
- **params**: The parameters passed in the operation. See section 4.9.4 for more details.
- **returnOrigin**: A pointer to a variable which will contain the return origin. This field may be NULL if the return origin is not needed.

**Note**: The `params` parameter is defined in the prototype as an array of length 4, implementers should be aware that the address of the start of the array is passed to the callee.

**Specification Number**: 10  **Function Number**: 0x402

**Return Code**

- If the return origin is different from TEE_ORIGIN_TRUSTED_APP, one of the following return codes can be returned:
- TEE_SUCCESS: In case of success.
- TEE_ERROR_OUT_OF_MEMORY: If not enough resources are available to perform the operation.
- TEE_ERROR_TARGET_DEAD: If the destination Trusted Application has panicked during the operation.
- TEE_ERROR_CANCEL: If the request is cancelled by anything other than the destination Trusted Application.

- If the return origin is TEE_ORIGIN_TRUSTED_APP, the return code is defined by the protocol exposed by the destination Trusted Application.

**Panic Reasons**

- If the implementation detects that the security characteristics of a memory buffer would be downgraded by the requested access rights. See Table 4-5.
- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
- If memory which was allocated with TEE_MALLOC_NO_SHARE is referenced by one of the parameters.

**Backward Compatibility**

- The error code TEE_CANCEL was added in TEE Internal Core v1.2.
4.9.4 Operation Parameters in the Internal Client API

The functions TEE_OpenTASession and TEE_InvokeTACCommand take paramTypes and params as arguments. The calling Trusted Application can use these arguments to pass up to four parameters.

Each of the parameters has a type, which is one of the TEE_PARAM_TYPE_XXX values listed in Table 4-1 on page 52. The content of paramTypes SHOULD be built using the macro TEE_PARAM_TYPES (see section 4.3.6.1).

Unless all parameter types are set to TEE_PARAM_TYPE_NONE, params SHALL NOT be NULL and SHALL point to an array of four TEE_Param elements. Each of the params[i] is interpreted as follows.

When the operation starts, the Framework reads the parameters as described in Table 4-15.

Table 4-15: Interpretation of params[i] on Entry to Internal Client API

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Interpretation of params[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PARAM_TYPE_NONE</td>
<td>Ignored.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_OUTPUT</td>
<td>Contains two integers in params[i].value.a and params[i].value.b.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INOUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_OUTPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INOUT</td>
<td></td>
</tr>
</tbody>
</table>

During the operation, the destination Trusted Application can update the contents of the OUTPUT or INOUT Memory References.

When the operation completes, the Framework updates the structure params[i] as described in Table 4-16.

Table 4-16: Effects of Internal Client API on params[i]

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Effects on params[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PARAM_TYPE_NONE</td>
<td>Unchanged.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INPUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_OUTPUT</td>
<td>params[i].value.a and params[i].value.b are updated with the value sent by the destination Trusted Application.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_VALUE_INOUT</td>
<td></td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_OUTPUT</td>
<td>params[i].memref.size is updated to reflect the actual or requested size of the buffer.</td>
</tr>
<tr>
<td>TEE_PARAM_TYPE_MEMREF_INOUT</td>
<td></td>
</tr>
</tbody>
</table>
The implementation SHALL enforce the following restrictions on TEE_PARAM_TYPE_MEMREF_XXX values:

- Where all or part of the referenced memory buffer was passed to the TA from the REE or from another TA, the implementation SHALL NOT result in downgrade of the security characteristics of the buffer – see Table 4-5.

- Where all or part of the referenced buffer was allocated by the TA with the TEE_MALLOC_NO_SHARE hint, the implementation SHALL raise a panic for the TA.
4.10 Cancellation Functions

This section defines functions for Trusted Applications to handle cancellation requested by a Client where a Client is either a REE Client Application or a TA.

When a Client requests cancellation using the function TEEC_RequestCancellation (in the case of an REE Client using the [Client API]) or a cancellation is created through a timeout (in the case of a TA Client), the implementation SHALL do the following:

- If the operation has not reached the TA yet but has been queued in the TEE, then it SHALL be retired from the queue and fail with the return code:
  - For an REE Client, TEEC_ERROR_CANCEL and the origin TEEC_ORIGIN_TEE;
  - For a TEE Client, TEE_ERROR_CANCEL and the origin TEE_ORIGIN_TEE.

- If the operation has been transmitted to the Trusted Application, the implementation SHALL set the Cancellation Flag of the task executing the command. If the Peripheral end Event API is present, a TEE_Event_ClientCancel event shall be inserted into the event queue by the session peripheral.

- If the Trusted Application has unmasked the effects of cancellation by using the function TEE_UnmaskCancellation, and if the task is engaged in a cancellable function when the Cancellation Flag is set, then that cancellable function is interrupted. The Trusted Application can detect that the function has been interrupted because it returns TEE_ERROR_CANCEL. It can then execute cleanup code and possibly fail the current client operation, although it may well report a success.
  - Note that this version of the specification defines the following cancellable functions: TEE_Wait and TEE_Event_Wait.
  - The functions TEE_OpenTASession and TEE_InvokeTACommand, while not cancellable per se, SHALL transmit cancellation requests: If the Cancellation Flag is set and the effects of cancellation are not masked, then the Trusted Core Framework SHALL consider that the cancellation of the corresponding operation is requested.

- When the Cancellation Flag is set for a given task, the function TEE_GetCancellationFlag SHALL return true, but only in the case the cancellations are not masked. This allows the Trusted Application to poll the Cancellation Flag, for example, when it is engaged in a lengthy active computation not using cancellable functions such as TEE_Wait.

4.10.1 TEE_GetCancellationFlag

Since: TEE Internal API v1.0

bool TEE_GetCancellationFlag( void );

Description

The TEE_GetCancellationFlag function determines whether the current task's Cancellation Flag is set. If cancellations are masked, this function SHALL return false. This function cannot panic.

Specification Number: 10  Function Number: 0x501

Return Value

- false if the Cancellation Flag is not set or if cancellations are masked
true if the Cancellation Flag is set and cancellations are not masked
4.10.2  TEE_UnmaskCancellation

Since: TEE Internal API v1.0

bool TEE_UnmaskCancellation( void );

Description

The TEE_UnmaskCancellation function unmasks the effects of cancellation for the current task.

When cancellation requests are unmasked, the Cancellation Flag interrupts cancellable functions such as TEE_Wait and requests the cancellation of operations started with TEE_OpenTASession or TEE_InvokeTACommand.

By default, tasks created to handle a TA entry point have cancellation masked, so that a TA does not have to cope with the effects of cancellation requests.

Specification Number: 10  Function Number: 0x503

Return Value

• true if cancellations were masked prior to calling this function
• false otherwise

Panic Reasons

• If the Implementation detects any error.

4.10.3  TEE_MaskCancellation

Since: TEE Internal API v1.0

bool TEE_MaskCancellation( void );

Description

The TEE_MaskCancellation function masks the effects of cancellation for the current task.

When cancellation requests are masked, the Cancellation Flag does not have an effect on the cancellable functions and cannot be retrieved using TEE_GetCancellationFlag.

By default, tasks created to handle a TA entry point have cancellation masked, so that a TA does not have to cope with the effects of cancellation requests.

Specification Number: 10  Function Number: 0x502

Return Value

• true if cancellations were masked prior to calling this function
• false otherwise

Panic Reasons

• If the Implementation detects any error.
4.11 Memory Management Functions

This section defines the following functions:

- A function to check the access rights of a given buffer. This can be used in particular to check if the buffer belongs to shared memory.
- Access to an instance data register, which provides a possibly more efficient alternative to using read-write C global variables
- A malloc facility
- A few utilities to copy and fill data blocks

4.11.1 TEE_CheckMemoryAccessRights

Since: TEE Internal API v1.2 – See Backward Compatibility note below.

```c
TEE_Result TEE_CheckMemoryAccessRights(
    uint32_t accessFlags,
    [inbuf] void*    buffer, size_t size);
```

Description

The `TEE_CheckMemoryAccessRights` function causes the Implementation to examine a buffer of memory specified in the parameters `buffer` and `size` and to determine whether the current Trusted Application instance has the access rights requested in the parameter `accessFlags`. If the characteristics of the buffer are compatible with `accessFlags`, then the function returns `TEE_SUCCESS`. Otherwise, it returns `TEE_ERROR_ACCESS_DENIED`. Note that the buffer SHOULD NOT be accessed by the function, but the Implementation SHOULD check the access rights based on the address of the buffer and internal memory management information.

The parameter `accessFlags` can contain one or more of the following flags:

- `TEE_MEMORY_ACCESS_READ`: Check that the buffer is entirely readable by the current Trusted Application instance.
- `TEE_MEMORY_ACCESS_WRITE`: Check that the buffer is entirely writable by the current Trusted Application instance.
- `TEE_MEMORY_ACCESS_ANY_OWNER`:
  - If this flag is not set, then the function checks that the buffer is not shared, i.e. whether it can be safely passed in an `[in]` or `[out]` parameter.
  - If this flag is set, then the function does not check ownership. It returns `TEE_SUCCESS` if the Trusted Application instance has read or write access to the buffer, independently of whether the buffer resides in memory owned by a Client or not.
- All other flags are reserved for future use and SHOULD be set to `0`.

The result of this function is valid until:

- The allocated memory area containing the supplied buffer is passed to `TEE_Realloc` or `TEE_Free`.
- One of the entry points of the Trusted Application returns.
- Actors outside of the TEE change the memory access rights when the memory is shared with an outside entity.
In the first two situations, the access rights of a given buffer MAY change and the Trusted Application SHOULD call the function TEE_CheckMemoryAccessRights again.

When this function returns TEE_SUCCESS, and as long as this result is still valid, the Implementation SHALL guarantee the following properties:

- For the flag TEE_MEMORY_ACCESS_READ and TEE_MEMORY_ACCESS_WRITE, the Implementation SHALL guarantee that subsequent read or write accesses by the Trusted Application wherever in the buffer will succeed and will not panic.

- When the flag TEE_MEMORY_ACCESS_ANY_OWNER is not set, the Implementation SHALL guarantee that the memory buffer is owned either by the Trusted Application instance or by a more trusted component, and cannot be controlled, modified, or observed by a less trusted component, such as the Client of the Trusted Application. This means that the Trusted Application can assume the following guarantees:
  - Read-after-read consistency: If the Trusted Application performs two successive read accesses to the buffer at the same address and if, between the two read accesses, it performs no write, either directly or indirectly through the API to that address, then the two reads SHALL return the same result.
  - Read-after-write consistency: If the Trusted Application writes some data in the buffer and subsequently reads the same address and if it performs no write, either directly or indirectly through the API to that address in between, the read SHALL return the data.
  - Non-observability: If the Trusted Application writes some data in the buffer, then the data SHALL NOT be observable by components less trusted than the Trusted Application itself.

Note that when true memory sharing is implemented between Clients and the Trusted Application, the Memory Reference Parameters passed to the TA entry points will typically not satisfy these requirements. In this case, the function TEE_CheckMemoryAccessRights SHALL return TEE_ERROR_ACCESS_DENIED. The code handling such buffers has to be especially careful to avoid security issues brought by this lack of guarantees.

For example, it can read each byte in the buffer only once and refrain from writing temporary data in the buffer.

Additionally, the Implementation SHALL guarantee that some types of memory blocks have a minimum set of access rights:

- The following blocks SHALL allow read and write accesses, SHALL be owned by the Trusted Application instance, and SHOULD NOT allow code execution:
  - All blocks returned by TEE_Malloc or TEE_Realloc
  - All the local and global non-const C variables
  - The TEE_Param structures passed to the entry points TA_OpenSessionEntryPoint and TA_InvokeCommandEntryPoint. This applies to the immediate contents of the TEE_Param structures, but not to the pointers contained in the fields of such structures, which can of course point to memory owned by the client. Note that this also means that these TEE_Param structures SHALL NOT directly point to the corresponding structures in the TEE Client API (see [Client API]) or the Internal Client API (see section 4.9). The Implementation SHALL perform a copy into a safe TA-owned memory buffer before passing the structures to the entry points.

- The following blocks SHALL allow read accesses, SHALL be owned by the Trusted Application instance, and SHOULD NOT allow code execution:
  - All const local or global C variables

- The following blocks MAY allow read accesses, SHALL be owned by the Trusted Application instance, and SHALL allow code execution:
  - The code of the Trusted Application itself
When a particular parameter passed in the structure `TEE_Param` to a TA entry point is a Memory Reference as specified in its parameter type, then this block, as described by the initial values of the fields `buffer` and `size` in that structure, SHALL allow read and/or write accesses as specified in the parameter type. As noted above, this buffer is not required to reside in memory owned by the TA instance.

Finally, any Implementation SHALL also guarantee that the NULL pointer cannot be dereferenced. If a Trusted Application attempts to read one byte at the address NULL, it SHALL panic. This guarantee SHALL extend to a segment of addresses starting at NULL, but the size of this segment is implementation-dependent.

**Parameters**

- `accessFlags`: The access flags to check
- `buffer`, `size`: The description of the buffer to check

**Specification Number:** 10  
**Function Number:** 0x601

**Return Code**

- `TEE_SUCCESS`: If the entire buffer allows the requested accesses
- `TEE_ERROR_ACCESS_DENIED`: If at least one byte in the buffer is not accessible with the requested accesses

**Panic Reasons**

`TEE_CheckMemoryAccessRights` SHALL NOT panic for any reason.

**Backward Compatibility**

Prior to TEE Internal Core API v1.2, `TEE_CheckMemoryAccessRights` did not specify the `[inbuf]` annotation on `buffer`.

TEE Internal Core API v1.1 used a different type for the `size`.

---

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
4.11.2 TEE_SetInstanceData

Since: TEE Internal API v1.0

```c
void TEE_SetInstanceData(
    [ctx] void* instanceData );
```

**Description**

The TEE_SetInstanceData and TEE_GetInstanceData functions provide an alternative to writable global data (writable variables with global scope and writable static variables with global or function scope). While an Implementation SHALL support C global variables, using these functions may be sometimes more efficient, especially if only a single instance data variable is required.

These two functions can be used to register and access an instance variable. Typically this instance variable can be used to hold a pointer to a Trusted Application-defined memory block containing any writable data that needs instance global scope, or writable static data that needs instance function scope.

The value of this pointer is not interpreted by the Framework, and is simply passed back to other TA_ functions within this session. Note that *instanceData may be set with a pointer to a buffer allocated by the Trusted Application instance or with anything else, such as an integer, a handle, etc. The Framework will not automatically free *instanceData when the session is closed; the Trusted Application instance is responsible for freeing memory if required.

An equivalent session context variable for managing session global and static data exists for sessions (see TA_OpenSessionEntryPoint, TA_InvokeCommandEntryPoint, and TA_CloseSessionEntryPoint in section 4.3).

This function sets the Trusted Application instance data pointer. The data pointer can then be retrieved by the Trusted Application instance by calling the TEE_GetInstanceData function.

**Parameters**

- **instanceData**: A pointer to the global Trusted Application instance data. This pointer may be NULL.

**Specification Number**: 10  **Function Number**: 0x609

**Panic Reasons**

- If the Implementation detects any error.
4.11.3 TEE_GetInstanceData

Since: TEE Internal API v1.0

```c
[ctx] void* TEE_GetInstanceData(void);
```

Description

The `TEE_GetInstanceData` function retrieves the instance data pointer set by the Trusted Application using the `TEE_SetInstanceData` function.

Specification Number: 10  Function Number: 0x603

Return Value

The value returned is the previously set pointer to the Trusted Application instance data, or `NULL` if no instance data pointer has yet been set.

Panic Reasons

- If the Implementation detects any error.
4.11.4 TEEMalloc

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void* TEEMalloc(
    size_t   size,
    uint32_t hint);
```

Description

The `TEEMalloc` function allocates space for an object whose size in bytes is specified in the parameter `size`.

The pointer returned is guaranteed to be aligned such that it may be assigned as a pointer to any basic C type.

The parameter `hint` is a hint to the allocator. The valid values for the hint are defined in Table 4-17. The valid hint values are a bitmask and can be independently set. This parameter allows Trusted Applications to refer to various pools of memory or to request special characteristics for the allocated memory by using an implementation-defined hint. Future versions of this specification may introduce additional standard hints.

The hint values should be treated as a mask – they can be logically 'or'd together. In Table 4-17, when 'x' appears in a field it means that the value of the bit or bits can be 1 or 0. When 'y' appears in a field it means that the value of that bit or bits is irrelevant to the definition of that row, UNLESS already defined in a previous row, and can be either 1 or 0.

***Table 4-17: Valid Hint Values***

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_MALLOC_FILL_ZERO</td>
<td>0 x x 0</td>
<td>Memory block returned SHALL be filled with zeros. Note: TEE_MALLOC_NO_FILL has precedence over TEE_MALLOC_FILL_ZERO.</td>
</tr>
<tr>
<td>TEE_MALLOC_NO_FILL</td>
<td>0 x x 1</td>
<td>Memory block returned may not be filled with zeros</td>
</tr>
<tr>
<td>TEE_MALLOC_NO_SHARE</td>
<td>0 x 1 x</td>
<td>The returned block of memory will not be shared with other TA instances.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0 y</td>
<td>Reserved for future versions of this specification.</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>1 y</td>
<td>Reserved for implementation-defined hints.</td>
</tr>
</tbody>
</table>
The hint SHALL be attached to the allocated block and SHALL be used when the block is reallocated with TEE_Realloc.

If the space cannot be allocated, given the current hint value (for example because the hint value is not implemented), a NULL pointer SHALL be returned.

TEE_MALLOC_NO_SHARE provides a mechanism for a TA developer to indicate that the allocation request is not to be shared with other TAs. Implementations MAY choose to use this hint to allocate memory from memory pools which are optimized for performance at the expense of sharing.

TEE_MALLOC_NO_FILL provides a mechanism to allow a TA to indicate that it does not assume that memory will be zero filled. It SHALL be used in conjunction with TEE_MALLOC_NO_SHARE.

A Trusted OS MAY use the TEE_MALLOC_NO_FILL hint to avoid clearing memory on allocation where it is safe to do so. When allocating to a TA, a Trusted OS SHALL zero fill memory which:

- Has previously been allocated to another TA instance;
- Has previously been allocated to internal structures of the TEE.
- Does not have the TEE_MALLOC_NO_SHARE hint.

Parameters

- size: The size of the buffer to be allocated.
- hint: A hint to the allocator. See Table 4-17 for valid values.

Specification Number: 10 Function Number: 0x604

Return Value

Upon successful completion, with size not equal to zero, the function returns a pointer to the allocated space. If the space cannot be allocated, given the current hint value, a NULL pointer is returned.

If the size of the requested space is zero:

- The value returned is undefined but guaranteed to be different from NULL. This non-NULL value ensures that the hint can be associated with the returned pointer for use by TEE_Realloc.
- The Trusted Application SHALL NOT access the returned pointer. The Trusted Application SHOULD panic if the memory pointed to by such a pointer is accessed for either read or write.

Panic Reasons

- If the Implementation detects any error.
- If TEE_MALLOC_NO_FILL is used without TEE_MALLOC_NO_SHARE.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the size.

TEE_MALLOC_NO_SHARE and TEE_MALLOC_NO_FILL were added in TEE Internal Core API v1.2.
4.11.5 TEE_Realloc

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
void* TEE_Realloc(
    [inout] void* buffer,
    size_t newSize);
```

Description

The `TEE_Realloc` function changes the size of the memory object pointed to by `buffer` to the size specified by `newSize`. The content of the object remains unchanged up to the lesser of the new and old sizes. Space in excess of the old size contains unspecified content.

If the new size of the memory object requires movement of the object, the space for the previous instantiation of the object is deallocated. If the space cannot be allocated, the original object remains allocated, and this function returns a `NULL` pointer.

If `buffer` is `NULL`, `TEE_Realloc` is equivalent to `TEE_Malloc` for the specified size. The associated hint applied SHALL be the default value defined in `TEE_Malloc`.

It is a Programmer Error if `buffer` does not match a pointer previously returned by `TEE_Malloc` or `TEE_Realloc`, or if the space has previously been deallocated by a call to `TEE_Free` or `TEE_Realloc`.

If the hint initially provided when the block was allocated with `TEE_Malloc` is 0, then the extended space is filled with zeroes. In general, the function `TEE_Realloc` SHOULD allocate the new memory buffer using exactly the same hint as for the buffer initially allocated with `TEE_Malloc`. In any case, it SHALL NOT downgrade the security or performance characteristics of the buffer.

Note that any pointer returned by `TEE_Malloc` or `TEE_Realloc` and not yet freed or reallocated can be passed to `TEE_Realloc`. This includes the special non-NULL pointer returned when an allocation for 0 bytes is requested.

Parameters

- `buffer`: The pointer to the object to be reallocated
- `newSize`: The new size required for the object

Specification Number: 10   Function Number: 0x608

Return Value

Upon successful completion, `TEE_Realloc` returns a pointer to the (possibly moved) allocated space.

If there is not enough available memory, `TEE_Realloc` returns a `NULL` pointer and the original buffer is still allocated and unchanged.

Panic Reasons

- If the Implementation detects any error.

Backward Compatibility

Since: TEE Internal API v1.0
Versions of TEE_Realloc prior to TEE Internal API v1.2 used the \texttt{[in]} annotation for buffer.

Versions of TEE_Realloc prior to TEE Internal Core API v1.2 used a \texttt{uint32_t} type for the size parameter.

On a Trusted OS with natural word length greater than 32 bits this leads to operation limitations, and the size parameter was changed to a \texttt{size_t}.
4.11.6  TEE_Free

Since: TEE Internal API v1.0

void TEE_Free(void *buffer);

Description

The TEE_Free function causes the space pointed to by buffer to be deallocated; that is, made available for further allocation.

If buffer is a NULL pointer, TEE_Free does nothing. Otherwise, it is a Programmer Error if the argument does not match a pointer previously returned by the TEE_Malloc or TEE_Realloc if the space has been deallocated by a call to TEE_Free or TEE_Realloc.

Parameters

- buffer: The pointer to the memory block to be freed

Specification Number: 10  Function Number: 0x602

Panic Reasons

- If the Implementation detects any error.
4.11.7 TEE_MemMove

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
void TEE_MemMove(
    [outbuf(size)] void* dest,
    [inbuf(size)] void* src,
    size_t size);
```

Description

The TEE_MemMove function copies `size` bytes from the buffer pointed to by `src` into the buffer pointed to by `dest`.

Copying takes place as if the `size` bytes from the buffer pointed to by `src` are first copied into a temporary array of `size` bytes that does not overlap the buffers pointed to by `dest` and `src`, and then the `size` bytes from the temporary array are copied into the buffer pointed to by `dest`.

Parameters

- `dest`: A pointer to the destination buffer
- `src`: A pointer to the source buffer
- `size`: The number of bytes to be copied

Specification Number: 10    Function Number: 0x607

Panic Reasons

- If the implementation detects any error.

Backward Compatibility

Before: TEE Internal Core API v1.2

Versions of TEE_MemMove prior to TEE Internal Core API v1.2 used a `uint32_t` type for the size parameter. On a trusted OS with natural word length greater than 32 bits this leads to operation limitations, and the size parameter was changed to a `size_t`.

A backward compatible version of TEE_MemMove can be selected at compile time if the version compatibility definitions (see section 3.5.1) indicate that compatibility with a version of this specification before v1.2 is required.

```c
int32_t TEE_MemMove(
    [inbuf(size)] void* buffer1,
    [inbuf(size)] void* buffer2,
    uint32_t size);
```
4.11.8 TEE_MemCompare

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
int32_t TEE_MemCompare(
    [inbuf(size)] void* buffer1,
    [inbuf(size)] void* buffer2,
    size_t size);
```

Description

The TEE_MemCompare function compares the first `size` bytes of the buffer pointed to by `buffer1` to the first `size` bytes of the buffer pointed to by `buffer2`.

Parameters

- `buffer1`: A pointer to the first buffer
- `buffer2`: A pointer to the second buffer
- `size`: The number of bytes to be compared

Specification Number: 10  Function Number: 0x605

Return Value

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `uint8_t`) that differ in the objects being compared.

- If the first byte that differs is higher in `buffer1`, then return an integer greater than zero.
- If the first `size` bytes of the two buffers are identical, then return zero.
- If the first byte that differs is higher in `buffer2`, then return an integer lower than zero.

Panic Reasons

- If the Implementation detects any error.

Backward Compatibility

Before: TEE Internal Core API v1.2

Versions of TEE_MemCompare prior to TEE Internal Core API v1.2 used a `uint32_t` type for the `size` parameter. On a Trusted OS with natural word length greater than 32 bits this leads to operation limitations, and the `size` parameter was changed to a `size_t`.

A backward compatible version of TEE_MemCompare can be selected at compile time if the version compatibility definitions (see section 3.5.1) indicate that compatibility with a version of this specification before v1.2 is required.

```c
int32_t TEE_MemCompare(
    [inbuf(size)] void* buffer1,
    [inbuf(size)] void* buffer2,
    uint32_t size);
```
4.11.9  TEE_MemFill

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
void TEE_MemFill(
    [outbuf(size)] void* buffer,
    uint8_t x,
    size_t size);
```

Description

The TEE_MemFill function writes the byte `x` into the first `size` bytes of the buffer pointed to by `buffer`.

Parameters

- `buffer`: A pointer to the destination buffer
- `x`: The value to be set
- `size`: The number of bytes to be set

Specification Number: 10  Function Number: 0x606

Panic Reasons

- If the Implementation detects any error.

Backward Compatibility

Before: TEE Internal Core API v1.2

In versions of this specification prior to TEE Internal Core API v1.2, TEE_MemFill used `uint32_t` type for the `x` and `size` parameters. The previous definition of `x` explicitly required coercion to a byte type, so this has been made explicit. Using `uint32_t` for a size parameter can lead limitations on some platforms, and the size parameter was changed to a `size_t`.

A backward compatible version of TEE_MemFill can be selected at compile time if the version compatibility definitions (see section 3.5.1) indicate that compatibility with a version of this specification before v1.2 is required.

```c
void TEE_MemFill(
    [outbuf(size)] void* buffer,
    uint32_t x,
    uint32_t size);
```
5 Trusted Storage API for Data and Keys

This chapter includes the following sections:

5.1 Summary of Features and Design

This section provides a summary of the features and design of the Trusted Storage API.

- Each TA has access to a set of Trusted Storage Spaces, identified by 32-bit Storage Identifiers.
  - This specification defines a single Trusted Storage Space for each TA, which is its own private storage space. The identifier for this storage space is TEE_STORAGE_PRIVATE.
  - Unless explicitly overridden by other specifications, the objects in any Trusted Storage Space are accessible only to the TA that created them and SHALL NOT be visible to other TEE entities except those associated directly with implementing the Trusted Storage System.
  - Other storage identifiers may be defined in future versions of this specification or by an Implementation, e.g. to refer to storage spaces shared among multiple TAs or for communicating between boot-time entities and run-time Trusted Applications.

- A Trusted Storage Space contains Persistent Objects. Each persistent object is identified by an Object Identifier, which is a variable-length binary buffer from 0 to 64 bytes. Object identifiers can contain any bytes, including bytes corresponding to non-printable characters.

- A persistent object can be a Cryptographic Key Object, a Cryptographic Key-Pair Object, or a Data Object.

- Each persistent object has a type, which precisely defines the content of the object. For example, there are object types for AES keys, RSA key-pairs, data objects, etc.

- All persistent objects have an associated Data Stream. Persistent data objects have only a data stream. Persistent cryptographic objects (that is, keys or key-pairs) have a data stream, Object Attributes, and metadata.
  - The Data Stream is entirely managed in the TA memory space. It can be loaded into a TA-allocated buffer when the object is opened or stored from a TA-allocated buffer when the object is created. It can also be accessed as a stream, so it can be used to store large amounts of data accessed by small chunks.
  - Object Attributes are used for small amounts of data (typically a few tens or hundreds of bytes). They can be stored in a memory pool that is separated from the TA instance and some attributes may be hidden from the TA itself. Attributes are used to store the key material in a structured way. For example, an RSA key-pair has an attribute for the modulus, the public exponent, the private exponent, etc. When an object is created, all mandatory Object Attributes SHALL be specified and optional attributes MAY be specified.
Note that an Implementation is allowed to store more information in an object than the visible attributes. For example, some data might be pre-computed and stored internally to accelerate subsequent cryptographic operations.

- The metadata associated with each cryptographic object includes:

  - **Key Size** in bits. The precise meaning depends on the key algorithm. For example, AES key can have 128 bits, 192 bits, or 256 bits; RSA keys can have 1024 bits or 2048 bits or any other supported size, etc.

  - **Key Usage Flags**, which define the operations permitted with the key as well as whether the sensitive parts of the key material can be retrieved by the TA or not.

- A TA can also allocate **Transient Objects**. Compared to persistent objects:

  - Transient objects are held in memory and are automatically wiped and reclaimed when they are closed or when the TA instance is destroyed.

  - Transient objects contain only attributes and no data stream.

  - A transient object can be **uninitialized**, in which case it is an object container allocated with a certain object type and maximum size but with no attributes. A transient object becomes **initialized** when its attributes are populated. Note that persistent objects are always created initialized. This means that when the TA wants to generate or derive a persistent key, it has to first use a transient object then write the attributes of a transient object into a persistent object.

  - Transient objects have no identifier, they are only manipulated through object handles.

  - Currently, transient objects are used for cryptographic keys and key-pairs.

- Any function that accesses a persistent object handle MAY return a status of TEE_ERROR_CORRUPT_OBJECT or TEE_ERROR_CORRUPT_OBJECT_2, which indicates that corruption of the object has been detected. Before this status is returned, the Implementation SHALL delete the corrupt object and SHALL close the associated handle; subsequent use of the handle SHALL cause a panic.

- Any function that accesses a persistent object MAY return a status of TEE_ERROR_STORAGE_NOT_AVAILABLE or TEE_ERROR_STORAGE_NOT_AVAILABLE_2, which indicates that the storage system in which the object is stored is not accessible for some reason.

- Persistent and transient objects are manipulated through opaque **Object Handles**.

  - Some functions accept both types of object handles. For example, a cryptographic operation can be started with either a transient key handle or a persistent key handle.

  - Some functions accept only handles on transient objects. For example, populating the attributes of an object works only with a transient object because it requires an uninitialized object and persistent objects are always fully initialized.

  - Finally, the file-like API functions to access the data stream work only with persistent objects because transient objects have no data stream.

Cryptographic operations are described in Chapter 6.
5.2 Trusted Storage and Rollback Detection

The Trusted Storage SHALL provide a minimum level of protection against rollback attacks on persistent objects; however it is accepted that the actually physical storage may be in an unsecure area and so is vulnerable to actions from outside of the TEE.

The level of protection that a Trusted Application can assume from the rollback detection mechanism of the Trusted Storage is implementation defined but can be discovered programmatically by querying the implementation property:

gpd.tee.trustedStorage.rollbackDetection.protectionLevel

Since: TEE Internal API v1.1

Table 5-1: Values of gpd.tee.trustedStorage.rollbackDetection.protectionLevel

<table>
<thead>
<tr>
<th>Property Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Rollback detection mechanism for the Trusted Storage is enforced at the REE level.</td>
</tr>
<tr>
<td>1000</td>
<td>Rollback detection mechanism for the Trusted Storage is based on TEE-controlled hardware. This hardware SHALL be out of reach of software attacks from the REE. Users may still be able to roll back the Trusted Storage but this SHALL be detected by the Implementation.</td>
</tr>
<tr>
<td></td>
<td>All other values are reserved.</td>
</tr>
</tbody>
</table>
5.3 Data Types

5.3.1 TEE_Attribute

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

An array of this type is passed whenever a set of attributes is specified as argument to a function of the API.

```c
typedef struct {
    uint32_t attributeID;
    union {
        struct {
            [inbuf] void* buffer; size_t length;
        } ref;
        struct {
            uint32_t a;
            uint32_t b;
        } value;
    } content;
} TEE_Attribute;
```

An attribute can be either a buffer attribute or a value attribute. This is determined by bit \[29\] of the attribute identifier. If this bit is set to 0, then the attribute is a buffer attribute and the field ref SHALL be selected. If the bit is set to 1, then it is a value attribute and the field value SHALL be selected.

When an array of attributes is passed to a function, either to populate an object or to specify operation parameters, and if an attribute identifier is present twice in the array, then only the first occurrence is used.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the length.

5.3.2 TEE_ObjectInfo

Since: TEE Internal API v1.0

```c
typedef struct {
    uint32_t objectType;
    uint32_t objectSize;
    uint32_t maxObjectSize;
    uint32_t objectUsage;
    uint32_t dataSize;
    uint32_t dataPosition;
    uint32_t handleFlags;
} TEE_ObjectInfo;
```

See the documentation of function TEE_GetObjectInfo1 in section 5.5.1 for a description of this structure.
5.3.3 TEE_Whence

Since: TEE Internal API v1.0, redefined v1.2 – See Backward Compatibility note below.

```c
typedef uint32_t TEE_Whence;
```

This structure indicates the possible start offset when moving a data position in the data stream associated with a persistent object. The following table lists the legal values for TEE_Whence. All other values are reserved.

**Table 5-1b: TEE_Whence Constants**

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_DATA_SEEK_SET</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_DATA_SEEK_CUR</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_DATA_SEEK_END</td>
<td>0x00000002</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x00000003 – 0x7FFFFFFE</td>
</tr>
<tr>
<td>TEE_WHENCE_ILLEGAL_VALUE</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>0x80000000 – 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

Note: TEE_WHENCE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.

Backward Compatibility

Prior to TEE Internal Core API v1.2, TEE_Whence was defined as an int.

5.3.4 TEE_ObjectHandle

Since: TEE Internal API v1.0

```c
typedef struct __TEE_ObjectHandle* TEE_ObjectHandle;
```

TEE_ObjectHandle is an opaque handle on an object. These handles are returned by the functions TEE_AllocateTransientObject (section 5.6.1), TEE_OpenPersistentObject (section 5.7.1), and TEE_CreatePersistentObject (section 5.7.2).

5.3.5 TEE_ObjectEnumHandle

Since: TEE Internal API v1.0

```c
typedef struct __TEE_ObjectEnumHandle* TEE_ObjectEnumHandle;
```

TEE_ObjectEnumHandle is an opaque handle on an object enumerator. These handles are returned by the function TEE_AllocatePersistentObjectEnumerator specified in section 5.8.1.
5.4 Constants

5.4.1 Constants Used in Trusted Storage API for Data and Keys

The following tables pertain to the Trusted Storage API for Data and Keys (Chapter 5).

Table 5-2: Object Storage Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_STORAGE_PRIVATE</td>
<td>0x00000001</td>
</tr>
<tr>
<td>Reserved for future use</td>
<td>0x00000002-0x7FFFFFFE</td>
</tr>
<tr>
<td>TEE_STORAGE_ILLEGAL_VALUE</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>Reserved for implementation defined storage</td>
<td>0x80000000-0x7FFFFFFF</td>
</tr>
</tbody>
</table>

Note: TEE_STORAGE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.

Table 5-3: Data Flag Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_DATA_FLAG_ACCESS_READ</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_ACCESS_WRITE</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_ACCESS_WRITE_META</td>
<td>0x00000004</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_SHARE_READ</td>
<td>0x00000010</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_SHARE_WRITE</td>
<td>0x00000020</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_OVERWRITE</td>
<td>0x00000040</td>
</tr>
<tr>
<td>TEE_DATA_FLAG_EXCLUSIVE</td>
<td>0x00000040</td>
</tr>
</tbody>
</table>

Table 5-4: Usage Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_USAGE_EXTRACTABLE</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_USAGE_ENCRYPT</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_USAGE_DECRYPT</td>
<td>0x00000004</td>
</tr>
<tr>
<td>TEE_USAGE_MAC</td>
<td>0x00000008</td>
</tr>
<tr>
<td>TEE_USAGE_SIGN</td>
<td>0x00000010</td>
</tr>
<tr>
<td>TEE_USAGE_VERIFY</td>
<td>0x00000020</td>
</tr>
<tr>
<td>TEE_USAGE_DERIVE</td>
<td>0x00000040</td>
</tr>
</tbody>
</table>
### 5.4.2 Constants Used in Cryptographic Operations API

The following tables pertain to the Cryptographic Operations API (Chapter 6).

#### Table 5-5: Handle Flag Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_HANDLE_FLAG_PERSISTENT</td>
<td>0x00010000</td>
</tr>
<tr>
<td>TEE_HANDLE_FLAG_INITIALIZED</td>
<td>0x00020000</td>
</tr>
<tr>
<td>TEE_HANDLE_FLAG_KEY_SET</td>
<td>0x00040000</td>
</tr>
<tr>
<td>TEE_HANDLE_FLAG_EXPECT_TWO_KEYS</td>
<td>0x00080000</td>
</tr>
</tbody>
</table>

#### Table 5-6: Operation Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_OPERATION_CIPHER</td>
<td>1</td>
</tr>
<tr>
<td>TEE_OPERATION_MAC</td>
<td>3</td>
</tr>
<tr>
<td>TEE_OPERATION_AE</td>
<td>4</td>
</tr>
<tr>
<td>TEE_OPERATION_DIGEST</td>
<td>5</td>
</tr>
<tr>
<td>TEE_OPERATIONASYMETRIC_CIPHER</td>
<td>6</td>
</tr>
<tr>
<td>TEE_OPERATIONASYMETRIC_SIGNATURE</td>
<td>7</td>
</tr>
<tr>
<td>TEE_OPERATION_KEY_DERIVATION</td>
<td>8</td>
</tr>
<tr>
<td>Reserved for future use</td>
<td>0x00000009-0x7FFFFFFF</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>0x08000000-0xFFFFFFFF</td>
</tr>
</tbody>
</table>

#### Table 5-7: Operation States

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_OPERATION_STATE_INITIAL</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_OPERATION_STATE_ACTIVE</td>
<td>0x00000001</td>
</tr>
<tr>
<td>Reserved for future use</td>
<td>0x00000002-0x7FFFFFFF</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>0x08000000-0xFFFFFFFF</td>
</tr>
</tbody>
</table>

#### Table 5-8: Miscellaneous Constants [formerly Table 5-8]

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_DATA_MAX_POSITION</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>TEE_OBJECT_ID_MAX_LEN</td>
<td>64</td>
</tr>
</tbody>
</table>
5.5 Generic Object Functions

These functions can be called on both transient and persistent object handles.

5.5.1 TEE_GetObjectInfo1

Since: TEE Internal Core API v1.1

```
TEE_Result TEE_GetObjectInfo1(
    TEE_ObjectHandle   object,
    [out] TEE_ObjectInfo* objectInfo);
```

Description

This function replaces the TEE_GetObjectInfo function, whose use is deprecated.

The TEE_GetObjectInfo1 function returns the characteristics of an object. It fills in the following fields in the structure TEE_ObjectInfo (section 5.3.2):

- **objectType**: The parameter `objectType` passed when the object was created
- **objectSize**: The current size in bits of the object as determined by its attributes. This will always be less than or equal to `maxObjectSize`. Set to 0 for uninitialized and data only objects.
- **maxObjectSize**: The maximum `objectSize` which this object can represent.
  - For a persistent object, set to `objectSize`
  - For a transient object, set to the parameter `maxObjectSize` passed to TEE_AllocateTransientObject
- **objectUsage**: A bit vector of the TEE_USAGE_XXX bits defined in Table 5-4.
- **dataSize**
  - For a persistent object, set to the current size of the data associated with the object
  - For a transient object, always set to 0
- **dataPosition**
  - For a persistent object, set to the current position in the data for this handle. Data positions for different handles on the same object may differ.
  - For a transient object, set to 0
- **handleFlags**: A bit vector containing one or more of the following flags:
  - TEE_HANDLE_FLAG_PERSISTENT: Set for a persistent object
  - TEE_HANDLE_FLAG_INITIALIZED
    - For a persistent object, always set
    - For a transient object, initially cleared, then set when the object becomes initialized
  - TEE_DATA_FLAG_XXX: Only for persistent objects, the flags used to open or create the object

Parameters

- **object**: Handle of the object
- **objectInfo**: Pointer to a structure filled with the object information
Specification Number: 10  Function Number: 0x706

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_CORRUPT_OBJECT: If the persistent object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If object is not a valid opened object handle.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.5.2 TEE_RestrictObjectUsage1

Since: TEE Internal Core API v1.1

```c
TEE_Result TEE_RestrictObjectUsage1(
    TEE_ObjectHandle object,
    uint32_t          objectUsage );
```

Description

This function replaces the TEE_RestrictObjectInfo function, whose use is deprecated.

The TEE_RestrictObjectUsage1 function restricts the object usage flags of an object handle to contain at most the flags passed in the objectUsage parameter.

For each bit in the parameter objectUsage:

- If the bit is set to 1, the corresponding usage flag in the object is left unchanged.
- If the bit is set to 0, the corresponding usage flag in the object is cleared.

For example, if the usage flags of the object are set to TEE_USAGE_ENCRYPT | TEE_USAGE_DECRYPT and if objectUsage is set to TEE_USAGE_ENCRYPT | TEE_USAGE_EXTRACTABLE, then the only remaining usage flag in the object after calling the function TEE_RestrictObjectUsage1 is TEE_USAGE_ENCRYPT.

Note that an object usage flag can only be cleared. Once it is cleared, it cannot be set to 1 again on a persistent object.

A transient object’s object usage flags are reset to 1 using the TEE_ResetTransientObject function.

For a persistent object, setting the object usage SHALL be an atomic operation.

Parameters

- object: Handle on an object
- objectUsage: New object usage, an OR combination of one or more of the TEE_USAGE_XXX constants defined in Table 5-4

Specification Number: 10 Function Number: 0x707

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_CORRUPT_OBJECT: If the persistent object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If object is not a valid opened object handle.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.5.3 TEE_GetObjectBufferAttribute

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_GetObjectBufferAttribute(
    TEE_ObjectHandle object,
    uint32_t attributeID,
    [outbuf] void* buffer, size_t* size );
```

**Description**

The TEE_GetObjectBufferAttribute function extracts one buffer attribute from an object. The attribute is identified by the argument `attributeID`. The precise meaning of this parameter depends on the container type and size and is defined in section 6.1.1.

Bit [29] of the attribute identifier SHALL be set to 0; i.e. it SHALL denote a buffer attribute.

There are two kinds of object attributes, which are identified by a bit in their handle value (see Table 6-17):

- Public object attributes can always be extracted whatever the status of the container.
- Protected attributes can be extracted only if the object’s key usage contains the `TEE_USAGE_EXTRACTABLE` flag.

See section 6.1.1 for a definition of all available object attributes, their formats, and their level of protection.

Note: It is recommended that TA writers do not rely on implementations stripping leading zeros from bignum attributes. However, calling TEE_GetObjectBufferAttribute with a NULL buffer is guaranteed to return a size sufficient to hold the attribute.

**Parameters**

- `object`: Handle of the object
- `attributeID`: Identifier of the attribute to retrieve
- `buffer`, `size`: Output buffer to get the content of the attribute

**Specification Number:** 10  **Function Number:** 0x702

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the attribute is not found on this object
- `TEE_ERROR_SHORT_BUFFER`: If `buffer` is NULL or too small to contain the key part
- `TEE_ERROR_CORRUPT_OBJECT`: If the persistent object is corrupt. The object handle is closed.
- `TEE_ERROR_STORAGE_NOT_AVAILABLE`: If the persistent object is stored in a storage area which is currently inaccessible.

**Panic Reasons**

- If `object` is not a valid opened object handle.
- If the object is not initialized.
- If Bit [29] of `attributeID` is not set to 0, so the attribute is not a buffer attribute.
• If Bit [28] of `attributeID` is set to 0, denoting a protected attribute, and the object usage does not contain the `TEE_USAG_EXTRACTABLE` flag.

• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `size`. 
5.5.4 TEE_GetObjectValueAttribute

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetObjectValueAttribute(
    TEE_ObjectHandle object,
    uint32_t attributeID,
    [outopt] uint32_t* a,
    [outopt] uint32_t* b);
```

Description

The TEE_GetObjectValueAttribute function extracts a value attribute from an object. The attribute is identified by the argument attributeID. The precise meaning of this parameter depends on the container type and size and is defined in section 6.1.1. Bit [29] of the attribute identifier SHALL be set to 1, i.e. it SHALL denote a value attribute.

They are two kinds of object attributes, which are identified by a bit in their handle value (see Table 6-17):

- Public object attributes can always be extracted whatever the status of the container.
- Protected attributes can be extracted only if the object’s key usage contains the TEE_USAGE_EXTRACTABLE flag.

See section 6.1.1 for a definition of all available object attributes and their level of protection.

Where the format of the attribute (see Table 6-16) does not define a meaning for b, the value returned for b is implementation defined.

Parameters

- object: Handle of the object
- attributeID: Identifier of the attribute to retrieve
- a, b: Pointers on the placeholders filled with the attribute fields a and b. Each can be NULL if the corresponding field is not of interest to the caller.

Specification Number: 10 Function Number: 0x704

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If the attribute is not found on this object
- TEE_ERROR_ACCESS_DENIED: Deprecated: Handled by a panic
- TEE_ERROR_CORRUPT_OBJECT: If the persistent object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If object is not a valid opened object handle.
- If the object is not initialized.
- If Bit [29] of attributeID is not set to 1, so the attribute is not a value attribute.
• If Bit [28] of attributeID is set to 0, denoting a protected attribute, and the object usage does not contain the TEE_USAGE_EXTRACTABLE flag.
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

5.5.5 TEE_CloseObject

Since: TEE Internal API v1.0

void TEE_CloseObject( TEE_ObjectHandle object );

Description
The TEE_CloseObject function closes an opened object handle. The object can be persistent or transient. For transient objects, TEE_CloseObject is equivalent to TEE_FreeTransientObject.

This function will operate correctly even if the object or the containing storage is corrupt.

Parameters
• object: Handle on the object to close. If set to TEE_HANDLE_NULL, does nothing.

Specification Number: 10 Function Number: 0x701

Panic Reasons
• If object is not a valid opened object handle and is not equal to TEE_HANDLE_NULL.
• If the Implementation detects any other error.
5.6 Transient Object Functions

5.6.1 TEE_AllocateTransientObject

Since: TEE Internal API v1.0

```c
TEE_Result TEE_AllocateTransientObject(
    uint32_t           objectType,
    uint32_t           maxObjectSize,
    [out] TEE_ObjectHandle*  object );
```

Description

The TEE_AllocateTransientObject function allocates an uninitialized transient object, i.e. a container for attributes. Transient objects are used to hold a cryptographic object (key or key-pair).

The object type SHALL be specified. The maximum key size SHALL also be specified with all of the object types defined in Table 5-9.

The value TEE_KEYSIZE_NO_KEY SHOULD be used for maxObjectSize for object types that do not require a key so that all the container resources can be pre-allocated. A Trusted OS SHALL treat object types which are not defined in Table 5-9 as though they require TEE_KEYSIZE_NO_KEY for backward compatibility reasons.

As allocated, the container is uninitialized. It can be initialized by subsequently importing the object material, generating an object, deriving an object, or loading an object from the Trusted Storage.

The initial value of the key usage associated with the container is 0xFFFFFFFF, which means that it contains all usage flags. You can use the function TEE_RestrictObjectUsage1 to restrict the usage of the container.

The returned handle is used to refer to the newly-created container in all subsequent functions that require an object container: key management and operation functions. The handle remains valid until the container is deallocated using the function TEE_FreeTransientObject.

As shown in Table 5-9, the object type determines the possible object size to be passed to TEE_AllocateTransientObject, which is not necessarily the size of the object to allocate. In particular, for key objects the size to be passed is one of the appropriate key sizes described in Table 5-9.

Note that a compliant Implementation SHALL implement all the keys, algorithms, and key sizes described in Table 5-9 except the elliptic curve cryptographic types which are optional; support for other sizes or algorithms is implementation-defined.
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Possible Key Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_AES</td>
<td>128, 192, or 256 bits</td>
</tr>
<tr>
<td>TEE_TYPE_DES</td>
<td>Always 64 bits including the parity bits. This gives an effective key size of 56 bits</td>
</tr>
<tr>
<td>TEE_TYPE_DES3</td>
<td>128 or 192 bits including the parity bits. This gives effective key sizes of 112 or 168 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_MD5</td>
<td>Between 64 and 512 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA1</td>
<td>Between 80 and 512 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA224</td>
<td>Between 112 and 512 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA256</td>
<td>Between 192 and 1024 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA384</td>
<td>Between 256 and 1024 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA512</td>
<td>Between 256 and 1024 bits, multiple of 8 bits</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_PUBLIC_KEY</td>
<td>The number of bits in the modulus. 256, 512, 768, 1024, 1536 and 2048 bit keys SHALL be supported. Support for other key sizes including bigger key sizes is implementation-dependent. Minimum key size is 256 bits.</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_KEYPAIR</td>
<td>Same as for RSA public key size.</td>
</tr>
</tbody>
</table>
| TEE_TYPE_DSA_PUBLIC_KEY    | Depends on Algorithm:  
EE_ALG_DSA_SHA1: Between 512 and 1024 bits, multiple of 64 bits  
EE_ALG_DSA_SHA224: 2048 bits  
EE_ALG_DSA_SHA256: 2048 or 3072 bits |
| TEE_TYPE_DSA_KEYPAIR       | Same as for DSA public key size.                      |
| TEE_TYPE_DH_KEYPAIR        | From 256 to 2048 bits, multiple of 8 bits.            |
| TEE_TYPE_ECDSA_PUBLIC_KEY  | Between 160 and 521 bits. Conditional: Available only if at least one of the ECC the curves defined in Table 6-14 with "generic" equal to "Y" is supported. |
| TEE_TYPE_ECDSA_KEYPAIR     | Between 160 and 521 bits. Conditional: Available only if at least one of the ECC curves defined in Table 6-14 with "generic" equal to "Y" is supported. SHALL be same value as for ECDSA public key size (for values, see Table 6-14). |
| TEE_TYPE_ECDH_PUBLIC_KEY   | Between 160 and 521 bits. Conditional: Available only if at least one of the ECC curves defined in Table 6-14 with "generic" equal to "Y" is supported. |

3 WARNING: Given the increases in computing power, it is necessary to increase the strength of encryption used with time. Many of the algorithms and key sizes included are known to be weak and are included to support legacy implementations only. TA designers should regularly review the choice of cryptographic primitives and key sizes used in their applications and should refer to appropriate Government guidelines.
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Possible Key Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_ECDH_KEYPAIR</td>
<td>Between 160 and 521 bits. Conditional: Available only if at least one of the ECC curves defined in Table 6-14 with &quot;generic&quot; equal to &quot;Y&quot; is supported. SHALL be same value as for ECDH public key size (for values, see Table 6-14).</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_PUBLIC_KEY</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_25519 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_KEYPAIR</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_25519 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_PUBLIC_KEY</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_25519 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_KEYPAIR</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_25519 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_PUBLIC_KEY</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_KEYPAIR</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_PUBLIC_KEY</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_KEYPAIR</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_PUBLIC_KEY</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_KEYPAIR</td>
<td>256 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 defined in Table 6-14 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_SM4</td>
<td>128 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SM3</td>
<td>Between 80 and 1024 bits, multiple of 8 bits. Conditional: Available only if TEE_ECC_CURVE_SM2 is supported.</td>
</tr>
<tr>
<td>TEE_TYPE_GENERIC_SECRET</td>
<td>Multiple of 8 bits, up to 4096 bits. This type is intended for secret data that has been derived from a key derivation scheme.</td>
</tr>
<tr>
<td>TEE_TYPE_DATA</td>
<td>0 – All data is in the associated data stream.</td>
</tr>
</tbody>
</table>

**Parameters**

- **objectType**: Type of uninitialized object container to be created (see Table 6-13).
- **maxObjectSize**: Key Size of the object. Valid values depend on the object type and are defined in Table 5-9 above.
- **object**: Filled with a handle on the newly created key container.
Specification Number: 10  Function Number: 0x801

Return Code

- TEE_SUCCESS: On success.
- TEE_ERROR_OUT_OF_MEMORY: If not enough resources are available to allocate the object handle.
- TEE_ERROR_NOT_SUPPORTED: If the key size is not supported or the object type is not supported.

Panic Reasons

- If the implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
5.6.2 TEE_FreeTransientObject

Since: TEE Internal API v1.0

```c
void TEE_FreeTransientObject(
  TEE_ObjectHandle object);
```

Description

The `TEE_FreeTransientObject` function deallocates a transient object previously allocated with `TEE_AllocateTransientObject`. After this function has been called, the object handle is no longer valid and all resources associated with the transient object SHALL have been reclaimed.

If the object is initialized, the object attributes are cleared before the object is deallocated.

This function does nothing if `object` is `TEE_HANDLE_NULL`.

Parameters

- `object`: Handle on the object to free

Specification Number: 10   Function Number: 0x803

Panic Reasons

- If `object` is not a valid opened object handle and is not equal to `TEE_HANDLE_NULL`.
- If the Implementation detects any other error.

5.6.3 TEE_ResetTransientObject

Since: TEE Internal API v1.0

```c
void TEE_ResetTransientObject(
  TEE_ObjectHandle object);
```

Description

The `TEE_ResetTransientObject` function resets a transient object to its initial state after allocation. If the object is currently initialized, the function clears the object of all its material. The object is then uninitialized again.

In any case, the function resets the key usage of the container to `0xFFFFFFFFF`.

This function does nothing if `object` is set to `TEE_HANDLE_NULL`.

Parameters

- `object`: Handle on a transient object to reset

Specification Number: 10   Function Number: 0x808

Panic Reasons

- If `object` is not a valid opened object handle and is not equal to `TEE_HANDLE_NULL`.
- If the Implementation detects any other error.
5.6.4 TEE_PopulateTransientObject

Since: TEE Internal API v1.0

```c
TEE_Result TEE_PopulateTransientObject(
    TEE_ObjectHandle object,
    [in] TEE_Attribute* attrs, uint32_t attrCount );
```

Description

The `TEE_PopulateTransientObject` function populates an uninitialized object container with object attributes passed by the TA in the `attrs` parameter.

When this function is called, the object SHALL be uninitialized. If the object is initialized, the caller SHALL first clear it using the function `TEE_ResetTransientObject`.

Note that if the object type is a key-pair, then this function sets both the private and public attributes of the key-pair.

As shown in Table 5-10, the interpretation of the `attrs` parameter depends on the object type. The values of all attributes are copied into the object so that the `attrs` array and all the memory buffers it points to may be freed after this routine returns without affecting the object.

```
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_AES</td>
<td>For all secret key objects, the <code>TEE_ATTR_SECRET_VALUE</code> SHALL be provided.</td>
</tr>
<tr>
<td>TEE_TYPE_DES</td>
<td>For <code>TEE_TYPE_DES</code> and <code>TEE_TYPE_DES3</code>, the buffer associated with this attribute SHALL include parity bits.</td>
</tr>
<tr>
<td>TEE_TYPE_DES3</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_SM4</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_MD5</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SM3</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_GENERIC_SECRET</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_RSA_PUBLIC_KEY</td>
<td>The following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_MODULUS</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_PUBLIC_EXPONENT</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_RSA_KEYPAIR</td>
<td>The following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_MODULUS</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_PUBLIC_EXPONENT</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_PRIVATE_EXPONENT</td>
</tr>
<tr>
<td></td>
<td>The CRT parameters are optional. If any of these attributes is provided, then all of them SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_PRIME1</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_PRIME2</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_EXPONENT1</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_EXPONENT2</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_RSA_COEFFICIENT</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_PUBLIC_KEY</td>
<td>Conditional: If ECC is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_CURVE</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_KEYPAIR</td>
<td>Conditional: If ECC is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_CURVE</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_PUBLIC_KEY</td>
<td>Conditional: If ECC is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_CURVE</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_KEYPAIR</td>
<td>Conditional: If ECC is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_CURVE</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_PUBLIC_KEY</td>
<td>The following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_DSA_PRIME</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_DSA_SUBPRIME</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_DSA_BASE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_DSA_PUBLIC_VALUE</td>
</tr>
<tr>
<td>Object Type</td>
<td>Attributes</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_KEYPAIR</td>
<td>The following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DSA_PRIME</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DSA_SUBPRIME</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DSA_BASE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DSA_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DSA_PUBLIC_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_DH_KEYPAIR</td>
<td>The following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_PRIME</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_BASE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_PUBLIC_VALUE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>The following parameters can optionally be passed:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_SUBPRIME (q)</td>
</tr>
<tr>
<td></td>
<td>If present, constrains the private value x to be in the range [2, q-2], and a mismatch will cause a TEE_ERROR_BAD_PARAMETERS error.</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_DH_X_BITS (l)</td>
</tr>
<tr>
<td></td>
<td>If present, constrains the private value x to have l bits, and a mismatch will cause a TEE_ERROR_BAD_PARAMETERS error.</td>
</tr>
<tr>
<td></td>
<td>If neither of these optional parts is specified, then the only constraint on x is that it is less than p-1.</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_PUBLIC_KEY</td>
<td>Conditional: If TEE_ECC_CURVE_25519 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_ED25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_KEYPAIR</td>
<td>Conditional: If TEE_ECC_CURVE_25519 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_ED25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_ED25519_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_PUBLIC_KEY</td>
<td>Conditional: If TEE_ECC_CURVE_25519 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_X25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_KEYPAIR</td>
<td>Conditional: If TEE_ECC_CURVE_25519 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_X25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_X25519_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_PUBLIC_KEY</td>
<td>Conditional: If TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided (each 32 bytes):</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>- TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>Object Type</td>
<td>Attributes</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_KEYPAIR</td>
<td>Conditional: if TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_PUBLIC_KEY</td>
<td>Conditional: if TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_KEYPAIR</td>
<td>Conditional: if TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_PUBLIC_KEY</td>
<td>Conditional: if TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_KEYPAIR</td>
<td>Conditional: if TEE_ECC_CURVE_SM2 is supported, then the following attributes SHALL be provided:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
</tbody>
</table>

All mandatory attributes SHALL be specified, otherwise the routine will panic.

If attribute values are larger than the maximum size specified when the object was created, the Implementation SHALL panic.

The Implementation can attempt to detect whether the attribute values are consistent; for example, if the numbers supposed to be prime are indeed prime. However, it is not required to do these checks fully and reliably. If it detects invalid attributes, it SHALL return the error code TEE_ERROR_BAD_PARAMETERS and SHALL NOT panic. If it does not detect any inconsistencies, it SHALL be able to later proceed with all operations associated with the object without error. In this case, it is not required to make sensible computations, but all computations SHALL terminate and output some result.

Only the attributes specified in Table 5-10 associated with the object’s type are valid. The presence of any other attribute in the attribute list is an error and will cause the routine to panic.

**Parameters**

- **object**: Handle on an already created transient and uninitialized object
- **attrs, attrCount**: Array of object attributes
Specification Number: 10  Function Number: 0x807

Return Code

- **TEE_SUCCESS**: In case of success. In this case, the content of the object SHALL be initialized.
- **TEE_ERROR_BAD_PARAMETERS**: If an incorrect or inconsistent attribute value is detected. In this case, the content of the object SHALL remain uninitialized.

Panic Reasons

- If object is not a valid opened object handle that is transient and uninitialized.
- If some mandatory attribute is missing.
- If an attribute which is not defined for the object’s type is present in attrs.
- If an attribute value is too big to fit within the maximum object size specified when the object was created.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.6.5 TEE_InitRefAttribute, TEE_InitValueAttribute

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_InitRefAttribute(
    [out] TEE_Attribute* attr,
    uint32_t attributeID,
    [inbuf] void* buffer, size_t length);
```

```c
void TEE_InitValueAttribute(
    [out] TEE_Attribute* attr,
    uint32_t attributeID,
    uint32_t a,
    uint32_t b);
```

Description

The TEE_InitRefAttribute and TEE_InitValueAttribute helper functions can be used to populate a single attribute either with a reference to a buffer or with integer values.

For example, the following code can be used to initialize a DH key generation:

```c
TEE_Attribute attrs[3];
TEE_InitRefAttribute(&attrs[0], TEE_ATTR_DH_PRIME, &p, len);
TEE_InitRefAttribute(&attrs[1], TEE_ATTR_DH_BASE, &g, len);
TEE_InitValueAttribute(&attrs[2], TEE_ATTR_DH_X_BITS, xBits, 0);
TEE_GenerateKey(key, 1024, attrs, sizeof(attrs)/sizeof(TEE_Attribute));
```

Note that in the case of TEE_InitRefAttribute, only the buffer pointer is copied, not the content of the buffer. This means that the attribute structure maintains a pointer back to the supplied buffer. It is the responsibility of the TA author to ensure that the contents of the buffer maintain their value until the attributes array is no longer in use.

Parameters

- attr: attribute structure (defined in section 5.3.1) to initialize
- attributeID: Identifier of the attribute to populate, defined in section 6.1.1
- buffer, length: Input buffer that holds the content of the attribute. Assigned to the corresponding members of the attribute structure defined in section 5.3.1.
- a: unsigned integer value to assign to the a member of the attribute structure defined in section 5.3.1
- b: unsigned integer value to assign to the b member of the attribute structure defined in section 5.3.1

InitRefAttribute: Specification Number: 10 Function Number: 0x805

InitValueAttribute: Specification Number: 10 Function Number: 0x806

Panic Reasons

- If Bit [29] of attributeID describing whether the attribute identifier is a value or reference (as discussed in Table 6-17) is not consistent with the function.
- If the Implementation detects any other error.
Backward Compatibility

TEE Internal Core API v1.1 used a different type for the length.
5.6.6 TEE_CopyObjectAttributes1

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
TEE_Result TEE_CopyObjectAttributes1(
    [out] TEE_ObjectHandle destObject,
    [in]  TEE_ObjectHandle srcObject);
```

Description

This function replaces the TEE_CopyObjectAttributes function, whose use is deprecated.

The TEE_CopyObjectAttributes1 function populates an uninitialized object handle with the attributes of another object handle; that is, it populates the attributes of destObject with the attributes of srcObject.

It is most useful in the following situations:

- To extract the public key attributes from a key-pair object
- To copy the attributes from a persistent object into a transient object

destObject SHALL refer to an uninitialized object handle and SHALL therefore be a transient object.

The source and destination objects SHALL have compatible types and sizes in the following sense:

- The type of destObject SHALL be a subtype of srcObject, i.e. one of the conditions listed in Table 5-11 SHALL be true.

### Table 5-11: TEE_CopyObjectAttributes1 Parameter Types

<table>
<thead>
<tr>
<th>Type of srcObject</th>
<th>Type of destObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Equal to type of srcObject</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_KEYPAIR</td>
<td>TEE_TYPE_RSA_PUBLIC_KEY</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_KEYPAIR</td>
<td>TEE_TYPE_DSA_PUBLIC_KEY</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_KEYPAIR (optional)</td>
<td>TEE_TYPE_ECDSA_PUBLIC_KEY (optional)</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_KEYPAIR (optional)</td>
<td>TEE_TYPE_ECDH_PUBLIC_KEY (optional)</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_KEYPAIR (optional)</td>
<td>TEE_TYPE_ED25519_PUBLIC_KEY (optional)</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_KEYPAIR (optional)</td>
<td>TEE_TYPE_X25519_PUBLIC_KEY (optional)</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_KEYPAIR (optional)</td>
<td>TEE_TYPE_SM2_DSA_PUBLIC_KEY (optional)</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_KEYPAIR (optional)</td>
<td>TEE_TYPE_SM2_PKE_PUBLIC_KEY (optional)</td>
</tr>
</tbody>
</table>

- The size of srcObject SHALL be less than or equal to the maximum size of destObject.

The effect of this function on destObject is identical to the function TEE_PopulateTransientObject except that the attributes are taken from srcObject instead of from parameters.

The object usage of destObject is set to the bitwise AND of the current object usage of destObject and the object usage of srcObject.

Parameters

- destObject: Handle on an uninitialized transient object
• srcObject: Handle on an initialized object

**Specification Number:** 10  **Function Number:** 0x809

**Return Code**

• TEE_SUCCESS: In case of success.
• TEE_ERROR_CORRUPT_OBJECT: If the persistent object is corrupt. The object handle is closed.
• TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

**Panic Reasons**

• If srcObject is not initialized.
• If destObject is initialized.
• If the type and size of srcObject and destObject are not compatible.
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

Prior to TEE Internal Core API v1.2, TEE_CopyObjectAttributes1 did not specify the [in] or [out] annotations.
### 5.6.7 TEE_GenerateKey

**Since:** TEE Internal API v1.0

```c
TEE_Result TEE_GenerateKey(
    TEE_ObjectHandle object,
    uint32_t keySize,
    [in] TEE_Attribute* params, uint32_t paramCount );
```

**Description**

The `TEE_GenerateKey` function generates a random key or a key-pair and populates a transient key object with the generated key material.

The size of the desired key is passed in the `keySize` parameter and SHALL be less than or equal to the maximum key size specified when the transient object was created. The valid values for key size are defined in Table 5-9.

As shown in Table 5-12, the generation algorithm can take parameters depending on the object type.

#### Table 5-12: TEE_GenerateKey Parameters

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_AES</td>
<td>No parameter is necessary. The function generates the attribute TEE_ATTR_SECRET_VALUE. The generated value SHALL be the full key size.</td>
</tr>
<tr>
<td>TEE_TYPE_DES</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_DES3</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_SM4</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_MD5</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SM3</td>
<td></td>
</tr>
<tr>
<td>TEE_TYPE_GENERIC_SECRET</td>
<td></td>
</tr>
<tr>
<td>Object Type</td>
<td>Details</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_KEYPAIR</td>
<td>No parameter is required. The TEE_ATTR_RSA_PUBLIC_EXPONENT attribute may be specified; if omitted, the default value is 65537. Key generation SHALL follow the rules defined in [NIST SP800-56B]. The function generates and populates the following attributes: TEE_ATTR_RSA_MODULUS TEE_ATTR_RSA_PUBLIC_EXPONENT (if not specified) TEE_ATTR_RSA_PRIVATE_EXPONENT TEE_ATTR_RSA_PRIME1 TEE_ATTR_RSA_PRIME2 TEE_ATTR_RSA_EXPONENT1 TEE_ATTR_RSA_EXPONENT2 TEE_ATTR_RSA_COEFFICIENT</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_KEYPAIR</td>
<td>The following domain parameters SHALL be passed to the function: TEE_ATTR_DSA_PRIME TEE_ATTR_DSA_SUBPRIME TEE_ATTR_DSA_BASE The function generates and populates the following attributes: TEE_ATTR_DSA_PUBLIC_VALUE TEE_ATTR_DSA_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_DH_KEYPAIR</td>
<td>The following domain parameters SHALL be passed to the function: TEE_ATTR_DH_PRIME TEE_ATTR_DH_BASE The following parameters can optionally be passed: TEE_ATTR_DH_SUBPRIME (q): If present, constrains the private value x to be in the range ([2, q-2]) TEE_ATTR_DH_X_BITS (l): If present, constrains the private value x to have (l) bits If neither of these optional parts is specified, then the only constraint on x is that it is less than (p-1). The function generates and populates the following attributes: TEE_ATTR_DH_PUBLIC_VALUE TEE_ATTR_DH_PRIVATE_VALUE TEE_ATTR_DH_X_BITS (number of bits in x)</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_KEYPAIR</td>
<td>The following domain parameters SHALL be passed to the function: TEE_ATTR_ECC_CURVE The function generates and populates the following attributes: TEE_ATTR_ECC_PUBLIC_VALUE_X TEE_ATTR_ECC_PUBLIC_VALUE_Y TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td>Object Type</td>
<td>Details</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_KEYPAIR</td>
<td>The following domain parameters SHALL be passed to the function:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_CURVE</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_KEYPAIR</td>
<td>No parameter is required</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ED25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ED25519_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_KEYPAIR</td>
<td>No parameter is required</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_X25519_PUBLIC_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_X25519_PRIVATE_VALUE</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_KEYPAIR</td>
<td>No parameter is required</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_KEYPAIR</td>
<td>No parameter is required</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_KEYPAIR</td>
<td>No parameter is required</td>
</tr>
<tr>
<td></td>
<td>The function generates and populates the following attributes:</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
</tr>
</tbody>
</table>

Once the key material has been generated, the transient object is populated exactly as in the function TEE_PopulateTransientObject except that the key material is randomly generated internally instead of being passed by the caller.

**Parameters**

- **object**: Handle on an uninitialized transient key to populate with the generated key
- **keySize**: Requested key size. SHALL be less than or equal to the maximum key size specified when the object container was created. SHALL be a valid value as defined in Table 5-9.
- **params, paramCount**: Parameters for the key generation. The values of all parameters are copied into the object so that the params array and all the memory buffers it points to may be freed after this routine returns without affecting the object.

---

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
Specification Number: 10  Function Number: 0x804

Return Code

- TEE_SUCCESS: On success.
- TEE_ERROR_BAD_PARAMETERS: If an incorrect or inconsistent attribute is detected. The checks that are performed depend on the implementation.

Panic Reasons

- If object is not a valid opened object handle that is transient and uninitialized.
- If keySize is not supported or is too large.
- If a mandatory parameter is missing.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.7 Persistent Object Functions

5.7.1 TEE_OpenPersistentObject

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_OpenPersistentObject(
    uint32_t          storageID,
    [in(objectIDLength)] void*             objectID, size_t objectIDLen,
    uint32_t          flags,
    [out] TEE_ObjectHandle* object );
```

Description

The `TEE_OpenPersistentObject` function opens a handle on an existing persistent object. It returns a handle that can be used to access the object's attributes and data stream.

The `storageID` parameter indicates which Trusted Storage Space to access. Possible values are defined in Table 5-2.

The `flags` parameter is a set of flags that controls the access rights and sharing permissions with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- Access control flags:
  - TEE_DATA_FLAG_ACCESS_READ: The object is opened with the read access right. This allows the Trusted Application to call the function `TEE_ReadObjectData`.
  - TEE_DATA_FLAG_ACCESS_WRITE: The object is opened with the write access right. This allows the Trusted Application to call the functions `TEE_WriteObjectData` and `TEE_TruncateObjectData`.
  - TEE_DATA_FLAG_ACCESS_WRITE_META: The object is opened with the write-meta access right. This allows the Trusted Application to call the functions `TEE_CloseAndDeletePersistentObject` and `TEE_RenamePersistentObject`.

- Sharing permission control flags:
  - TEE_DATA_FLAG_SHARE_READ: The caller allows another handle on the object to be created with read access.
  - TEE_DATA_FLAG_SHARE_WRITE: The caller allows another handle on the object to be created with write access.

- Other flags are reserved for future use and SHALL be set to 0.

Multiple handles may be opened on the same object simultaneously, but sharing SHALL be explicitly allowed as described in section 5.7.3.

The initial data position in the data stream is set to 0.

Every Trusted Storage implementation is expected to return `TEE_ERROR_CORRUPT_OBJECT` if a Trusted Application attempts to open an object and the TEE determines that its contents (or those of the storage itself) have been tampered with or rolled back.

Parameters

- `storageID`: The storage to use. Valid values are defined in Table 5-2.
objectID, objectIDLen: The object identifier. Note that this buffer cannot reside in shared memory.

flags: The flags which determine the settings under which the object is opened.

object: A pointer to the handle, which contains the opened handle upon successful completion.

If this function fails for any reason, the value pointed to by object is set to TEE_HANDLE_NULL.

When the object handle is no longer required, it SHALL be closed using a call to the TEE_CloseObject function.

Specification Number: 10  Function Number: 0x903

Return Code

• TEE_SUCCESS: In case of success.
• TEE_ERROR_ITEM_NOT_FOUND: If the storage denoted by storageID does not exist or if the object identifier cannot be found in the storage
• TEE_ERROR_ACCESS_CONFLICT: If an access right conflict (see section 5.7.3) was detected while opening the object
• TEE_ERROR_OUT_OF_MEMORY: If there is not enough memory to complete the operation
• TEE_ERROR_CORRUPT_OBJECT: If the storage or object is corrupt
• TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible. It may be associated with the device but unplugged, busy, or inaccessible for some other reason.

Panic Reasons

• If objectIDLen is greater than TEE_OBJECT_ID_MAX_LEN.
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the objectIDLen.
5.7.2 TEE_CreatePersistentObject

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_CreatePersistentObject(
    uint32_t          storageID,
    [in(objectIDLength)] void*             objectID, size_t objectIDLen,
    uint32_t          flags,
    TEE_ObjectHandle  attributes,
    [inbuf] void*             initialData, size_t initialDataLen,
    [out] TEE_ObjectHandle* object );
```

Description

The `TEE_CreatePersistentObject` function creates a persistent object with initial attributes and an initial data stream content, and optionally returns either a handle on the created object, or `TEE_HANDLE_NULL` upon failure.

The `storageID` parameter indicates which Trusted Storage Space to access. Possible values are defined in Table 5-2.

The `flags` parameter is a set of flags that controls the access rights, sharing permissions, and object creation mechanism with which the object handle is opened. The value of the `flags` parameter is constructed by a bitwise-inclusive OR of flags from the following list:

- **Access control flags:**
  - `TEE_DATA_FLAG_ACCESS_READ`: The object is opened with the read access right. This allows the Trusted Application to call the function `TEE_ReadObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE`: The object is opened with the write access right. This allows the Trusted Application to call the functions `TEE_WriteObjectData` and `TEE_TruncateObjectData`.
  - `TEE_DATA_FLAG_ACCESS_WRITE_META`: The object is opened with the write-meta access right. This allows the Trusted Application to call the functions `TEE_CloseAndDeletePersistentObject` and `TEE_RenamePersistentObject`.

- **Sharing permission control flags:**
  - `TEE_DATA_FLAG_SHARE_READ`: The caller allows another handle on the object to be created with read access.
  - `TEE_DATA_FLAG_SHARE_WRITE`: The caller allows another handle on the object to be created with write access.

- **`TEE_DATA_FLAG_OVERWRITE`:** As summarized in Table 5-13:
  - If this flag is present and the object exists, then the object is deleted and re-created as an atomic operation: that is the TA sees either the old object or the new one.
  - If the flag is absent and the object exists, then the function SHALL return `TEE_ERROR_ACCESS_CONFLICT`.

- **Other flags are reserved for future use and SHALL be set to 0.**

The attributes of the newly created persistent object are taken from `attributes`, which can be another persistent object or an initialized transient object. The object type, size, and usage are copied from attributes.
To create a pure data object, the `attributes` argument can also be `NULL`. If `attributes` is `NULL`, the object type SHALL be set to `TEE_TYPE_DATA` to create a pure data object.

Multiple handles may be opened on the same object simultaneously, but sharing SHALL be explicitly allowed as described in section 5.7.3.

The initial data position in the data stream is set to 0.

### Table 5-13: Effect of `TEE_DATA_FLAG_OVERWRITE` on Behavior of `TEE_CreatePersistentObject`

<table>
<thead>
<tr>
<th>TEE_DATA_FLAG_OVERWRITE in flags</th>
<th>Object Exists</th>
<th>Object Created?</th>
<th>Return Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absent</td>
<td>No</td>
<td>Yes</td>
<td>TEE_SUCCESS</td>
</tr>
<tr>
<td>Absent</td>
<td>Yes</td>
<td>No</td>
<td>TEE_ERROR_ACCESS_CONFLICT</td>
</tr>
<tr>
<td>Present</td>
<td>No</td>
<td>Yes</td>
<td>TEE_SUCCESS</td>
</tr>
<tr>
<td>Present</td>
<td>Yes</td>
<td>Deleted and re-created as an atomic operation</td>
<td>TEE_SUCCESS</td>
</tr>
</tbody>
</table>

### Parameters

- `storageID`: The storage to use. Valid values are defined in Table 5-2.
- `objectID`, `objectIDLen`: The object identifier. Note that this cannot reside in shared memory.
- `flags`: The flags which determine the settings under which the object is opened.
- `attributes`: A handle on a persistent object or an initialized transient object from which to take the persistent object attributes. Can be `TEE_HANDLE_NULL` if the persistent object contains no attribute; for example, if it is a pure data object.
- `initialData`, `initialDataLen`: The initial data content of the persistent object.
- `object`: A pointer to the handle, which contains the opened handle upon successful completion. If this function fails for any reason, the value pointed to by `object` is set to `TEE_HANDLE_NULL`. When the object handle is no longer required, it SHALL be closed using a call to the `TEE_CloseObject` function.

### Specification Number: 10  Function Number: 0x902

### Return Code

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the storage denoted by `storageID` does not exist.
- `TEE_ERROR_ACCESS_CONFLICT`: If an access right conflict (see section 5.7.3) was detected while opening the object.
- `TEE_ERROR_OUT_OF_MEMORY`: If there is not enough memory to complete the operation.
- `TEE_ERROR_STORAGE_NO_SPACE`: If insufficient space is available to create the persistent object.
- `TEE_ERROR_CORRUPT_OBJECT`: If the storage is corrupt.
- `TEE_ERROR_STORAGE_NOT_AVAILABLE`: If the persistent object is stored in a storage area which is currently inaccessible. It may be associated with the device but unplugged, busy, or inaccessible for some other reason.
Panic Reasons

- If `objectIdLen` is greater than `TEE_OBJECT_ID_MAX_LEN`.
- If `attributes` is not `TEE_HANDLE_NULL` and is not a valid handle on an initialized object containing the type and attributes of the persistent object to create.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `objectIdLen` and `initialDataLen`. 
5.7.3 Persistent Object Sharing Rules

Multiple handles may be opened on the same object simultaneously using the functions TEE_OpenPersistentObject or TEE_CreatePersistentObject, but sharing SHALL be explicitly allowed. More precisely, at any one time the following constraints apply: If more than one handle is opened on the same object, and if any of these object handles was opened with the flag TEE_DATA_FLAG_ACCESS_READ, then all the object handles SHALL have been opened with the flag TEE_DATA_FLAG_SHARE_READ. There is a corresponding constraint with the flags TEE_DATA_FLAG_ACCESS_WRITE and TEE_DATA_FLAG_SHARE_WRITE. Accessing an object with ACCESS_WRITE_META rights is exclusive and can never be shared.

When one of the functions TEE_OpenPersistentObject or TEE_CreatePersistentObject is called and if opening the object would violate these constraints, then the function returns the return code TEE_ERROR_ACCESS_CONFLICT.

Any bits in flags not defined in Table 5-3 of section 5.4 are reserved for future use and SHALL be set to zero.

The examples in Table 5-14 illustrate the behavior of the TEE_OpenPersistentObject function when called twice on the same object. Note that for readability, the flag names used in Table 5-14 have been abbreviated by removing the 'TEE_DATA_FLAG_' prefix from their name, and any non-TEE_SUCCESS return codes have been shortened by removing the 'TEE_ERROR_' prefix.

---

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
<table>
<thead>
<tr>
<th>Value of flags for First Open/Create</th>
<th>Value of flags for Second Open/Create</th>
<th>Return Code of Second Open/Create</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_READ</td>
<td>ACCESS_READ</td>
<td>ACCESS_CONFLICT</td>
<td>The object handles have not been opened with the flag SHARE_READ. Only the first call will succeed.</td>
</tr>
<tr>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>ACCESS_READ</td>
<td>ACCESS_CONFLICT</td>
</tr>
<tr>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
</tr>
<tr>
<td>ACCESS_READ</td>
<td>ACCESS_WRITE</td>
<td>ACCESS_CONFLICT</td>
<td>Objects are not opened with share flags. Only the first call will succeed.</td>
</tr>
<tr>
<td>ACCESS_WRITE_META</td>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>ACCESS_WRITE</td>
</tr>
<tr>
<td>ACCESS_WRITE_META</td>
<td>(Anything)</td>
<td>(Anything)</td>
<td>ACCESS_CONFLICT</td>
</tr>
<tr>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>SHARE_WRITE</td>
<td>ACCESS_WRITE</td>
</tr>
<tr>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>ACCESS_WRITE</td>
<td>SHARE_WRITE</td>
</tr>
<tr>
<td>SHARE_READ</td>
<td>ACCESS_WRITE</td>
<td>SHARE_WRITE</td>
<td>ACCESS_CONFLICT</td>
</tr>
<tr>
<td>0</td>
<td>ACCESS_READ</td>
<td>SHARE_READ</td>
<td>ACCESS_CONFLICT</td>
</tr>
</tbody>
</table>
5.7.4 TEE_CloseAndDeletePersistentObject1

Since: TEE Internal Core API v1.1

```c
TEE_Result TEE_CloseAndDeletePersistentObject1( TEE_ObjectHandle object );
```

**Description**

This function replaces the `TEE_CloseAndDeletePersistentObject` function, whose use is deprecated.

The `TEE_CloseAndDeletePersistentObject1` function marks an object for deletion and closes the object handle.

The object handle SHALL have been opened with the write-meta access right, which means access to the object is exclusive.

Deleting an object is atomic; once this function returns, the object is definitely deleted and no more open handles for the object exist. This SHALL be the case even if the object or the storage containing it have become corrupted.

The only reason this routine can fail is if the storage area containing the object becomes inaccessible (e.g. the user removes the media holding the object). In this case `TEE_ERROR_STORAGE_NOT_AVAILABLE` SHALL be returned.

If `object` is `TEE_HANDLE_NULL`, the function does nothing.

**Parameters**

- `object`: The object handle

**Specification Number**: 10  **Function Number**: 0x905

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_STORAGE_NOT_AVAILABLE`: If the persistent object is stored in a storage area which is currently inaccessible.

**Panic Reasons**

- If `object` is not a valid handle on a persistent object opened with the write-meta access right.
- If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.7.5 TEE_RenamePersistentObject

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_RenamePersistentObject(
    TEE_ObjectHandle object,
    [in(newObjectIDLen)] void* newObjectID, size_t newObjectIDLen );
```

Description

The function TEE_RenamePersistentObject changes the identifier of an object. The object handle SHALL have been opened with the write-meta access right, which means access to the object is exclusive.

Renaming an object is an atomic operation; either the object is renamed or nothing happens.

Parameters

- **object**: The object handle
- **newObjectID, newObjectIDLen**: A buffer containing the new object identifier. The identifier contains arbitrary bytes, including the zero byte. The identifier length SHALL be less than or equal to TEE_OBJECT_ID_MAX_LEN and can be zero. The buffer containing the new object identifier cannot reside in shared memory.

Specification Number: 10 Function Number: 0x904

Return Code

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_ACCESS_CONFLICT**: If an object with the same identifier already exists
- **TEE_ERROR_CORRUPT_OBJECT**: If the object is corrupt. The object handle is closed.
- **TEE_ERROR_STORAGE_NOT_AVAILABLE**: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If **object** is not a valid handle on a persistent object that has been opened with the write-meta access right.
- If **newObjectID** resides in shared memory.
- If **newObjectIDLen** is more than TEE_OBJECT_ID_MAX_LEN.
- If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the **newObjectIDLen**.
5.8 Persistent Object Enumeration Functions

5.8.1 TEE_AllocatePersistentObjectEnumerator

Since: TEE Internal API v1.0

```c
TEE_Result TEE_AllocatePersistentObjectEnumerator(
    [out] TEE_ObjectEnumHandle* objectEnumerator );
```

Description

The `TEE_AllocatePersistentObjectEnumerator` function allocates a handle on an object enumerator. Once an object enumerator handle has been allocated, it can be reused for multiple enumerations.

Parameters

- `objectEnumerator`: A pointer filled with the newly-allocated object enumerator handle on success. Set to `TEE_HANDLE_NULL` in case of error.

Specification Number: 10  Function Number: 0xA01

Return Code

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OUT_OF_MEMORY`: If there is not enough memory to allocate the enumerator handle

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

5.8.2 TEE_FreePersistentObjectEnumerator

Since: TEE Internal API v1.0

```c
void TEE_FreePersistentObjectEnumerator(
    TEE_ObjectEnumHandle objectEnumerator );
```

Description

The `TEE_FreePersistentObjectEnumerator` function deallocates all resources associated with an object enumerator handle. After this function is called, the handle is no longer valid.

Parameters

- `objectEnumerator`: The handle to close. If `objectEnumerator` is `TEE_HANDLE_NULL`, then this function does nothing.

Specification Number: 10  Function Number: 0xA02

Panic Reasons

- If `objectEnumerator` is not a valid handle on an object enumerator.
- If the Implementation detects any other error.
### 5.8.3 TEE_ResetPersistentObjectEnumerator

Since: TEE Internal API v1.0

```c
void TEE_ResetPersistentObjectEnumerator(
    TEE_ObjectEnumHandle objectEnumerator);
```

**Description**

The `TEE_ResetPersistentObjectEnumerator` function resets an object enumerator handle to its initial state after allocation. If an enumeration has been started, it is stopped. This function does nothing if `objectEnumerator` is `TEE_HANDLE_NULL`.

**Parameters**

- `objectEnumerator`: The handle to reset

**Specification Number**: 10  **Function Number**: 0xA04

**Panic Reasons**

- If `objectEnumerator` is not `TEE_HANDLE_NULL` and is not a valid handle on an object enumerator.
- If the Implementation detects any other error.
5.8.4 TEE_StartPersistentObjectEnumerator

Since: TEE Internal API v1.0

```c
TEE_Result TEE_StartPersistentObjectEnumerator(
    TEE_ObjectEnumHandle objectEnumerator,
    uint32_t             storageID );
```

Description

The `TEE_StartPersistentObjectEnumerator` function starts the enumeration of all the persistent objects in a given Trusted Storage. The object information can be retrieved by calling the function `TEE_GetNextPersistentObject` repeatedly. The enumeration does not necessarily reflect a given consistent state of the storage: During the enumeration, other TAs or other instances of the TA may create, delete, or rename objects. It is not guaranteed that all objects will be returned if objects are created or destroyed while the enumeration is in progress.

To stop an enumeration, the TA can call the function `TEE_ResetPersistentObjectEnumerator`, which detaches the enumerator from the Trusted Storage. The TA can call the function `TEE_FreePersistentObjectEnumerator` to completely deallocate the object enumerator.

If this function is called on an enumerator that has already been started, the enumeration is first reset then started.

Parameters

- `objectEnumerator`: A valid handle on an object enumerator
- `storageID`: The identifier of the storage in which the objects SHALL be enumerated. Possible values are defined in Table 5-2.

Specification Number: 10 Function Number: 0xA05

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If the storage does not exist or if there is no object in the specified storage
- TEE_ERROR_CORRUPT_OBJECT: If the storage is corrupt
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If `objectEnumerator` is not a valid handle on an object enumerator.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.8.5 TEE_GetNextPersistentObject

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```
TEE_Result TEE_GetNextPersistentObject(
    TEE_ObjectEnumHandle objectEnumerator,
    [out] TEE_ObjectInfo* objectInfo,
    [out] void* objectID,
    [out] size_t* objectIDLen );
```

Description

The TEE_GetNextPersistentObject function gets the next object in an enumeration and returns information about the object: type, size, identifier, etc. If there are no more objects in the enumeration or if there is no enumeration started, then the function returns TEE_ERROR_ITEM_NOT_FOUND.

If while enumerating objects a corrupt object is detected, then its object ID SHALL be returned in objectID, objectInfo SHALL be zeroed, and the function SHALL return TEE_ERROR_CORRUPT_OBJECT.

Parameters

- objectEnumerator: A handle on the object enumeration
- objectInfo: A pointer to a TEE_ObjectInfo filled with the object information as specified in the function TEE_GetObjectInfo1 in section 5.5.1. It may be NULL.
- objectID: Pointer to an array able to hold at least TEE_OBJECT_ID_MAX_LEN bytes. On return, the object identifier is written to this location
- objectIDLen: Filled with the size of the object identifier (from 0 to TEE_OBJECT_ID_MAX_LEN)

Specification Number: 10 Function Number: 0xA03

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_ITEM_NOT_FOUND: If there are no more elements in the object enumeration or if no enumeration is started on this handle
- TEE_ERROR_CORRUPT_OBJECT: If the storage or returned object is corrupt
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If objectEnumerator is not a valid handle on an object enumerator.
- If objectID is NULL.
- If objectIDLen is NULL.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the objectIDLen.
5.9 Data Stream Access Functions

These functions can be used to access the data stream of persistent objects. They work like a file API.

5.9.1 TEE_ReadObjectData

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_ReadObjectData(
    TEE_ObjectHandle  object,
    [out] void*        buffer,
    size_t            size,
    [out] uint32_t*    count);
```

Description

The TEE_ReadObjectData function attempts to read `size` bytes from the data stream associated with the object `object` into the buffer pointed to by `buffer`.

The object handle SHALL have been opened with the read access right.

The bytes are read starting at the position in the data stream currently stored in the object handle. The handle’s position is incremented by the number of bytes actually read.

On completion TEE_ReadObjectData sets the number of bytes actually read in the uint32_t pointed to by `count`. The value written to `*count` may be less than `size` if the number of bytes until the end-of-stream is less than `size`. It is set to 0 if the position at the start of the read operation is at or beyond the end-of-stream. These are the only cases where `*count` may be less than `size`.

No data transfer can occur past the current end of stream. If an attempt is made to read past the end-of-stream, the TEE_ReadObjectData function stops reading data at the end-of-stream and returns the data read up to that point. This is still a success. The position indicator is then set at the end-of-stream. If the position is at, or past, the end of the data when this function is called, then no bytes are copied to `*buffer` and `*count` is set to 0.

Parameters

- `object`: The object handle
- `buffer`: A pointer to the memory which, upon successful completion, contains the bytes read
- `size`: The number of bytes to read
- `count`: A pointer to the variable which upon successful completion contains the number of bytes read

Specification Number: 10  Function Number: 0xB01

Return Code

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_CORRUPT_OBJECT`: If the object is corrupt. The object handle is closed.
- `TEE_ERROR_STORAGE_NOT_AVAILABLE`: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If `object` is not a valid handle on a persistent object opened with the read access right.
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `size`. 
5.9.2 TEE_WriteObjectData

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```
TEE_Result TEE_WriteObjectData(
    TEE_ObjectHandle object,
    [in] void* buffer, size_t size );
```

Description

The TEE_WriteObjectData function writes size bytes from the buffer pointed to by buffer to the data stream associated with the open object handle object.

The object handle SHALL have been opened with the write access permission.

If the current data position points before the end-of-stream, then size bytes are written to the data stream, overwriting bytes starting at the current data position. If the current data position points beyond the stream’s end, then the data stream is first extended with zero bytes until the length indicated by the data position indicator is reached, and then size bytes are written to the stream. Thus, the size of the data stream can be increased as a result of this operation.

If the operation would move the data position indicator to beyond its maximum possible value, then TEE_ERROR_OVERFLOW is returned and the operation fails.

The data position indicator is advanced by size. The data position indicators of other object handles opened on the same object are not changed.

Writing in a data stream is atomic; either the entire operation completes successfully or no write is done.

Parameters

- object: The object handle
- buffer: The buffer containing the data to be written
- size: The number of bytes to write

Specification Number: 10 Function Number: 0xB04

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_STORAGE_NO_SPACE: If insufficient storage space is available
- TEE_ERROR_OVERFLOW: If the value of the data position indicator resulting from this operation would be greater than TEE_DATA_MAX_POSITION
- TEE_ERROR_CORRUPT_OBJECT: If the object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If object is not a valid handle on a persistent object opened with the write access right.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
Backward Compatibility

TEE Internal Core API v1.1 used a different type for the size.
5.9.3 TEE_TruncateObjectData

Since: TEE Internal API v1.0

```c
TEE_Result TEE_TruncateObjectData(
    TEE_ObjectHandle object,
    uint32_t size);
```

Description

The function TEE_TruncateObjectData changes the size of a data stream. If `size` is less than the current size of the data stream then all bytes beyond `size` are removed. If `size` is greater than the current size of the data stream then the data stream is extended by adding zero bytes at the end of the stream.

The object handle SHALL have been opened with the write access permission.

This operation does not change the data position of any handle opened on the object. Note that if the current data position of such a handle is beyond `size`, the data position will point beyond the object data's end after truncation.

Truncating a data stream is atomic: Either the data stream is successfully truncated or nothing happens.

Parameters

- `object`: The object handle
- `size`: The new size of the data stream

Specification Number: 10 Function Number: 0xB03

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_STORAGE_NO_SPACE: If insufficient storage space is available to perform the operation
- TEE_ERROR_CORRUPT_OBJECT: If the object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If `object` is not a valid handle on a persistent object opened with the write access right.
- If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
5.9.4 TEE_SeekObjectData

Since: TEE Internal API v1.0

```c
TEE_Result TEE_SeekObjectData(
    TEE_ObjectHandle object,
    int32_t offset,
    TEE_Whence whence);
```

Description

The TEE_SeekObjectData function sets the data position indicator associated with the object handle. The parameter `whence` controls the meaning of `offset`:

- If `whence` is TEE_DATA_SEEK_SET, the data position is set to `offset` bytes from the beginning of the data stream.
- If `whence` is TEE_DATA_SEEK_CUR, the data position is set to its current position plus `offset`.
- If `whence` is TEE_DATA_SEEK_END, the data position is set to the size of the object data plus `offset`.

The TEE_SeekObjectData function may be used to set the data position beyond the end of stream; this does not constitute an error. However, the data position indicator does have a maximum value which is TEE_DATA_MAX_POSITION. If the value of the data position indicator resulting from this operation would be greater than TEE_DATA_MAX_POSITION, the error TEE_ERROR_OVERFLOW is returned.

If an attempt is made to move the data position before the beginning of the data stream, the data position is set at the beginning of the stream. This does not constitute an error.

Parameters

- `object`: The object handle
- `offset`: The number of bytes to move the data position. A positive value moves the data position forward; a negative value moves the data position backward.
- `whence`: The position in the data stream from which to calculate the new position

Specification Number: 10 Function Number: 0xB02

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OVERFLOW: If the value of the data position indicator resulting from this operation would be greater than TEE_DATA_MAX_POSITION
- TEE_ERROR_CORRUPT_OBJECT: If the object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

Panic Reasons

- If `object` is not a valid handle on a persistent object.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.
6 Cryptographic Operations API

This part of the Cryptographic API defines how to actually perform cryptographic operations:

- Cryptographic operations can be pre-allocated for a given operation type, algorithm, and key size. Resulting Cryptographic Operation Handles can be reused for multiple operations.
- When required by the operation, the Cryptographic Operation Key can be set up independently and reused for multiple operations. Note that some cryptographic algorithms, such as AES-XTS, require two keys.
- An operation may be in two states: initial state where nothing is going on and active state where an operation is in progress.
- The cryptographic algorithms listed in Table 6-1 are supported in this specification.

Table 6-1: Supported Cryptographic Algorithms

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Supported Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digests</td>
<td>MD5, SHA-1, SHA-256, SHA-224, SHA-384, SHA-512, SM3-256</td>
</tr>
<tr>
<td>Symmetric ciphers</td>
<td>DES, Triple-DES with double-length and triple-length keys</td>
</tr>
<tr>
<td></td>
<td>AES, SM4</td>
</tr>
<tr>
<td>Message Authentication Codes</td>
<td>DES-MAC, AES-MAC, AES-CMAC, HMAC with one of the supported digests</td>
</tr>
<tr>
<td>(MACs)</td>
<td></td>
</tr>
<tr>
<td>Authenticated Encryption (AE)</td>
<td>AES-CCM with support for Additional Authenticated Data (AAD)</td>
</tr>
<tr>
<td></td>
<td>AES-GCM with support for Additional Authenticated Data (AAD)</td>
</tr>
<tr>
<td>Asymmetric Encryption Schemes</td>
<td>RSA PKCS1-V1.5, RSA OAEP</td>
</tr>
<tr>
<td>Asymmetric Signature Schemes</td>
<td>DSA, RSA PKCS1-V1.5, RSA PSS</td>
</tr>
<tr>
<td>Key Exchange Algorithms</td>
<td>Diffie-Hellman</td>
</tr>
</tbody>
</table>

4 WARNING: Given the increases in computing power, it is necessary to increase the strength of encryption used with time. Many of the algorithms and key sizes included are known to be weak and are included to support legacy implementations only. TA designers should regularly review the choice of cryptographic primitives and key sizes used in their applications and should refer to appropriate Government guidelines.
There are a number of cryptographic algorithms which are optional in this specification. However, if these are present, they SHALL be supported as defined in Table 6-2 if at least one of the algorithms for which they are defined is supported.

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Algorithm Name</th>
<th>When Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric Signature Schemes on generic curve types</td>
<td>ECDSA</td>
<td>Any of the curves in Table 6-14 for which &quot;generic&quot; is Y</td>
</tr>
<tr>
<td>Key Exchange Algorithms on generic curve types</td>
<td>ECDH</td>
<td>Any of the curves in Table 6-14 for which &quot;generic&quot; is Y</td>
</tr>
<tr>
<td>Asymmetric Signature on Edwards Curves</td>
<td>ED25519</td>
<td>Any Edwards curve is supported</td>
</tr>
<tr>
<td>Key Exchange Algorithms on Edwards Curves</td>
<td>X25519</td>
<td>Any Edwards curve is supported</td>
</tr>
<tr>
<td>Various asymmetric Elliptic Curve-based cryptographic schemes using the SM2 curve.</td>
<td>SM2</td>
<td>SM2 is supported</td>
</tr>
<tr>
<td>Various signature and HMAC schemes based on the SM3 hash function.</td>
<td>SM3</td>
<td>SM2 is supported (SM2 support implies support for SM3. See Table 4-14).</td>
</tr>
<tr>
<td>Various symmetric encryption-based schemes based on SM4 symmetric encryption</td>
<td>SM4</td>
<td>SM2 is supported (SM2 support implies support for SM4. See Table 4-14).</td>
</tr>
</tbody>
</table>

Digest, symmetric ciphers, MACs, and AE operations are always multi-stage, i.e. data can be provided in successive chunks to the API. On the other hand, asymmetric operations are always single stage. Note that signature and verification operations operate on a digest computed by the caller.

Operation states can be copied from one operation handle into an uninitialized operation handle. This allows the TA to duplicate or fork a multi-stage operation, for example.
6.1 Data Types

6.1.1 TEE_OperationMode

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

The TEE_OperationMode type is used to specify one of the available cryptographic operations. Table 6-3 defines the legal values of TEE_OperationMode.

typedef uint32_t TEE_OperationMode;

Table 6-3: Possible TEE_OperationMode Values

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_MODE_ENCRYPT</td>
<td>0x00000000</td>
<td>Encryption mode</td>
</tr>
<tr>
<td>TEE_MODE_DECRYPT</td>
<td>0x00000001</td>
<td>Decryption mode</td>
</tr>
<tr>
<td>TEE_MODE_SIGN</td>
<td>0x00000002</td>
<td>Signature generation mode</td>
</tr>
<tr>
<td>TEE_MODE_VERIFY</td>
<td>0x00000003</td>
<td>Signature verification mode</td>
</tr>
<tr>
<td>TEE_MODE_MAC</td>
<td>0x00000004</td>
<td>MAC mode</td>
</tr>
<tr>
<td>TEE_MODE_DIGEST</td>
<td>0x00000005</td>
<td>Digest mode</td>
</tr>
<tr>
<td>TEE_MODE_DERIVE</td>
<td>0x00000006</td>
<td>Key derivation mode</td>
</tr>
<tr>
<td>Reserved for future GlobalPlatform specifications</td>
<td>0x00000007 – 0x7FFFFFFE</td>
<td></td>
</tr>
<tr>
<td>TEE_MODE_ILLEGAL_VALUE</td>
<td>0x7FFFFFFF</td>
<td>Implementation defined</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>0x80000000 – 0xFFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>

Note: TEE_MODE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.

Backward Compatibility

Prior to TEE Internal Core API v1.2, TEE_OperationMode was defined as an enum.
6.1.2 TEE_OperationInfo

Since: TEE Internal API v1.0

```
typedef struct {
    uint32_t algorithm;
    uint32_t operationClass;
    uint32_t mode;
    uint32_t digestLength;
    uint32_t maxKeySize;
    uint32_t keySize;
    uint32_t requiredKeyUsage;
    uint32_t handleState;
} TEE_OperationInfo;
```

See the documentation of function TEE_GetOperationInfo in section 6.2.3 for a description of this structure.

6.1.3 TEE_OperationInfoMultiple

Since: TEE Internal Core API v1.1

```
typedef struct {
    uint32_t keySize;
    uint32_t requiredKeyUsage;
} TEE_OperationInfoKey;

typedef struct {
    uint32_t             algorithm;
    uint32_t             operationClass;
    uint32_t             mode;
    uint32_t             digestLength;
    uint32_t             maxKeySize;
    uint32_t             handleState;
    uint32_t             operationState;
    uint32_t             numberOfKeys;
    TEE_OperationInfoKey keyInformation[];
} TEE_OperationInfoMultiple;
```

See the documentation of function TEE_GetOperationInfoMultiple in section 6.2.4 for a description of this structure.

The buffer size to allocate to hold details of N keys is given by

```
sizeof(TEE_OperationInfoMultiple) + N * sizeof(TEE_OperationInfoKey)
```
6.1.4 TEE_OperationHandle

Since: TEE Internal Core API v1.0

```c
typedef struct __TEE_OperationHandle* TEE_OperationHandle;
```

TEE_OperationHandle is an opaque handle on a cryptographic operation. These handles are returned by the function TEE_AllocateOperation specified in section 6.2.1.
6.2 **Generic Operation Functions**

These functions are common to all the types of cryptographic operations, which are:

- Digests
- Symmetric ciphers
- MACs
- Authenticated Encryptions
- Asymmetric operations
- Key Derivations

### 6.2.1 TEE_AllocateOperation

Since: TEE Internal API v1.0

```c
TEE_Result TEE_AllocateOperation(
    TEE_OperationHandle* operation,
    uint32_t             algorithm,
    uint32_t             mode,
    uint32_t             maxKeySize );
```

**Description**

The **TEE_AllocateOperation** function allocates a handle for a new cryptographic operation and sets the mode and algorithm type. If this function does not return with **TEE_SUCCESS** then there is no valid handle value.

Once a cryptographic operation has been created, the implementation SHALL guarantee that all resources necessary for the operation are allocated and that any operation with a key of at most `maxKeySize` bits can be performed. For algorithms that take multiple keys, for example the AES XTS algorithm, the `maxKeySize` parameter specifies the size of the largest key. It is up to the implementation to properly allocate space for multiple keys if the algorithm so requires.

The parameter `algorithm` SHALL be one of the constants defined in section 6.10.1.

The parameter `mode` SHALL be one of the constants defined in section 6.1.1. It SHALL be compatible with the algorithm as defined by Table 6-4.

The parameter `maxKeySize` SHALL be a valid value as defined in Table 5-9 for the algorithm, for algorithms referenced in Table 5-9. For all other algorithms, the `maxKeySize` parameter may have any value.

The operation is placed in **initial** state.
## Table 6-4: TEE_AllocateOperation Allowed Modes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_AES_ECB_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CBC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CTR</td>
<td>TEE_MODE_ENCRYPT</td>
</tr>
<tr>
<td>TEE_ALG_AES_CTS</td>
<td>TEE_MODE_DECRYPT</td>
</tr>
<tr>
<td>TEE_ALG_AES_XTS</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CCM</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_GCM</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_ECB_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_ECB_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_NOPAD</td>
<td>TEE_MODE_MAC</td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_PKCS5</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CMAC</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_PKCS5</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_PKCS5</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_MDS</td>
<td>TEE_MODE_SIGN</td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA1</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA1</td>
<td>TEE_MODE_SIGN</td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA224</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ED25519</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM2_DSA_SM3</td>
<td></td>
</tr>
<tr>
<td>(if supported)</td>
<td></td>
</tr>
</tbody>
</table>
### Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_RSAES_PKCS1_V1_5</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1</td>
<td>TEE_MODE_ENCRYPT</td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224</td>
<td>TEE_MODE_DECRYPT</td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSA_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM2_PKE</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM4_ECB_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM4_CBC_NOPAD</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM4_CTR</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DH_DERIVE_SHARED_SECRET</td>
<td>TEE_MODE_DERIVE</td>
</tr>
<tr>
<td>TEE_ALG_ECDH_DERIVE_SHARED_SECRET</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_X25519</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM2_KEP</td>
<td></td>
</tr>
<tr>
<td>(if supported)</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_MD5</td>
<td>TEE_MODE_DIGEST</td>
</tr>
<tr>
<td>TEE_ALG_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM3</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_MD5</td>
<td>TEE_MODE_MAC</td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA1</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA224</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA256</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA384</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA512</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SM3</td>
<td></td>
</tr>
</tbody>
</table>

Note that all algorithms listed in Table 6-4 SHALL be supported by any compliant Implementation with the exception of the elliptic curve algorithms which are marked as optional, but a particular implementation may also support more implementation-defined algorithms, modes, or key sizes.

### Parameters

- **operation**: Reference to generated operation handle
- **algorithm**: One of the cipher algorithms listed in section 6.1.1
- **mode**: The operation mode
- **maxKeySize**: Maximum key size in bits for the operation – must be a valid value for the algorithm as defined in Table 5-9.
 Specification Number: 10  Function Number: 0xC01

 Return Code
 • TEE_SUCCESS: In case of success.
 • TEE_ERROR_OUT_OF_MEMORY: If there are not enough resources to allocate the operation.
 • TEE_ERROR_NOT_SUPPORTED: If the mode is not compatible with the algorithm or key size or if the algorithm is not one of the listed algorithms or if maxKeySize is not appropriate for the algorithm.

 Panic Reasons
 • If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
### 6.2.2 TEE_FreeOperation

**Since:** TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_FreeOperation( TEE_OperationHandle operation );
```

**Description**

The TEE_FreeOperation function deallocates all resources associated with an operation handle. After this function is called, the operation handle is no longer valid. All cryptographic material in the operation is destroyed.

The function does nothing if `operation` is `TEE_HANDLE_NULL`.

**Parameters**

- `operation`: Reference to operation handle

**Specification Number:** 10  
**Function Number:** 0xC03

**Panic Reasons**

- If `operation` is not a valid handle on an operation and is not equal to `TEE_HANDLE_NULL`.
- If the Implementation detects any other error.

**Backward Compatibility**

Prior to TEE Internal Core API v1.2, TEE_FreeOperation MAY Panic if `operation` is `TEE_HANDLE_NULL`. 
6.2.3 TEE_GetOperationInfo

Since: TEE Internal API v1.0

```c
void TEE_GetOperationInfo(
    TEE_OperationHandle operation,
    [out] TEE_OperationInfo* operationInfo );
```

Description

The TEE_GetOperationInfo function returns information about an operation handle. It fills the following fields in the structure operationInfo (defined in section 6.2.1):

- `algorithm`, `mode`, `maxKeySize`: The parameters passed to the function TEE_AllocateOperation
- `operationClass`: One of the constants from Table 5-6, describing the kind of operation.
- `keySize`: If a key is programmed in the operation, the actual size of this key. If multiple keys are required by this type of operation, then this value SHALL be set to 0.
- `requiredKeyUsage`: A bit vector that describes the necessary bits in the object usage for TEE_SetOperationKey or TEE_SetOperationKey2 to succeed without panicking. Set to 0 for a digest operation. If multiple keys are required by this type of operation, then this value SHALL be set to 0.
- `digestLength`: For a MAC, AE, or Digest digest, describes the number of bytes in the digest or tag
- `handleState`: A bit vector describing the current state of the operation. Can contain any combination of the following flags or 0 if no flags are appropriate:
  - TEE_HANDLE_FLAG_EXPECT_TWO_KEYS: Set if the algorithm expects two keys to be set, using TEE_SetOperationKey2. This happens only if `algorithm` is set to TEE_ALG_AES_128_XTS. In this case `keySize` and `requiredKeyUsage` are both set to 0; the required information can be retrieved using the TEE_GetOperationInfoMultiple routine defined in section 6.2.4.
  - TEE_HANDLE_FLAG_KEY_SET: Set if the operation key has been set. Always set for digest operations.
  - TEE_HANDLE_FLAG_INITIALIZED: Set for multi-stage operations and for Digest operations.

Parameters

- `operation`: Handle on the operation
- `operationInfo`: Pointer to a structure filled with the operation information

Specification Number: 10   Function Number: 0xC04

Panic Reasons

- If `operation` is not a valid opened operation handle.
- If the Implementation detects any other error.
6.2.4 TEE_GetOperationInfoMultiple

Since: TEE Internal Core API v1.1 – See Backward Compatibility note below.

```c
TEE_Result TEE_GetOperationInfoMultiple(
    TEE_OperationHandle operation,
    [outbuf] TEE_OperationInfoMultiple* operationInfoMultiple, size_t* operationSize );
```

Description

The TEE_GetOperationInfoMultiple function returns information about an operation handle. It fills the following fields in the structure operationInfoMultiple (defined in section 6.1.3):

- `algorithm`, `mode`, `maxKeySize`: The parameters passed to the function TEE_AlipulateOperation
- `operationClass`: One of the constants from Table 5-6, describing the kind of operation.
- `digestLength`: For a MAC, AE, or Digest digest, describes the number of bytes in the digest or tag.
- `handleState`: A bit vector describing the current state of the operation. Contains one or more of the following flags:
  - TEE_HANDLE_FLAG_EXPECT_TWO_KEYS: Set if the algorithm expects two keys to be set, using TEE_SetOperationKey2. This happens only if `algorithm` is set to TEE_ALG_AES_XTS.
  - TEE_HANDLE_FLAG_KEY_SET: Set if all required operation keys have been set. Always set for Digest operations.
  - TEE_HANDLE_FLAG_INITIALIZED: For multi-stage operations, i.e. all but TEE_OPERATION_ASYMMETRIC_XXX operation classes, whether the operation has been initialized using one of the TEE_XXXInit functions. This flag is always set for Digest operations.
- `operationState`: One of the values from Table 5-7. This is set to OPERATION_STATE_ACTIVE if the operation is in active state and to OPERATION_STATE_INITIAL if the operation is in initial state.
- `numberOfKeys`: This is set to the number of keys required by this operation. It indicates the number of TEE_OperationInfoKey structures which follow. May be 0 for an operation which requires no keys.
- `keyInformation`: This array contains `numberOfKeys` entries, each of which defines the details for one key used by the operation, in the order they are defined. For each element:
  - `keySize`: If a key is programmed in the operation, the actual size of this key, otherwise 0.
  - `requiredKeyUsage`: A bit vector that describes the necessary bits in the object usage for TEE_SetOperationKey or TEE_SetOperationKey2 to succeed without panicking.

Parameters

- `operation`: Handle on the operation
- `operationInfoMultiple`, `operationSize`: Buffer filled with the operation information. The number of keys which can be contained is given by:
  ```c
  (*operationSize–sizeof(TEE_OperationInfoMultiple))/sizeof(TEE_OperationInfoKey)+1
  ```
**Specification Number:** 10  
**Function Number:** 0xC08

### Return Code
- **TEE_SUCCESS:** In case of success.
- **TEE_ERROR_SHORT_BUFFER:** If the `operationInfo` buffer is not large enough to hold a `TEE_OperationInfoMultiple` (defined in section 6.1.3) structure containing the required number of keys.

### Panic Reasons
- If `operation` is not a valid opened operation handle.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

### Backward Compatibility
TEE Internal Core API v1.1 used a different type for the `operationSize`. 
6.2.5 TEE_ResetOperation

Since: TEE Internal API v1.0

```c
void TEE_ResetOperation( TEE_OperationHandle operation );
```

Description

For a multi-stage operation, the `TEE_ResetOperation` function resets the `TEE_OperationHandle` to the state after the initial `TEE_AllocateOperation` call with the addition of any keys which were configured subsequent to this so that the `TEE_OperationHandle` can be reused with the same keys.

This function can be called on any operation and at any time after the key is set, but is meaningful only for the multi-stage operations, i.e. symmetric ciphers, MACs, AEs, and digests.

When such a multi-stage operation is active, i.e. when it has been initialized but not yet successfully finalized, then the operation is reset to initial state. The operation key(s) are not cleared.

Parameters

- operation: Handle on the operation

Specification Number: 10 Function Number: 0xC05

Panic Reasons

- If operation is not a valid opened operation handle.
- If the key has not been set yet.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.
6.2.6 TEE_SetOperationKey

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_SetOperationKey(
    TEE_OperationHandle operation,
    [in] TEE_ObjectHandle key );
```

Description

The `TEE_SetOperationKey` function programs the key of an operation; that is, it associates an operation with a key.

The key material is copied from the key object handle into the operation. After the key has been set, there is no longer any link between the operation and the key object. The object handle can be closed or reset and this will not affect the operation. This copied material exists until the operation is freed using `TEE_FreeOperation` or another key is set into the operation.

This function accepts handles on both transient key objects and persistent key objects.

The operation SHALL be in initial state before the operation and remains in initial state afterwards.

The key object type and size SHALL be compatible with the type and size of the operation. The operation mode SHALL be compatible with key usage:

- In general, the operation mode SHALL be allowed in the object usage.
- For the `TEE_ALG_RSA_NOPAD` algorithm:
  - The only supported modes are `TEE_MODE_ENCRYPT` and `TEE_MODE_DECRYPT`.
  - For `TEE_MODE_ENCRYPT`, the object usage SHALL contain both the `TEE_USAGE_ENCRYPT` and `TEE_USAGE_VERIFY` flags.
  - For `TEE_MODE_DECRYPT`, the object usage SHALL contain both the `TEE_USAGE_DECRYPT` and `TEE_USAGE_SIGN` flags.
- For a public key object, the allowed operation modes depend on the type of key and are specified in Table 6-5.

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Allowed Operation Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_RSA_PUBLIC_KEY</td>
<td>TEE_MODE_VERIFY or TEE_MODE_ENCRYPT</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_PUBLIC_KEY</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_PUBLIC_KEY</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_PUBLIC_KEY</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_PUBLIC_KEY</td>
<td>TEE_MODE_DERIVE</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_PUBLIC_KEY</td>
<td>TEE_MODE_DERIVE</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_PUBLIC_KEY</td>
<td>TEE_MODE_VERIFY</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_PUBLIC_KEY</td>
<td>TEE_MODE_DERIVE</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_PUBLIC_KEY</td>
<td>TEE_MODE_ENCRYPT or TEE_MODE_DECRYPT</td>
</tr>
</tbody>
</table>
• If the object is a key-pair then the key parts used in the operation depend on the operation mode as defined in Table 6-6.

<table>
<thead>
<tr>
<th>Operation Mode</th>
<th>Key Parts Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_MODE_VERIFY</td>
<td>Public</td>
</tr>
<tr>
<td>TEE_MODE_SIGN</td>
<td>Private</td>
</tr>
<tr>
<td>TEE_MODE_ENCRYPT</td>
<td>Public</td>
</tr>
<tr>
<td>TEE_MODE_DECRYPT</td>
<td>Private</td>
</tr>
<tr>
<td>TEE_MODE_DERIVE</td>
<td>Public and Private</td>
</tr>
</tbody>
</table>

If key is set to TEE_HANDLE_NULL, then the operation key is cleared.

If a key is present in the operation then it is cleared and all key material copied into the operation is destroyed before the new key is inserted.

**Parameters**

- operation: Operation handle
- key: A handle on a key object

**Specification Number:** 10  **Function Number:** 0xC06

**Return Code**

- TEE_SUCCESS: In case of success.
- TEE_ERROR_CORRUPT_OBJECT: If the object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the persistent object is stored in a storage area which is currently inaccessible.

**Panic Reasons**

- If operation is not a valid opened operation handle.
- If key is not TEE_HANDLE_NULL and is not a valid handle on a key object.
- If key is not initialized.
- If the operation expects no key (digest mode) or two keys (AES-XTS algorithm).
- If the type, size, or usage of key is not compatible with the algorithm, mode, or size of the operation.
- If operation is not in initial state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

Prior to TEE Internal Core API v1.2, TEE_SetOperationKey did not specify the [in] annotation on key.
6.2.7 TEE_SetOperationKey2

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```
TEE_Result TEE_SetOperationKey2(
    [in] TEE_OperationHandle operation,
    [in] TEE_ObjectHandle    key1,
    [in] TEE_ObjectHandle    key2 );
```

Description

The TEE_SetOperationKey2 function initializes an existing operation with two keys. This is used only for the algorithm TEE_ALG_AES_XTS and TEE_ALG_SM2_KEP.

This function works like TEE_SetOperationKey except that two keys are set instead of a single key. key1 and key2 SHALL both be non-NULL or both NULL. key1 and key2 SHALL NOT refer to the same key. In the case of TEE_ALG_SM2_KEP, key1 is the handle to the key object that contains the long-term key, and key2 is the handle to the key object that contains the ephemeral key.

Parameters

- operation: Operation handle
- key1: A handle on a key object
- key2: A handle on a key object

Specification Number: 10 Function Number: 0xC07

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_CORRUPT_OBJECT: If the key1 object is corrupt. The object handle is closed.
- TEE_ERROR_CORRUPT_OBJECT_2: If the key2 object is corrupt. The object handle is closed.
- TEE_ERROR_STORAGE_NOT_AVAILABLE: If the key1 object is stored in a storage area which is currently inaccessible.
- TEE_ERROR_STORAGE_NOT_AVAILABLE_2: If the key2 object is stored in a storage area which is currently inaccessible.
- TEE_ERROR_SECURITY: If the key1 object and the key2 object are the same.

Panic Reasons

- If operation is not a valid opened operation handle.
- If key1 and key2 are not both TEE_HANDLE_NULL and key1 or key2 or both are not valid handles on a key object.
- If key1 and/or key2 are not initialized.
- If the operation expects no key (digest mode) or a single key (all but AES-XTS and SM2-KEP algorithms).
- If the type, size, or usage of key1 or key2 is not compatible with the algorithm, mode, or size of the operation.
• If operation is not in **initial** state.

• Hardware or cryptographic algorithm failure

• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

If backward compatibility with a version of this specification before v1.2 is indicated by a TA, the implementation MAY allow `key1` and `key2` to be the same.

Prior to TEE Internal Core API v1.2, `TEE_SetOperationKey2` did not specify the `[in]` annotation.
6.2.8 TEE_CopyOperation

Since: TEE Internal API v1.2 – See Backward Compatibility note below.

```c
void TEE_CopyOperation(
    [out] TEE_OperationHandle dstOperation,
    [in]  TEE_OperationHandle srcOperation);
```

Description

The TEE_CopyOperation function copies an operation state from one operation handle into another operation handle. This also copies the key material associated with the source operation.

The state of srcOperation including the key material currently set up is copied into dstOperation.

This function is useful in the following use cases:

- "Forking" a digest operation after feeding some amount of initial data
- Computing intermediate digests

The algorithm and mode of dstOperation SHALL be equal to the algorithm and mode of srcOperation.

The state of srcOperation (initial/active) is copied to dstOperation.

If srcOperation has no key programmed, then the key in dstOperation is cleared. If there is a key programmed in srcOperation, then the maximum key size of dstOperation SHALL be greater than or equal to the actual key size of srcOperation.

Parameters

- dstOperation: Handle on the destination operation
- srcOperation: Handle on the source operation

Specification Number: 10 Function Number: 0xC02

Panic Reasons

- If dstOperation or srcOperation is not a valid opened operation handle.
- If the algorithm or mode differ in dstOperation and srcOperation.
- If srcOperation has a key and its size is greater than the maximum key size of dstOperation.
- Hardware or cryptographic algorithm failure.
- If the Implementation detects any other error.

Backward Compatibility

Prior to TEE Internal Core API v1.2, TEE_CopyOperation did not specify the [in] or [out] annotations.
6.2.9 TEE_IsAlgorithmSupported

Since: TEE Internal Core API v1.2

```
TEE_Result TEE_IsAlgorithmSupported(
    [in]    uint32_t algId
    [in]    uint32_t element );
```

**Description**

The TEE_IsAlgorithmSupported function can be used to determine whether a combination of algId and element is supported. Implementations SHALL return false for any value of algDef or element which is reserved for future use.

**Parameters**

- **algId**: An algorithm identifier from Table 6-11
- **element**: A cryptographic element from Table 6-14. Where algId fully defines the required support, the special value TEE_OPTIONALELEMENT_NONE SHOULD be used.

**Specification Number**: 10    **Function Number**: 0xC09

**Return Value**

- **TEE_SUCCESS**: The requested combination of algId and element is supported.
- **TEE_ERROR_NOT_SUPPORTED**: The requested combination of algId and element is not supported.

**Panic Reasons**

TEE_IsAlgorithmSupported SHALL NOT panic.
6.3 Message Digest Functions

6.3.1 TEE_DigestUpdate

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_DigestUpdate(
    TEE_OperationHandle operation,
    [inbuf] void* chunk, size_t chunkSize );
```

Description

The **TEE_DigestUpdate** function accumulates message data for hashing. The message does not have to be block aligned. Subsequent calls to this function are possible.

The operation may be in either initial or active state and becomes active.

Parameters

- **operation**: Handle of a running Message Digest operation
- **chunk, chunkSize**: Chunk of data to be hashed

Specification Number: 10    Function Number: 0xD02

Panic Reasons

- If **operation** is not a valid operation handle of class TEE_OPERATION_DIGEST.
- If input data exceeds maximum length for algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the chunkSize.
6.3.2  TEE_DigestDoFinal

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

Description
The TEE_DigestDoFinal function finalizes the message digest operation and produces the message hash. Afterwards the Message Digest operation is reset to initial state and can be reused.

The input operation may be in either initial or active state.

Parameters
- operation: Handle of a running Message Digest operation
- chunk, chunkLen: Last chunk of data to be hashed
- hash, hashLen: Output buffer filled with the message hash

Specification Number: 10  Function Number: 0xD01

Return Code
- TEE_SUCCESS: On success.
- TEE_ERROR_SHORT_BUFFER: If the output buffer is too small. In this case, the operation is not finalized.

Panic Reasons
- If operation is not a valid operation handle of class TEE_OPERATION_DIGEST.
- If input data exceeds maximum length for algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility
TEE Internal Core API v1.1 used a different type for the chunkLen and hashLen.
6.4 Symmetric Cipher Functions

These functions define the way to perform symmetric cipher operations, such as AES. They cover both block ciphers and stream ciphers.

6.4.1 TEE_CipherInit

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_CipherInit(
    TEE_OperationHandle operation,
    [inbuf] void* IV, size_t IVLen );
```

Description

The TEE_CipherInit function starts the symmetric cipher operation.

- The operation SHALL have been associated with a key.
- If the operation is in active state, it is reset and then initialized.
- If the operation is in initial state, it is moved to active state.

Parameters

- operation: A handle on an opened cipher operation setup with a key
- IV, IVLen: Buffer containing the operation Initialization Vector as appropriate (as indicated in the following table).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>IV Required</th>
<th>Meaning of IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_AES_ECB_NOPAD</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CBC_NOPAD</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CTR</td>
<td>Yes</td>
<td>Initial Counter Value</td>
</tr>
<tr>
<td>TEE_ALG_AES_CTS</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_XTS</td>
<td>Yes</td>
<td>Tweak value</td>
</tr>
<tr>
<td>TEE_ALG_AES_CCM</td>
<td>Yes</td>
<td>Nonce value</td>
</tr>
<tr>
<td>TEE_ALG_AES_GCM</td>
<td>Yes</td>
<td>Nonce value</td>
</tr>
<tr>
<td>TEE_ALG_AES_GCM</td>
<td>Yes</td>
<td>Nonce value</td>
</tr>
<tr>
<td>TEE_ALG_DES_ECB_NOPAD</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_NOPAD</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_ECB_NOPAD</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_NOPAD</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM4 ECB_NOPAD</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM4 CBC_NOPAD</td>
<td>Yes</td>
<td>IV SHOULD be randomly generated. This is the responsibility of the caller.</td>
</tr>
<tr>
<td>TEE_ALG_SM4 CTR</td>
<td>Yes</td>
<td>Initial Counter Value</td>
</tr>
</tbody>
</table>
Specification Number: 10  Function Number: 0xE02

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_CIPHER.
- If no key is programmed in the operation.
- If the Initialization Vector does not have the length required by the algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the IVLen.
6.4.2 TEE_CipherUpdate

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_CipherUpdate(
    TEE_OperationHandle operation,
    [inbuf] void*               srcData, size_t srcLen,
    [outbuf] void*               destData, size_t *destLen );
```

Description

The TEE_CipherUpdate function encrypts or decrypts input data. Input data does not have to be a multiple of block size. Subsequent calls to this function are possible. Unless one or more calls of this function have supplied sufficient input data, no output is generated. The cipher operation is finalized with a call to TEE_CipherDoFinal.

The buffers srcData and destData SHALL be either completely disjoint or equal in their starting positions. The operation SHALL be in active state.

Parameters

- operation: Handle of a running Cipher operation
- srcData, srcLen: Input data buffer to be encrypted or decrypted
- destData, destLen: Output buffer

Specification Number: 10 Function Number: 0xE03

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to contain the output. In this case, the input is not fed into the algorithm.

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_CIPHER.
- If the operation has not been started yet with TEE_CipherInit or has already been finalized.
- If operation is not in active state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the srcLen and destLen.
6.4.3 TEE_CipherDoFinal

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_CipherDoFinal(
    TEE_OperationHandle operation,
    [inbuf] void* srcData, size_t srcLen,
    [outbufopt] void* destData, size_t *destLen );
```

**Description**

The TEE_CipherDoFinal function finalizes the cipher operation, processing data that has not been processed by previous calls to TEE_CipherUpdate as well as data supplied in srcData. The operation handle can be reused or re-initialized.

The buffers srcData and destData SHALL be either completely disjoint or equal in their starting positions.

The operation SHALL be in active state and is set to initial state afterwards.

**Parameters**

- `operation`: Handle of a running Cipher operation
- `srcData, srcLen`: Reference to final chunk of input data to be encrypted or decrypted
- `destData, destLen`: Output buffer. Can be omitted if the output is to be discarded, e.g. because it is known to be empty.

**Specification Number:** 10  **Function Number:** 0xE01

**Return Code**

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to contain the output

**Panic Reasons**

- If `operation` is not a valid operation handle of class TEE_OPERATION_CIPHER.
- If the operation has not been started yet with TEE_CipherInit or has already been finalized.
- If the total length of the input is not a multiple of a block size when the algorithm of the operation is a symmetric block cipher which does not specify padding.
- If `operation` is not in active state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the srcLen and destLen.
6.5 MAC Functions

These functions are used to perform MAC (Message Authentication Code) operations, such as HMAC or AES-CMAC operations.

These functions are not used for Authenticated Encryption algorithms, which SHALL use the functions defined in section 6.6.

6.5.1 TEE_MACInit

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_MACInit(
    TEE_OperationHandle operation,
    [inbuf] void* IV, size_t IVLen );
```

Description

The TEE_MACInit function initializes a MAC operation. The operation SHALL have been associated with a key. If the operation is in active state, it is reset and then initialized. If the operation is in initial state, it moves to active state. If the MAC algorithm does not require an IV, the parameters IV, IVLen are ignored.

Parameters

- operation: Operation handle
- IV, IVLen: Input buffer containing the operation Initialization Vector, if applicable

Specification Number: 10 Function Number: 0xF03

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_MAC.
- If no key is programmed in the operation.
- If the Initialization Vector does not have the length required by the algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the IVLen.
6.5.2 TEE_MACUpdate

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_MACUpdate(
    TEE_OperationHandle operation,
    [inbuf] void* chunk, size_t chunkSize);
```

Description

The **TEE_MACUpdate** function accumulates data for a MAC calculation. Input data does not have to be a multiple of the block size. Subsequent calls to this function are possible. **TEE_MACComputeFinal** or **TEE_MACCompareFinal** are called to complete the MAC operation. The operation SHALL be in **active** state.

Parameters

- `operation`: Handle of a running MAC operation
- `chunk`, `chunkSize`: Chunk of the message to be MACed

**Specification Number:** 10 **Function Number:** 0xF04

Panic Reasons

- If `operation` is not a valid operation handle of class **TEE_OPERATION_MAC**.
- If the operation has not been started yet with **TEE_MACInit** or has already been finalized.
- If input data exceeds maximum length for algorithm.
- If `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility

TEE Internal Core API v1.0 used a different type for the `chunkSize`. 
6.5.3 TEE_MACComputeFinal

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_MACComputeFinal(
    TEE_OperationHandle operation,
    [inbuf] void* message, size_t messageLen,
    [outbuf] void* mac, size_t *macLen );
```

Description

The TEE_MACComputeFinal function finalizes the MAC operation with a last chunk of message, and computes the MAC. Afterwards the operation handle can be reused or re-initialized with a new key.

The operation SHALL be in **active** state and moves to **initial** state afterwards.

Parameters

- **operation**: Handle of a MAC operation
- **message**, **messageLen**: Input buffer containing a last message chunk to MAC
- **mac**, **macLen**: Output buffer filled with the computed MAC

Specification Number: 10  
Function Number: 0xF02

Return Code

- **TEE_SUCCESS**: In case of success.
- **TEE_ERRORSHORT_BUFFER**: If the output buffer is not large enough to contain the computed MAC

Panic Reasons

- If **operation** is not a valid operation handle of class **TEE_OPERATION_MAC**.
- If the operation has not been started yet with **TEE_MACInit** or has already been finalized.
- If input data exceeds maximum length for algorithm.
- If **operation** is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the **messageLen** and **macLen**.
6.5.4 TEE_MACCompareFinal

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_MACCompareFinal(
    TEE_OperationHandle operation,
    [inbuf] void* message, size_t messageLen,
    [inbuf] void* mac, size_t macLen );
```

Description

The `TEE_MACCompareFinal` function finalizes the MAC operation and compares the MAC with the buffer passed to the function. Afterwards the operation handle can be reused and initialized with a new key.

The operation SHALL be in active state and moves to initial state afterwards.

Parameters

- `operation`: Handle of a MAC operation
- `message`, `messageLen`: Input buffer containing the last message chunk to MAC
- `mac`, `macLen`: Input buffer containing the MAC to check

Specification Number: 10 Function Number: 0xF01

Return Code

- `TEE_SUCCESS`: If the computed MAC corresponds to the MAC passed in the parameter `mac`.
- `TEE_ERROR_MAC_INVALID`: If the computed MAC does not correspond to the value passed in the parameter `mac`.

Panic Reasons

- If `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- If the operation has not been started yet with `TEE_MACInit` or has already been finalized.
- If input data exceeds maximum length for algorithm.
- If `operation` is not in active state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `messageLen` and `macLen`.
6.6 Authenticated Encryption Functions

These functions are used for Authenticated Encryption operations, i.e. the TEE_ALG_AES_CCM and TEE_ALG_AES_GCM algorithms.

6.6.1 TEE_AEInit

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```
TEE_Result TEE_AEInit(
    TEE_OperationHandle operation,
    [inbuf] void* nonce, size_t nonceLen,
    uint32_t tagLen,
    uint32_t AADLen,
    uint32_t payloadLen );
```

Description

The TEE_AEInit function initializes an Authentication Encryption operation.

The operation must be initial state and remains in the initial state afterwards.

Parameters

- operation: A handle on the operation
- nonce, nonceLen: The operation nonce or IV
- tagLen: Size in bits of the tag
  - For AES-GCM, can be 128, 120, 112, 104, or 96
  - For AES-CCM, can be 128, 112, 96, 80, 64, 48, or 32
- AADLen: Length in bytes of the AAD
  - Used only for AES-CCM. Ignored for AES-GCM.
- payloadLen: Length in bytes of the payload
  - Used only for AES-CCM. Ignored for AES-GCM.

Specification Number: 10  Function Number: 0x1003

Return Code

- TEE_SUCCESS: On success.
- TEE_ERROR_NOT_SUPPORTED: If the tag length is not supported by the algorithm

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_AE.
- If no key is programmed in the operation.
- If the nonce length is not compatible with the length required by the algorithm.
- If operation is not in initial state.
- Hardware or cryptographic algorithm failure
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `nonceLen`. 
6.6.2  TEE_AEUpdateAAD

Since: TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
void TEE_AEUpdateAAD(
    TEE_OperationHandle operation,
    [inbuf] void*               AADdata, size_t AADdataLen );
```

Description

The TEE_AEUpdateAAD function feeds a new chunk of Additional Authentication Data (AAD) to the AE
operation. Subsequent calls to this function are possible.

The buffers srcData and destData SHALL be either completely disjoint or equal in their starting positions.

The operation SHALL be in initial state and remains in initial state afterwards.

Parameters

- operation: Handle on the AE operation
- AADdata, AADdataLen: Input buffer containing the chunk of AAD

Specification Number: 10  Function Number: 0x1005

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_AE.
- If the operation has not started yet.
- If the AAD length has already been reached (AES-CCM only).
- If operation is not in initial state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility

Versions of TEE_AEUpdateAAD prior to v1.2 can be called in any state and entered active state on return.

TEE Internal Core API v1.1 used a different type for the AADdataLen.
### 6.6.3 TEE_AEUpdate

**Since:** TEE Internal Core API v1.2 – See Backward Compatibility note below.

```c
TEE_Result TEE_AEUpdate(
    TEE_OperationHandle operation,
    [inbuf] void* srcData, size_t srcLen,
    [outbuf] void* destData, size_t *destLen );
```

**Description**

The TEE_AEUpdate function accumulates data for an Authentication Encryption operation. Input data does not have to be a multiple of block size. Subsequent calls to this function are possible. Unless one or more calls of this function have supplied sufficient input data, no output is generated. Warning: when using this routine to decrypt the returned data may be corrupt since the integrity check is not performed until all the data has been processed. If this is a concern then only use the TEE_AEDecryptFinal routine.

The operation may be in either initial or active state and enters active state afterwards if srcLen != 0.

**Parameters**

- `operation`: Handle of a running AE operation
- `srcData`, `srcLen`: Input data buffer to be encrypted or decrypted
- `destData`, `destLen`: Output buffer

**Specification Number:** 10  **Function Number:** 0x1004

**Return Code**

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to contain the output

**Panic Reasons**

- If operation is not a valid operation handle of class TEE_OPERATION_AE.
- If the operation has not started yet.
- If the required AAD length has not been provided yet (AES-CCM only).
- If the payload length has already been reached (AES-CCM only).
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

Versions of TEE_AEUpdate prior to v1.2 can be called in any state and entered active state on return regardless of the value of srcLen.

TEE Internal Core API v1.1 used a different type for the srcLen and destLen.
6.6.4 TEE_AEEncryptFinal

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_AEEncryptFinal(
    TEE_OperationHandle operation,
    [inbuf] void* srcData, size_t srcLen,
    [outbuf] void* destData, size_t* destLen,
    [outbuf] void* tag, size_t* tagLen);
```

**Description**

The `TEE_AEEncryptFinal` function processes data that has not been processed by previous calls to `TEE_AEUpdate` as well as data supplied in `srcData`. It completes the AE operation and computes the tag. The operation handle can be reused or newly initialized. The buffers `srcData` and `destData` SHALL be either completely disjoint or equal in their starting positions. The operation may be in either *initial* or *active* state and enters *initial* state afterwards.

**Parameters**

- `operation`: Handle of a running AE operation
- `srcData`, `srcLen`: Reference to final chunk of input data to be encrypted
- `destData`, `destLen`: Output buffer. Can be omitted if the output is to be discarded, e.g. because it is known to be empty.
- `tag`, `tagLen`: Output buffer filled with the computed tag

**Specification Number:** 10  **Function Number:** 0x1002

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_SHORT_BUFFER`: If the output or tag buffer is not large enough to contain the output

**Panic Reasons**

- If `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- If the operation has not started yet.
- If the required AAD and payload have not been provided.
- Hardware or cryptographic algorithm failure.
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `srcLen`, `destLen`, and `tagLen`. 
### 6.6.5 TEE_AEDecryptFinal

**Since:** TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_AEDecryptFinal(
    TEE_OperationHandle operation,
    [inbuf]     void* srcData, size_t srcLen,
    [outbuf]    void* destData, size_t *destLen,
    [in]        void* tag, size_t tagLen);
```

**Description**

The `TEE_AEDecryptFinal` function processes data that has not been processed by previous calls to `TEE_AEUpdate` as well as data supplied in `srcData`. It completes the AE operation and compares the computed tag with the tag supplied in the parameter `tag`.

The operation handle can be reused or newly initialized.

The buffers `srcData` and `destData` SHALL be either completely disjoint or equal in their starting positions.

The operation may be in either initial or active state and enters initial state afterwards.

**Parameters**

- **operation**: Handle of a running AE operation
- **srcData**, **srcLen**: Reference to final chunk of input data to be decrypted
- **destData**, **destLen**: Output buffer. Can be omitted if the output is to be discarded, e.g. because it is known to be empty.
- **tag**, **tagLen**: Input buffer containing the tag to compare

**Specification Number:** 10  **Function Number:** 0x1001

**Return Code**

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_SHORT_BUFFER**: If the output buffer is not large enough to contain the output
- **TEE_ERROR_MAC_INVALID**: If the computed tag does not match the supplied tag

**Panic Reasons**

- If `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- If the operation has not started yet.
- If the required AAD and payload have not been provided.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `srcLen`, `destLen`, and `tagLen`. 
6.7 Asymmetric Functions

These functions allow the encryption and decryption of data using asymmetric algorithms, signatures of digests, and verification of signatures.

Note that asymmetric encryption is always “single-stage”, which differs from symmetric ciphers which are always “multi-stage”.

6.7.1 TEE_AsymmetricEncrypt, TEE_AsymmetricDecrypt

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_AsymmetricEncrypt(
    TEE_OperationHandle operation,
    TEE_Attribute* params, uint32_t paramCount,
    void* srcData, size_t srcLen,
    void* destData, size_t *destLen );
```

```c
TEE_Result TEE_AsymmetricDecrypt(
    TEE_OperationHandle operation,
    TEE_Attribute* params, uint32_t paramCount,
    void* srcData, size_t srcLen,
    void* destData, size_t *destLen );
```

Description

The TEE_AsymmetricEncrypt function encrypts a message within an asymmetric operation, and the TEE_AsymmetricDecrypt function decrypts the result.

These functions can be called only with an operation of the following algorithms:

- TEE_ALG_RSAES_PKCS1_V1_5
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384
- TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512
- TEE_ALG_RSA_NOPAD
- TEE_ALG_SM2_PKE (if supported)

The parameters params, paramCount contain the operation parameters listed in Table 6-7.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Operation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_XXX</td>
<td>TEE_ATTR_RSA_OAEP_LABEL: This parameter is optional. If not present, an empty label is assumed.</td>
</tr>
</tbody>
</table>
Parameters

- operation: Handle on the operation, which SHALL have been suitably set up with an operation key
- params, paramCount: Optional operation parameters
- srcData, srcLen: Input buffer
- destData, destLen: Output buffer

**TEE_AsymmetricDecrypt**: Specification Number: 10 Function Number: 0x1101

**TEE_AsymmetricEncrypt**: Specification Number: 10 Function Number: 0x1102

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SHORT_BUFFER: If the output buffer is not large enough to hold the result
- TEE_ERROR_BAD_PARAMETERS: If the length of the input buffer is not consistent with the algorithm or key size. Refer to Table 5-9 for algorithm references and supported sizes.
- TEE_ERROR_CIPHERTEXT_INVALID: If there is an error in the packing used on the ciphertext.

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATIONASYMMETRIC_CIPHER.
- If no key is programmed in the operation.
- If the mode is not compatible with the function.
- Hardware or cryptographic algorithm failure
- If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

Versions of this specification prior to v1.2 do not define TEE behavior in the event of incorrectly padded ciphertext. It is recommended that implementations generate the error TEE_BAD_PARAMETERS when the ciphertext is invalid. In particular, implementations SHOULD NOT Panic in this scenario.

TEE Internal Core API v1.1 used a different type for the srcLen and destLen of both functions.
6.7.2 TEE_AsymmetricSignDigest

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_AsymmetricSignDigest(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params, uint32_t paramCount,
    [inbuf] void* digest, size_t digestLen,
    [outbuf] void* signature, size_t *signatureLen
);
```

Description

The TEE_AsymmetricSignDigest function signs a message digest within an asymmetric operation.

Note that only an already hashed message can be signed, with the exception of TEE_ALG_ED25519 for which `digest` and `digestLen` refer to the message to be signed.

This function can be called only with an operation of the following algorithms:

- TEE_ALG_RSASSA_PKCS1_V1_5_MD5
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA1
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA224
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA256
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA384
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA512
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512
- TEE_ALG_DSA_SHA1
- TEE_ALG_DSA_SHA224
- TEE_ALG_DSA_SHA256
- TEE_ALG_ECDSA_SHA1 (if supported)
- TEE_ALG_ECDSA_SHA224 (if supported)
- TEE_ALG_ECDSA_SHA256 (if supported)
- TEE_ALG_ECDSA_SHA384 (if supported)
- TEE_ALG_ECDSA_SHA512 (if supported)
- TEE_ALG_ED25519 (if supported)
- TEE_ALG_SM2_DSA_SM3 (if supported)

The parameters params, paramCount contain the operation parameters listed in Table 6-8.
Table 6-8: Asymmetric Sign Operation Parameters

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Operation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX</td>
<td>TEE_ATTR_RSA_PSS_SALT_LENGTH: Number of bytes in the salt. This parameter is optional. If not present, the salt length is equal to the hash length.</td>
</tr>
</tbody>
</table>
| TEE_ALG_ED25519 | TEE_ATTR_ED25519_PH: Optional uint32_t, default 0.  
  - If non-zero, algorithm selected is Ed25519ph ([Ed25519]) and TEE_ATTR_ED25519_CTX must be present (but may be empty).  
  - If present, the value SHALL be present in attribute 'a'. Any value in 'b' SHALL be ignored.  
  TEE_ATTR_ED25519_CTX: Optional buffer, maximum length 255.  
  - If not present, algorithm is Ed25519.  
  - If present and TEE_ATTR_ED25519_PH is zero, algorithm is Ed25519ctx.  
  - If present and TEE_ATTR_ED25519_PH is non-zero, algorithm is Ed25519ph. |

Where a hash algorithm is specified in the algorithm, digestLen SHALL be equal to the digest length of this hash algorithm.

Parameters

- operation: Handle on the operation, which SHALL have been suitably set up with an operation key
- params, paramCount: Optional operation parameters
- digest, digestLen: Input buffer containing the input message digest
- signature, signatureLen: Output buffer written with the signature of the digest

Specification Number: 10  Function Number: 0x1103

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SHORT_BUFFER: If the signature buffer is not large enough to hold the result

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATIONASYMMETRIC_SIGNATURE.
- If no key is programmed in the operation.
- If the operation mode is not TEE_MODE_SIGN.
- If digestLen is not equal to the hash size of the algorithm
- Hardware or cryptographic algorithm failure
- If an optional algorithm which is not supported by the Trusted OS is passed in TEE_OperationHandle.
- If an illegal value is passed as an operation parameter.
• If the Implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `digestLen` and `signatureLen`. 
6.7.3 TEE_AsymmetricVerifyDigest

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_AsymmetricVerifyDigest(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params, uint32_t paramCount,
    [inbuf] void* digest, size_t digestLen,
    [inbuf] void* signature, size_t signatureLen);
```

Description

The TEE_AsymmetricVerifyDigest function verifies a message digest signature within an asymmetric operation.

This function can be called only with an operation of the following algorithms:

- TEE_ALG_RSASSA_PKCS1_V1_5_MD5
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA1
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA224
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA256
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA384
- TEE_ALG_RSASSA_PKCS1_V1_5_SHA512
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384
- TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512
- TEE_ALG_DSA_SHA1
- TEE_ALG_DSA_SHA224
- TEE_ALG_DSA_SHA256
- TEE_ALG_ECDSA_SHA1 (if supported)
- TEE_ALG_ECDSA_SHA224 (if supported)
- TEE_ALG_ECDSA_SHA256 (if supported)
- TEE_ALG_ECDSA_SHA384 (if supported)
- TEE_ALG_ECDSA_SHA512 (if supported)
- TEE_ALG_ED25519 (if supported)
- TEE_ALG_SM2_DSA_SM3 (if supported)

The parameters params, paramCount contain the operation parameters listed in Table 6-9.
Table 6-9: Asymmetric Verify Operation Parameters

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Operation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX</td>
<td>TEE_ATTR_RSA_PSS_SALT_LENGTH: Number of bytes in the salt. This parameter is optional. If not present, the salt length is equal to the hash length.</td>
</tr>
</tbody>
</table>
| TEE_ALG_ED25519                                               | TEE_ATTR_ED25519_PH: Optional uint32_t, default 0.  
|                                                              | o If non-zero, algorithm selected is Ed25519ph ([Ed25519]) and TEE_ATTR_ED25519_CTX must be present (but may be empty).  
|                                                              | TEE_ATTR_ED25519_CTX: Optional buffer, maximum length 255.  
|                                                              | o If not present, algorithm is Ed25519.  
|                                                              | o If present and TEE_ATTR_ED25519_PH is zero, algorithm is Ed25519ctx.  
|                                                              | o If present and TEE_ATTR_ED25519_PH is non-zero, algorithm is Ed25519ph. |

Where a hash algorithm is specified in the algorithm, digestLen SHALL be equal to the digest length of this hash algorithm.

Parameters

- operation: Handle on the operation, which SHALL have been suitably set up with an operation key
- params, paramCount: Optional operation parameters
- digest, digestLen: Input buffer containing the input message digest
- signature, signatureLen: Input buffer containing the signature to verify

Specification Number: 10  Function Number: 0x1104

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_SIGNATURE_INVALID: If the signature is invalid

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATIONASYMMETRIC_SIGNATURE.
- If no key is programmed in the operation.
- If the operation mode is not TEE_MODE_VERIFY.
- If digestLen is not equal to the hash size of the algorithm
- Hardware or cryptographic algorithm failure
- If an optional algorithm which is not supported by the Trusted OS is passed in TEE_OperationHandle.
- If an illegal value is passed as an operation parameter.
• If the implementation detects any other error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `digestLen` and `signatureLen`. 
6.8 Key Derivation Functions

6.8.1 TEE_DeriveKey

Since: TEE Internal API v1.0; changed in v1.2 – See Backward Compatibility note below.

```c
void TEE_DeriveKey(
    TEE_OperationHandle operation,
    [inout] TEE_Attribute* params, uint32_t paramCount,
    TEE_ObjectHandle derivedKey );
```

Description

The TEE_DeriveKey function takes one of the Asymmetric Derivation Operation Parameters in Table 6-10 as input, and outputs a key object.

The TEE_DeriveKey function can only be used with algorithms defined in Table 6-10.

The parameters params, paramCount contain the operation parameters listed in Table 6-10.
### Table 6-10: Asymmetric Derivation Operation Parameters

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Possible Operation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_DH_DERIVE_SHARED_SECRET</td>
<td>TEE_ATTR_DH_PUBLIC_VALUE: Public key of the other party. This parameter is mandatory.</td>
</tr>
<tr>
<td>TEE_ALG_ECDH_DERIVE_SHARED_SECRET</td>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X, TEE_ATTR_ECC_PUBLIC_VALUE_Y: Public key of the other party. These parameters are mandatory.</td>
</tr>
<tr>
<td>TEE_ALG_X25519</td>
<td>TEE_ATTR_X25519_PUBLIC_VALUE: Public key of the other party. This parameter is mandatory.</td>
</tr>
<tr>
<td>TEE_ALG_SM2_KEP (if supported)</td>
<td>Mandatory parameters: TEE_ATTR_ECC_PUBLIC_VALUE_X, TEE_ATTR_ECC_PUBLIC_VALUE_Y: Public key of the other party.</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_SM2_KEP_USER: Value specifying the role of the user. Value 0 means initiator and non-zero means responder.</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_SM2_KEP_CONFIRMATION_IN: Confirmation value from the other peer (optional).</td>
</tr>
<tr>
<td></td>
<td>TEE_ATTR_SM2_KEP_CONFIRMATION_OUT: Confirmation value of the caller (optional).</td>
</tr>
</tbody>
</table>

The `derivedKey` handle SHALL refer to an object with type `TEE_TYPE_GENERIC_SECRET`, unless the algorithm is `TEE_ALG_SM2_KEP`, in which case it MUST refer to an object of type `TEE_TYPE_GENERIC_SECRET`, `TEE_TYPE_SM4`, or `TEE_TYPE_HMAC_SM3`. The caller SHALL have set the private part of the operation DH key using the `TEE_SetOperationKey` function. The caller SHALL pass the other party’s public key as a parameter of the `TEE_DeriveKey` function. On completion the derived key is placed into the `TEE_ATTR_SECRET_VALUE` attribute of the `derivedKey` handle.
In the case of TEE_ALG_SM2_KEP, the caller SHALL have set the long-term and ephemeral private key of the
caller by using TEE_SetOperationKey2. The caller must provide additional attributes specifying role,  
ephemeral public key of other peer, and identifiers of both peers. Two roles exist, initiator and responder; one  
or both of the parties may confirm the Key Agreement result. The function computes and populates the  
TEE_ATTR_SM2_KEP_CONFIRMATION_OUT parameter, which the other peer will use as the  
TEE_ATTR_SM2_KEP_CONFIRMATION_IN parameter.

Parameters

- operation: Handle on the operation, which SHALL have been suitably set up with an operation key  
- params, paramCount: Operation parameters  
- derivedKey: Handle on an uninitialized transient object to be filled with the derived key

Specification Number: 10    Function Number: 0x1201

Panic Reasons

- If operation is not a valid operation handle of class TEE_OPERATION_KEY_DERIVATION. 
- If the object derivedKey is too small for the generated value.  
- If no key is programmed in the operation.  
- If a mandatory parameter is missing.  
- If the operation mode is not TEE_MODE_DERIVE.  
- Hardware or cryptographic algorithm failure  
- If an optional algorithm which is not supported by the Trusted OS is passed in  
  TEE_OperationHandle.  
- If the Implementation detects any other error.

Backward Compatibility

Since: TEE Internal API v1.0

Backward compatibility with a previous version of the Internal Core API can be selected at compile time (see  
section 3.5.1).

```c
void TEE_DeriveKey(  
  TEE_OperationHandle operation,  
  [in] TEE_Attribute* params, uint32_t paramCount,  
  TEE_ObjectHandle derivedKey );
```
6.9 Random Data Generation Function

6.9.1 TEE_GenerateRandom

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_GenerateRandom(
    [out] void* randomBuffer,
    size_t randomBufferLen);
```

Description
The TEE_GenerateRandom function generates random data.

Parameters
- `randomBuffer`: Reference to generated random data
- `randomBufferLen`: Byte length of requested random data

Specification Number: 10  Function Number: 0x1301

Panic Reasons
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

Backward Compatibility
TEE Internal Core API v1.1 used a different type for the `randomBufferLen`. 
6.10 Cryptographic Algorithms Specification

This section specifies the cryptographic algorithms, key types, and key parts supported in the Cryptographic Operations API.

Note that for the “NOPAD” symmetric algorithms, it is the responsibility of the TA to do the padding.

6.10.1 List of Algorithm Identifiers

Table 6-11 provides an exhaustive list of all algorithm identifiers specified in the Cryptographic Operations API. Normative references for the algorithms may be found in Annex C.

Implementations MAY define their own algorithms. Such algorithms SHALL have implementation-defined algorithm identifiers and these identifiers SHALL use 0xF0 as the most significant byte (i.e. they fall in the range 0xF0000000-0xF0FFFFFF).

Note: Previous versions of this specification used bit-fields to construct the algorithm identifier values. Beginning with version 1.2, this is no longer the case and no special significance is given to the bit positions within algorithm identifier values.

<table>
<thead>
<tr>
<th>Algorithm Identifier</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_AES_ECB_NOPAD</td>
<td>0x10000010</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CBC_NOPAD</td>
<td>0x10000110</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CTR</td>
<td>0x10000210</td>
<td>The counter SHALL be encoded as a 16-byte buffer in big-endian form. Between two consecutive blocks, the counter SHALL be incremented by 1. If it reaches the limit of all 128 bits set to 1, it SHALL wrap around to 0.</td>
</tr>
<tr>
<td>TEE_ALG_AES_CTS</td>
<td>0x10000310</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_XTS</td>
<td>0x10000410</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CBC_MAC_NOPAD</td>
<td>0x30000110</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CBC_MAC_PKCS5</td>
<td>0x30000510</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CMAC</td>
<td>0x30000610</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_CCM</td>
<td>0x40000710</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_AES_GCM</td>
<td>0x40000810</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_ECB_NOPAD</td>
<td>0x10000911</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_NOPAD</td>
<td>0x10001111</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_NOPAD</td>
<td>0x30000111</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_PKCS5</td>
<td>0x30000511</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_ECB_NOPAD</td>
<td>0x1000013</td>
<td>Triple DES SHALL be understood as Encrypt-Decrypt-Encrypt mode with two or three keys.</td>
</tr>
<tr>
<td>Algorithm Identifier</td>
<td>Value</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_NOPAD</td>
<td>0x10000113</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_NOPAD</td>
<td>0x30000113</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_PKCS5</td>
<td>0x30000513</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_MD5</td>
<td>0x70001830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA1</td>
<td>0x70002830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA224</td>
<td>0x70003830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA256</td>
<td>0x70004830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA384</td>
<td>0x70005830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA512</td>
<td>0x70006830</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1</td>
<td>0x70212930</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224</td>
<td>0x70313930</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256</td>
<td>0x70414930</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384</td>
<td>0x70515930</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512</td>
<td>0x70616930</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_V1_5</td>
<td>0x60000130</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1</td>
<td>0x60210230</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224</td>
<td>0x60310230</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256</td>
<td>0x60410230</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384</td>
<td>0x60510230</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512</td>
<td>0x60610230</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSA_NOPAD</td>
<td>0x60000030</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA1</td>
<td>0x70002131</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA224</td>
<td>0x70003131</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA256</td>
<td>0x70004131</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DH_DERIVE_SHARED_SECRET</td>
<td>0x80000032</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_MD5</td>
<td>0x5000001</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA1</td>
<td>0x5000002</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA224</td>
<td>0x5000003</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA256</td>
<td>0x5000004</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA384</td>
<td>0x5000005</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA512</td>
<td>0x5000006</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_MD5</td>
<td>0x30000001</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA1</td>
<td>0x30000002</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA224</td>
<td>0x30000003</td>
<td></td>
</tr>
<tr>
<td>Algorithm Identifier</td>
<td>Value</td>
<td>Comments</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA256</td>
<td>0x30000004</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA384</td>
<td>0x30000005</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA512</td>
<td>0x30000006</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SM3</td>
<td>0x30000007</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA1</td>
<td>0x70001042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA224</td>
<td>0x70002042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA256</td>
<td>0x70003042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA384</td>
<td>0x70004042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA512</td>
<td>0x70005042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ED25519</td>
<td>0x70006043</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ECDH_DERIVE_SHARED_SECRET</td>
<td>0x80000042</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_X25519</td>
<td>0x80000044</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM2_DSA_SM3</td>
<td>0x70006045</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM2_KEP</td>
<td>0x60000045</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM2_PKE</td>
<td>0x80000045</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM3</td>
<td>0x50000007</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM4_ECB_NOPAD</td>
<td>0x10000114</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM4_CBC_NOPAD</td>
<td>0x10000114</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_SM4_CTR</td>
<td>0x10000214</td>
<td>If supported</td>
</tr>
<tr>
<td>TEE_ALG_ILLEGAL_VALUE</td>
<td>0xEFFFFFFF</td>
<td>Reserved for GlobalPlatform</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compliance test applications</td>
</tr>
<tr>
<td>Reserved for...</td>
<td>0xF0000000 - 0xF0FFFFFF</td>
<td>All other values are reserved.</td>
</tr>
</tbody>
</table>

Table 6-12: Structure of Algorithm Identifier or Object Type Identifier

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>This table existed in previous versions of the specification and was removed in version 1.2.</td>
</tr>
</tbody>
</table>

Table 6-12b: Algorithm Subtype Identifier

<table>
<thead>
<tr>
<th>Value</th>
<th>Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This table existed in previous versions of the specification and was removed in version 1.2.</td>
</tr>
</tbody>
</table>
6.10.2 Object Types

Object handles are a special class of algorithm handle.

Implementations MAY define their own object handles. Such handles SHALL have implementation-defined object type identifiers and these identifiers SHALL use 0xF0 as the most significant byte (i.e. they fall in the range 0xF0000000-0xF0FFFFFF).

Note: Previous versions of this specification used bit-fields to construct the object type values. Beginning with version 1.2, this is no longer the case and no special significance is given to the bit positions within algorithm identifier values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_AES</td>
<td>0xA0000010</td>
</tr>
<tr>
<td>TEE_TYPE_DES</td>
<td>0xA0000011</td>
</tr>
<tr>
<td>TEE_TYPE_DES3</td>
<td>0xA0000013</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_MD5</td>
<td>0xA0000001</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA1</td>
<td>0xA0000002</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA224</td>
<td>0xA0000003</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA256</td>
<td>0xA0000004</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA384</td>
<td>0xA0000005</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SHA512</td>
<td>0xA0000006</td>
</tr>
<tr>
<td>TEE_TYPE_HMAC_SM3</td>
<td>0xA0000007</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_PUBLIC_KEY</td>
<td>0xA00000030</td>
</tr>
<tr>
<td>TEE_TYPE_RSA_KEYPAIR</td>
<td>0xA10000030</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_PUBLIC_KEY</td>
<td>0xA0000031</td>
</tr>
<tr>
<td>TEE_TYPE_DSA_KEYPAIR</td>
<td>0xA1000031</td>
</tr>
<tr>
<td>TEE_TYPE_DH_KEYPAIR</td>
<td>0xA1000032</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_PUBLIC_KEY</td>
<td>0xA0000041</td>
</tr>
<tr>
<td>TEE_TYPE_ECDSA_KEYPAIR</td>
<td>0xA1000041</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_PUBLIC_KEY</td>
<td>0xA0000042</td>
</tr>
<tr>
<td>TEE_TYPE_ECDH_KEYPAIR</td>
<td>0xA1000042</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_PUBLIC_KEY</td>
<td>0xA0000043</td>
</tr>
<tr>
<td>TEE_TYPE_ED25519_KEYPAIR</td>
<td>0xA1000043</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_PUBLIC_KEY</td>
<td>0xA0000044</td>
</tr>
<tr>
<td>TEE_TYPE_X25519_KEYPAIR</td>
<td>0xA1000044</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_PUBLIC_KEY</td>
<td>0xA0000045</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_DSA_KEYPAIR</td>
<td>0xA1000045</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_PUBLIC_KEY</td>
<td>0xA0000046</td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_KEP_KEYPAIR</td>
<td>0xA1000046</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_PUBLIC_KEY</td>
<td>0xA0000047</td>
</tr>
<tr>
<td>TEE_TYPE_SM2_PKE_KEYPAIR</td>
<td>0xA1000047</td>
</tr>
<tr>
<td>TEE_TYPE_SM4</td>
<td>0xA0000014</td>
</tr>
<tr>
<td>TEE_TYPE_GENERIC_SECRET</td>
<td>0xA000000</td>
</tr>
<tr>
<td>TEE_TYPE_CORRUPTED_OBJECT</td>
<td>0xA00000BE</td>
</tr>
<tr>
<td>TEE_TYPE_DATA</td>
<td>0xA00000BF</td>
</tr>
<tr>
<td>TEE_TYPE_ILLEGAL_VALUE</td>
<td>0xEFFFFFFF</td>
</tr>
<tr>
<td>Reserved for implementation-defined object handles</td>
<td>0xF0000000-0xF0FFFFFF</td>
</tr>
<tr>
<td>Reserved</td>
<td>All values not defined above.</td>
</tr>
</tbody>
</table>

Object types using implementation-specific algorithms are defined by the implementation.

TEE_TYPE_CORRUPTED_OBJECT is used solely in the deprecated TEE_GetObjectInfo function to indicate that the object on which it is being invoked has been corrupted in some way.

TEE_TYPE_DATA is used to represent objects which have no cryptographic attributes, just a data stream.

Note: TEE_TYPE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.
6.10.3 Optional Cryptographic Elements

This specification defines support for optional cryptographic elements as follows:

- NIST ECC curve definitions come from [NIST Re Cur].
- BSI ECC curve definitions come from [BSI TR 03111].
- Edwards ECC curve definitions from [X25519].
- SM2 curve definition from [SM2].

Identifiers which SHALL be used to identify optional cryptographic elements are listed in Table 6-14. The size parameter represents the length, in bits, of:

- The base field for elliptic curves.
- Not applicable for TEE_CRYPTO_ELEMENT_NONE.

In this version of the specification, a conforming implementation can support none, some, or all of the cryptographic elements listed in Table 6-14. The TEE_IsAlgorithmSupported function (see section 6.2.9) is provided to enable applications to determine whether a specific curve definition is supported.

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Generic</th>
<th>Identifier</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_CRYPTO_ELEMENT_NONE</td>
<td>-</td>
<td>Y</td>
<td>0x00000000</td>
<td>-</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_NIST_P192</td>
<td>NIST</td>
<td>Y</td>
<td>0x00000001</td>
<td>192 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_NIST_P224</td>
<td>NIST</td>
<td>Y</td>
<td>0x00000002</td>
<td>224 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_NIST_P256</td>
<td>NIST</td>
<td>Y</td>
<td>0x00000003</td>
<td>256 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_NIST_P384</td>
<td>NIST</td>
<td>Y</td>
<td>0x00000004</td>
<td>384 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_NIST_P521</td>
<td>NIST</td>
<td>Y</td>
<td>0x00000005</td>
<td>521 bits</td>
</tr>
<tr>
<td>Reserved for future NIST curves</td>
<td>-</td>
<td></td>
<td>0x00000006 - 0x000000FF</td>
<td></td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P160r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000010</td>
<td>160 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P192r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000012</td>
<td>192 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P224r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000013</td>
<td>224 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P256r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000014</td>
<td>256 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P320r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000015</td>
<td>320 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P384r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000016</td>
<td>384 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P512r1</td>
<td>BSI-R</td>
<td>Y</td>
<td>0x00000017</td>
<td>512 bits</td>
</tr>
<tr>
<td>Reserved for future BSI (R) curves</td>
<td>-</td>
<td></td>
<td>0x00000018 - 0x000001FF</td>
<td></td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P160t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000201</td>
<td>160 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P192t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000202</td>
<td>192 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P224t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000203</td>
<td>224 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P256t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000204</td>
<td>256 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P320t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000205</td>
<td>320 bits</td>
</tr>
<tr>
<td>Name</td>
<td>Source</td>
<td>Generic</td>
<td>Identifier</td>
<td>Size</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------</td>
<td>---------</td>
<td>---------------------</td>
<td>--------</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P384t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000206</td>
<td>384 bits</td>
</tr>
<tr>
<td>TEE_ECC_CURVE_BSI_P512t1</td>
<td>BSI-T</td>
<td>Y</td>
<td>0x00000207</td>
<td>512 bits</td>
</tr>
<tr>
<td>Reserved for future BSI (T) curves</td>
<td>-</td>
<td></td>
<td>0x00000208 - 0x000002FF</td>
<td></td>
</tr>
<tr>
<td>TEE_ECC_CURVE_25519</td>
<td>IETF</td>
<td>N</td>
<td>0x0000000000</td>
<td>256 bits</td>
</tr>
<tr>
<td>Reserved for future IETF curves</td>
<td>-</td>
<td></td>
<td>0x000000201 - 0x000002FF</td>
<td></td>
</tr>
<tr>
<td>TEE_ECC_CURVE_SM2</td>
<td>OCTA</td>
<td>N</td>
<td>0x00000000300</td>
<td>256 bits</td>
</tr>
<tr>
<td>Reserved for future curves defined by OCTA</td>
<td>-</td>
<td></td>
<td>0x000000301 - 0x000003FF</td>
<td></td>
</tr>
<tr>
<td>Reserved for future use</td>
<td>-</td>
<td></td>
<td>0x000000400 - 0x7FFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Implementation defined</td>
<td>-</td>
<td></td>
<td>0x80000000 - 0xFFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>

TEE_CRYPTO_ELEMENT_NONE is a special identifier which can be used when a function requires a value from Table 6-14, but no specific cryptographic element needs to be provided.

Backward Compatibility

If all of the NIST curves defined in Table 6-14 are supported by a Trusted OS, the implementation SHALL return true to queries of the deprecated property gpd.tee.cryptography.ecc (see section B.3), otherwise it SHALL return false to such queries.
### 6.11 Object or Operation Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Protection</th>
<th>Type</th>
<th>Format (Table 6-16)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ATTR_SECRET_VALUE</td>
<td>0xC00000000</td>
<td>Protected</td>
<td>Ref</td>
<td>binary</td>
<td>Used for all secret keys for symmetric ciphers, MACs, and HMACs</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_MODULUS</td>
<td>0xD0000130</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_RSA_PUBLIC_EXPONENT</td>
<td>0xD0000230</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_RSA_PRIVATE_EXPONENT</td>
<td>0xC0000330</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_RSA_PRIME1</td>
<td>0xC0000430</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>Usually referred to as (p).</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_PRIME2</td>
<td>0xC0000530</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(q)</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_EXPONENT1</td>
<td>0xC0000630</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(dp)</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_EXPONENT2</td>
<td>0xC0000730</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(dq)</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_COEFFICIENT</td>
<td>0xC0000830</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(iq)</td>
</tr>
<tr>
<td>TEE_ATTR_DSA_PRIME</td>
<td>0xD0001031</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(p)</td>
</tr>
<tr>
<td>TEE_ATTR_DSA_SUBPRIME</td>
<td>0xD0001131</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(q)</td>
</tr>
<tr>
<td>TEE_ATTR_DSA_BASE</td>
<td>0xD0001231</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(g)</td>
</tr>
<tr>
<td>TEE_ATTR_DSA_PUBLIC_VALUE</td>
<td>0xD0000131</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(y)</td>
</tr>
<tr>
<td>TEE_ATTR_DSA_PRIVATE_VALUE</td>
<td>0xC0000231</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(x)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_PRIME</td>
<td>0xD0001032</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(p)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_SUBPRIME</td>
<td>0xD0001132</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(q)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_BASE</td>
<td>0xD0001232</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(g)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_X_BITS</td>
<td>0xF0001332</td>
<td>Public</td>
<td>Value</td>
<td>int</td>
<td>(l)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_PUBLIC_VALUE</td>
<td>0xD0000132</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td>(y)</td>
</tr>
<tr>
<td>TEE_ATTR_DH_PRIVATE_VALUE</td>
<td>0xC0000232</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td>(x)</td>
</tr>
<tr>
<td>TEE_ATTR_RSA_OAEP_LABEL</td>
<td>0xD0000930</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_RSA_PSS_SALT_LENGTH</td>
<td>0xF0000A30</td>
<td>Public</td>
<td>Value</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
<td>0xD0000141</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
<td>0xD0000241</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
<td>0xC0000341</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Protection</td>
<td>Type</td>
<td>Format (Table 6-16)</td>
<td>Comment</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
<td>------------</td>
<td>--------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_ATTR_ECC_CURVE</td>
<td>0xF0000441</td>
<td>Public</td>
<td>Value</td>
<td>int</td>
<td>Identifier value from Table 6-14</td>
</tr>
<tr>
<td>TEE_ATTR_ED25519_CTX</td>
<td>0xD0000643</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string, per algorithm definition in [Ed25519]</td>
</tr>
<tr>
<td>TEE_ATTR_ED25519_PUBLIC_VALUE</td>
<td>0xD0000743</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string, per algorithm definition in [Ed25519]</td>
</tr>
<tr>
<td>TEE_ATTR_ED25519_PRIVATE_VALUE</td>
<td>0xC0000843</td>
<td>Protected</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string, per algorithm definition in [Ed25519]</td>
</tr>
<tr>
<td>TEE_ATTR_ED25519_PH</td>
<td>0xF0000543</td>
<td>Public</td>
<td>Value</td>
<td>int</td>
<td>Octet string, containing identifier of initiator</td>
</tr>
<tr>
<td>TEE_ATTR_X25519_PUBLIC_VALUE</td>
<td>0xD0000944</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string, per algorithm definition in [X25519]</td>
</tr>
<tr>
<td>TEE_ATTR_X25519_PRIVATE_VALUE</td>
<td>0xC0000A44</td>
<td>Protected</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string, per algorithm definition in [X25519]</td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_X</td>
<td>0xD0000146</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PUBLIC_VALUE_Y</td>
<td>0xD0000246</td>
<td>Public</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_ECC_PRIVATE_VALUE</td>
<td>0xD0000346</td>
<td>Protected</td>
<td>Ref</td>
<td>bignum</td>
<td></td>
</tr>
<tr>
<td>TEE_ATTR_SM2_ID_INITIATOR</td>
<td>0xD0000446</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string containing identifier of initiator</td>
</tr>
<tr>
<td>TEE_ATTR_SM2_ID_RESPONDER</td>
<td>0xD0000546</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string containing identifier of responder</td>
</tr>
<tr>
<td>TEE_ATTR_SM2_KEP_USER</td>
<td>0xF0000646</td>
<td>Public</td>
<td>value</td>
<td>int</td>
<td>zero mean initiator role, non-zero mean responder</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Protection</td>
<td>Type</td>
<td>Format (Table 6-16)</td>
<td>Comment</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------</td>
<td>------------</td>
<td>------</td>
<td>---------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TEE_ATTR_SM2_KEP_CONFIRMATION_IN</td>
<td>0xD0000746</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string containing value from other peer</td>
</tr>
<tr>
<td>TEE_ATTR_SM2_KEP_CONFIRMATION_OUT</td>
<td>0xD0000846</td>
<td>Public</td>
<td>Ref</td>
<td>binary</td>
<td>Octet string containing value from the caller</td>
</tr>
</tbody>
</table>

Table 6-16: Attribute Format Definitions

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>An array of unsigned octets</td>
</tr>
<tr>
<td>bignum</td>
<td>An unsigned bignum in big-endian binary format. Leading zero bytes are allowed.</td>
</tr>
<tr>
<td>int</td>
<td>Values attributes represented in a single integer returned/read from argument a.</td>
</tr>
</tbody>
</table>

Additional attributes may be defined for use with implementation defined algorithms.

Implementer’s Notes

Selected bits of the attribute identifiers are explained in Table 6-17.

Table 6-17: Partial Structure of Attribute Identifier

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit [29]</td>
<td>Defines whether the attribute is a buffer or value attribute</td>
<td>0: buffer attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: value attribute</td>
</tr>
<tr>
<td>Bit [28]</td>
<td>Defines whether the attribute is protected or public</td>
<td>0: protected attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: public attribute</td>
</tr>
</tbody>
</table>

A protected attribute cannot be extracted unless the object has the TEE_USAGE_EXTRACTABLE flag.

Table 6-18 defines constants that reflect setting bit [29] and bit [28], respectively, intended to help decode attribute identifiers.

Table 6-18: Attribute Identifier Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ATTR_FLAG_VALUE</td>
<td>0x20000000</td>
</tr>
<tr>
<td>TEE_ATTR_FLAG_PUBLIC</td>
<td>0x10000000</td>
</tr>
</tbody>
</table>
7 Time API

This API provides access to three sources of time:

- **System Time**
  - The origin of this system time is arbitrary and implementation-dependent. Different TA instances may even have different system times. The only guarantee is that the system time is not reset or rolled back during the life of a given TA instance, so it can be used to compute time differences and operation deadlines, for example. The system time SHALL NOT be affected by transitions through low power states.
  - System time is related to the function `TEE_Wait`, which waits for a given timeout or cancellation.
  - The level of trust that a Trusted Application can put on the system time is implementation defined but can be discovered programmatically by querying the implementation property `gpd.tee.systemTime.protectionLevel`. Typically, an implementation may rely on the REE timer (protection level 100) or on a dedicated secure timer hardware (protection level 1000).
  - System time SHALL advance within plus or minus 15% of the passage of real time in the outside world including while the device is in low power states, to ensure that appropriate security levels are maintained when, for example, system time is used to implement dictionary attack protection. This accuracy also applies to timeout values where they are specified in individual routines.

- **TA Persistent Time**, a real-time source of time
  - The origin of this time is set individually by each Trusted Application and SHALL persist across reboots.
  - The level of trust on the TA Persistent Time can be queried through the property `gpd.tee.TAPersistentTime.protectionLevel`.

- **REE Time**
  - This is as trusted as the REE itself and may also be tampered by the user.

All time functions use a millisecond resolution and split the time in the two fields of the structure `TEE_Time`:

When used to return a time value, this structure can represent times up to 06:28:15 UTC on Sun, 7 February 2106.

7.1 Data Types

7.1.1 TEE_Time

Since: TEE Internal API v1.0

```c
typedef struct {
    uint32_t seconds;
    uint32_t millis;
} TEE_Time;
```

When used to return a time value, this structure can represent times up to 06:28:15 UTC on Sun, 7 February 2106.
7.2 Time Functions

7.2.1 TEE_GetSystemTime

Since: TEE Internal API v1.0

```c
void TEE_GetSystemTime(
    [out] TEE_Time* time );
```

Description

The `TEE_GetSystemTime` function retrieves the current system time. The system time has an arbitrary implementation-defined origin that can vary across TA instances. The minimum guarantee is that the system time SHALL be monotonic for a given TA instance. Implementations are allowed to use the REE timers to implement this function but may also better protect the system time. A TA can discover the level of protection implementation by querying the implementation property `gpd.tee.systemTime.protectionLevel`. Possible values are listed in Table 7-1.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>System time based on REE-controlled timers. Can be tampered by the REE. The implementation SHALL still guarantee that the system time is monotonic, i.e. successive calls to <code>TEE_GetSystemTime</code> SHALL return increasing values of the system time.</td>
</tr>
<tr>
<td>1000</td>
<td>System time based on a TEE-controlled secure timer. The REE cannot interfere with the system time. It may still interfere with the scheduling of TEE tasks, but is not able to hide delays from a TA calling <code>TEE_GetSystemTime</code>.</td>
</tr>
</tbody>
</table>

Parameters

- time: Filled with the number of seconds and milliseconds since midnight on January 1, 1970, UTC

Specification Number: 10 Function Number: 0x1402

Panic Reasons

- If the implementation detects any error.
7.2.2 TEE_Wait

Since: TEE Internal API v1.0

```c
TEE_Result TEE_Wait( uint32_t timeout );
```

**Description**

The TEE_Wait function waits for the specified number of milliseconds or waits forever if `timeout` equals `TEE_TIMEOUT_INFINITE (0xFFFFFFFF)`.

When this function returns success, the implementation SHALL guarantee that the difference between two calls to `TEE_GetSystemTime` before and after the call to TEE_Wait is greater than or equal to the requested timeout. However, there may be additional implementation-dependent delays due to the scheduling of TEE tasks.

This function is cancellable, i.e. if the current task’s cancelled flag is set and the TA has unmasked the effects of cancellation, then this function returns earlier than the requested timeout with the return code `TEE_ERROR_CANCEL`. See section 4.10 for more details about cancellations.

**Parameters**

- `timeout`: The number of milliseconds to wait, or `TEE_TIMEOUT_INFINITE`

**Specification Number**: 10  **Function Number**: 0x1405

**Return Code**

- `TEE_SUCCESS`: On success.
- `TEE_ERROR_CANCEL`: If the wait has been cancelled.

**Panic Reasons**

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
7.2.3 TEE_GetTAPersistentTime

Since: TEE Internal API v1.0

```c
TEE_Result TEE_GetTAPersistentTime(
    [out] TEE_Time* time );
```

Description

The TEE_GetTAPersistentTime function retrieves the persistent time of the Trusted Application, expressed as a number of seconds and milliseconds since the arbitrary origin set by calling TEE_SetTAPersistentTime.

This function can return the following statuses (as well as other status values discussed in “Return Code”):

- TEE_SUCCESS means the persistent time is correctly set and has been retrieved into the parameter time.
- TEE_ERROR_TIME_NOT_SET is the initial status and means the persistent time has not been set. The Trusted Application SHALL set its persistent time by calling the function TEE_SetTAPersistentTime.
- TEE_ERROR_TIME_NEEDS_RESET means the persistent time has been set but may have been corrupted and SHALL no longer be trusted. In such a case it is recommended that the Trusted Application resynchronize the persistent time by calling the function TEE_SetTAPersistentTime.

Initially the time status is TEE_ERROR_TIME_NOT_SET. Once a Trusted Application has synchronized its persistent time by calling TEE_SetTAPersistentTime, the status can be TEE_SUCCESS or TEE_ERROR_TIME_NEEDS_RESET. Once the status has become TEE_ERROR_TIME_NEEDS_RESET, it will keep this status until the persistent time is re-synchronized by calling TEE_SetTAPersistentTime.

Figure 7-1 shows the state machine of the persistent time status.

![Figure 7-1: Persistent Time Status State Machine](image)

The meaning of the status TEE_ERROR_TIME_NEEDS_RESET depends on the protection level provided by the hardware implementation and the underlying real-time clock (RTC). This protection level can be queried by retrieving the implementation property `gpd.tee.TAPersistentTime.protectionLevel`, which can have one of the values listed in Table 7-2.

---

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.

The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
Table 7-2: Values of the gpd.tee.TAPersistentTime.protectionLevel Property

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Persistent time based on an REE-controlled real-time clock and on the TEE Trusted Storage for the storage of origins. The implementation SHALL guarantee that rollback of persistent time is detected to the fullest extent allowed by the Trusted Storage.</td>
</tr>
<tr>
<td>1000</td>
<td>Persistent time based on a TEE-controlled real-time clock and the TEE Trusted Storage. The real-time clock SHALL be out of reach of software attacks from the REE. Users may still be able to provoke a reset of the real-time clock but this SHALL be detected by the Implementation.</td>
</tr>
</tbody>
</table>

The number of seconds in the TA Persistent Time may exceed the range of the uint32_t integer type. In this case, the function SHALL return the error TEE_ERROR_OVERFLOW, but still computes the TA Persistent Time as specified above, except that the number of seconds is truncated to 32 bits before being written to time->seconds. For example, if the Trusted Application sets its persistent time to $2^{32} - 100$ seconds, then after 100 seconds, the TA Persistent Time is $2^{32}$, which is not representable with a uint32_t. In this case, the function TEE_GetTAPersistentTime SHALL return TEE_ERROR_OVERFLOW and set time->seconds to 0 (which is $2^{32}$ truncated to 32 bits).

Parameters

- time: A pointer to the TEE_Time structure to be set to the current TA Persistent Time. If an error other than TEE_ERROR_OVERFLOW is returned, this structure is filled with zeroes.

Specification Number: 10  Function Number: 0x1403

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_TIME_NOT_SET
- TEE_ERROR_TIME_NEEDS_RESET
- TEE_ERROR_OVERFLOW: The number of seconds in the TA Persistent Time overflows the range of a uint32_t. The field time->seconds is still set to the TA Persistent Time truncated to 32 bits (i.e. modulo $2^{32}$).
- TEE_ERROR_OUT_OF_MEMORY: If not enough memory is available to complete the operation

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
7.2.4 TEE_SetTAPersistentTime

Since: TEE Internal API v1.0

```c
TEE_Result TEE_SetTAPersistentTime(
    [in] TEE_Time* time);
```

**Description**

The `TEE_SetTAPersistentTime` function sets the persistent time of the current Trusted Application. Only the persistent time for the current Trusted Application is modified, not the persistent time of other Trusted Applications. This will affect all instances of the current Trusted Application. The modification is atomic and persistent across device reboots.

**Parameters**

- time: Filled with the persistent time of the current TA

**Specification Number:** 10  **Function Number:** 0x1404

**Return Code**

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OUT_OF_MEMORY: If not enough memory is available to complete the operation
- TEE_ERROR_STORAGE_NO_SPACE: If insufficient storage space is available to complete the operation

**Panic Reasons**

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
7.2.5 TEE_GetREETime

Since: TEE Internal API v1.0

```c
void TEE_GetREETime(
    [out] TEE_Time* time);
```

Description

The TEE_GetREETime function retrieves the current REE system time. This function retrieves the current time as seen from the point of view of the REE, expressed in the number of seconds since midnight on January 1, 1970, UTC.

In normal operation, the value returned SHOULD correspond to the real time, but it SHOULD NOT be considered as trusted, as it may be tampered by the user or the REE software.

Parameters

- time: Filled with the number of seconds and milliseconds since midnight on January 1, 1970, UTC

Specification Number: 10 Function Number: 0x1401

Panic Reasons

- If the Implementation detects any error.
8 TEE Arithmetical API

8.1 Introduction

All asymmetric cryptographic functions are implemented by using arithmetical functions, where operands are typically elements of finite fields or in mathematical structures containing finite field elements. The Cryptographic Operations API hides the complexity of the mathematics that is behind these operations. A developer who needs some cryptographic service does not need to know anything about the internal implementation.

However in practice developer may face the following difficulties: the API does not support the desired algorithm; or the API supports the algorithm, but with the wrong encodings, options, etc. The purpose of the TEE Arithmetical API is to provide building blocks so that the developer can implement missing asymmetric algorithms. In other words the arithmetical API can be used to implement a plug-in into the Cryptographic Operations API. Furthermore and to ease the design of speed efficient algorithms, the arithmetical API also gives access to a Fast Modular Multiplication primitive, referred to as FMM.

This specification mandates that all functions within the TEE Arithmetical API SHALL work when input and output TEE_BigInt values are within the interval \([-2^M + 1, 2^M - 1]\) (limits included), where M is an implementation-defined number of bits. Every Implementation SHALL ensure that M is at least 2048. The exact value of M can be retrieved as the implementation property \( \text{gpd.tee.arith.maxBigIntSize} \).

Throughout this chapter:

- The notation “n-bit integer” refers to an integer that can take values in the range \([-2^n + 1, 2^n - 1]\), including limits.
- The notation “magnitude(src)” denotes the minimum number of required bits to represent the absolute value of the big integer src in a natural binary representation. The developer may query the magnitude of a big integer by using the function TEE_BigIntGetBitCount(src), as described in section 8.7.5.

8.2 Error Handling and Parameter Checking

This low level arithmetical API performs very few checks on the parameters given to the functions. Most functions will return undefined results when called inappropriately but will not generate any error return codes.

Some functions in the API MAY work for inputs larger than indicated by the implementation property gpd.tee.arith.maxBigIntSize. This is however not guaranteed. When a function does not support a given bigInt size beyond this limit, it SHALL panic and not produce invalid results.
8.3 Data Types

This specification version has three data types for the arithmetical operations. These are TEE_BigInt, TEE_BigIntFMM, and TEE_BigIntFMMContext. Before using the arithmetic operations, the TA developer SHALL allocate and initialize the memory for the input and output operands. This API provides entry points to determine the correct sizes of the needed memory allocations.

8.3.1 TEE_BigInt

The TEE_BigInt type is a placeholder for the memory structure of the TEE core internal representation of a large multi-precision integer.

Since: TEE Internal API v1.0

typedef uint32_t TEE_BigInt;

The following constraints are put on the internal representation of the TEE_BigInt:

1) The size of the representation SHALL be a multiple of 4 bytes.
2) The extra memory within the representation to store metadata SHALL NOT exceed 8 bytes.
3) The representation SHALL be stored 32-bit aligned in memory.

Exactly how a multi-precision integer is represented internally is implementation-specific but it SHALL be stored within a structure of the maximum size given by the macro TEE_BigIntSizeInU32 (see section 8.4.1).

By defining a TEE_BigInt as a uint32_t for the TA, we allow the TA developer to allocate static space for multiple occurrences of TEE_BigInt at compile time which obey constraints 1 and 3. The allocation can be done with code similar to this:

```c
uint32_t twoints[2 * TEE_BigIntSizeInU32(1024)];
TEE_BigInt* first = twoints;
TEE_BigInt* second = twoints + TEE_BigIntSizeInU32(1024);
```

/* Or if we do it dynamically */

```c
TEE_BigInt* op1;
op1 = TEE_Malloc(TEE_BigIntSizeInU32(1024) * sizeof(TEE_BigInt),
                TEE_MALLOC_NO_FILL | TEE_MALLOC_NO_SHARE);
/* use op1 */
TEE_Free(op1);
```

Conversions from an external representation to the internal TEE_BigInt representation and vice versa can be done by using functions from section 8.6.

Most functions in the TEE Arithmetical API take one or more TEE_BigInt pointers as parameters; for example, func(TEE_BigInt *op1, TEE_BigInt *op2). When describing the parameters and what the function does, this specification will refer to the integer represented in the structure to which the pointer `op1` points, by simply writing `op1`. It will be clear from the context when the pointer value is referred to and when the integer value is referred to.

Since the internal representation of TEE_BigInt is implementation-specific, TA implementers SHALL pass the first address of a TEE_BigInt structure to functions that use them. A TEE_BigInt pointer that points to a location other than the start of a TEE_BigInt is a programmer error.
8.3.2 TEE_BigIntFMMContext

Usually, such a fast modular multiplication requires some additional data or derived numbers. That extra data is stored in a context that SHALL be passed to the fast modular multiplication function. The TEE_BigIntFMMContext is a placeholder for the TEE core internal representation of the context that is used in the fast modular multiplication operation.

Since: TEE Internal API v1.0

```
typedef uint32_t TEE_BigIntFMMContext;
```

The following constraints are put on the internal representation of the TEE_BigIntFMMContext:

1) The size of the representation SHALL be a multiple of 4 bytes.
2) The representation SHALL be stored 32-bit aligned in memory.

Exactly how this context is represented internally is implementation-specific but it SHALL be stored within a structure of the size given by the function TEE_BigIntFMMContextSizeInU32 (see section 8.4.2).

Similarly to TEE_BigInt, we expose this type as a uint32_t to the TA, which helps TEE_Malloc to align the structure correctly when allocating space for a TEE_BigIntFMMContext*.

8.3.3 TEE_BigIntFMM

Some implementations may have support for faster modular multiplication algorithms such as Montgomery or Barrett multiplication for use in modular exponentiation. Typically, those algorithms require some transformation of the input before the multiplication can be carried out. The TEE_BigIntFMM is a placeholder for the memory structure that holds an integer in such a transformed representation.

Since: TEE Internal API v1.0

```
typedef uint32_t TEE_BigIntFMM;
```

The following constraints are put on the internal representation of the TEE_BigIntFMM:

1) The size of the representation SHALL be a multiple of 4 bytes.
2) The representation SHALL be stored 32-bit aligned in memory.

Exactly how this transformed representation is stored internally is implementation-specific but it SHALL be stored within a structure of the maximum size given by the function TEE_BigIntFMMSizeInU32 (see section 8.4.3).

Similarly to TEE_BigInt, we expose this type as a uint32_t to the TA, which helps TEE_Malloc to align the structure correctly when allocating space for a TEE_BigIntFMM*.
8.4 Memory Allocation and Size of Objects

It is the responsibility of the Trusted Application to allocate and free memory for all TEE arithmetical objects, including all operation contexts, used in the Trusted Application. Once the arithmetical objects are allocated, the functions in the TEE Arithmetical API will never fail because of out-of-resources.

**TEE implementer's note:** Implementations of the TEE Arithmetical API SHOULD utilize memory from one or more pre-allocated pools to store intermediate results during computations to ensure that the functions do not fail because of lack of resources. All memory resources used internally SHALL be thread-safe. Such a pool of scratch memory could be:

- Internal memory of a hardware accelerator module
- Allocated from mutex protected system-wide memory
- Allocated from the heap of the TA instance, i.e. by using `TEE_Malloc` or `TEE_Realloc`

If the implementation uses a memory pool of temporary storage for intermediate results or if it uses hardware resources such as accelerators for some computations, the implementation SHALL either wait for the resource to become available or, for example in case of a busy hardware accelerator, resort to other means such as a software implementation.

8.4.1 TEE_BigIntSizeInU32

Since: TEE Internal API v1.0

```c
#define TEE_BigIntSizeInU32(n) ((((n)+31)/32)+2)
```

**Description**

The `TEE_BigIntSizeInU32` macro calculates the size of the array of `uint32_t` values needed to represent an n-bit integer. This is defined as a macro (thereby mandating the maximum size of the internal representation) rather than as a function so that TA developers can use the macro in a static compile-time declaration of an array. Note that the implementation of the internal arithmetic functions assumes that memory pointed to by the `TEE_BigInt*` is 32-bit aligned.

**Parameters**

- `n`: maximum number of bits to be representable
8.4.2  TEE_BigIntFMMContextSizeInU32

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
size_t TEE_BigIntFMMContextSizeInU32( size_t modulusSizeInBits );
```

Description

The `TEE_BigIntFMMContextSizeInU32` function returns the size of the array of `uint32_t` values needed to represent a fast modular context using a given modulus size. This function SHALL never fail.

Parameters

- `modulusSizeInBits`: Size of modulus in bits

Specification Number: 10  Function Number: 0x1502

Return Value

Number of bytes needed to store a `TEE_BigIntFMMContext` given a modulus of length `modulusSizeInBits`.

Panic Reasons

- If the Implementation detects any error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `modulusSizeInBits`. 
8.4.3 TEE_BigIntFMMSizeInU32

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
size_t TEE_BigIntFMMSizeInU32( size_t modulusSizeInBits );
```

Description

The TEE_BigIntFMMSizeInU32 function returns the size of the array of `uint32_t` values needed to represent an integer in the fast modular multiplication representation, given the size of the modulus in bits. This function SHALL never fail.

Normally from a mathematical point of view, this function would have needed the context to compute the exact required size. However, it is beneficial to have a function that does not take an initialized context as a parameter and thus the implementation may overstate the required memory size. It is nevertheless likely that a given implementation of the fast modular multiplication can calculate a very reasonable upper-bound estimate based on the modulus size.

Parameters

- `modulusSizeInBits`: Size of modulus in bits

Specification Number: 10  Function Number: 0x1501

Return Value

Number of bytes needed to store a `TEE_BigIntFMM` given a modulus of length `modulusSizeInBits`.

Panic Reasons

- If the implementation detects any error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `modulusSizeInBits`. 
8.5 Initialization Functions

These functions initialize the arithmetical objects after the TA has allocated the memory to store them. The Trusted Application SHALL call the corresponding initialization function after it has allocated the memory for the arithmetical object.

8.5.1 TEE_BigIntInit

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_BigIntInit(
    [out] TEE_BigInt *bigInt,
    size_t len);
```

Description

The TEE_BigIntInit function initializes `bigInt` and sets its represented value to zero. This function assumes that `bigInt` points to a memory area of `len` `uint32_t`. This can be done for example with the following memory allocation:

```c
TEE_BigInt *a;
size_t len;
len = (size_t) TEE_BigIntSizeInU32(bitSize);
a = (TEE_BigInt*)TEE_Malloc(len*sizeof(TEE_BigInt), TEE_MALLOC_NO_FILL|TEE_MALLOC_NO_SHARE);
TEE_BigIntInit(a, len);
```

Parameters

- `bigInt`: A pointer to the TEE_BigInt to be initialized
- `len`: The size in `uint32_t` of the memory pointed to by `bigInt`

Specification Number: 10 Function Number: 0x1601

Panic Reasons

- If the Implementation detects any error.
- If the provided value of `len` is larger than the number of bytes needed to represent `gpd.tee.arith.maxBigIntSize`.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the `len`. Versions of TEE Internal Core API prior to v1.2 might not panic for large values of `len`.

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
8.5.2 TEE_BigIntInitFMMContext1

Since: TEE Internal API v1.2.

---

Since: TEE Internal API v1.2.

TEE_Result TEE_BigIntInitFMMContext1(
    [out] TEE_BigIntFMMContext *context,
    size_t len,
    [in] TEE_BigInt *modulus);

---

Description

This function replaces the TEE_BigIntInitFMMContext function, whose use is deprecated.

The TEE_BigIntInitFMMContext function calculates the necessary prerequisites for the fast modular multiplication and stores them in a context. This function assumes that context points to a memory area of len uint32_t. This can be done for example with the following memory allocation:

```c
TEE_BigIntFMMContext* ctx;
size_t len = (size_t) TEE_BigIntFMMContextSizeInU32(bitsize);
ctx = (TEE_BigIntFMMContext*)TEE_Malloc(len * sizeof(TEE_BigIntFMContext),
        TEE_MALLOC_NO_FILL | TEE_MALLOC_NO_SHARE);
/*Code for initializing modulus*/

TEE_BigIntInitFMMContext(ctx, len, modulus);
```

Even though a fast multiplication might be mathematically defined for any modulus, normally there are restrictions in order for it to be fast on a computer. This specification mandates that all implementations SHALL work for all odd moduli larger than 2 and less than 2 to the power of the implementation defined property gpd.tee.arith.maxBigIntSize.

It is not required that even moduli be supported. Common usage of this function will not make use of even moduli and so for performance reasons a technique without full even moduli support MAY be used. For this reason, partial or complete even moduli support are optional, and if an implementation will not be able to provide a result for a specific case of even moduli then it shall return TEE_ERROR_NOT_SUPPORTED.

Parameters

- **context**: A pointer to the TEE_BigIntFMMContext to be initialized
- **len**: The size in uint32_t of the memory pointed to by context
- **modulus**: The modulus, an odd integer larger than 2 and less than 2 to the power of gpd.tee.arith.maxBigIntSize

---

Specification Number: 10  Function Number: 0x1604

Return Code

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_NOT_SUPPORTED**: The underlying implementation is unable to perform the operation on a particular modulus value. This may only be returned for even moduli inside the valid range, outside that range the described PANIC will occur.

Panic Reasons

- If the Implementation detects any error.
- If the provided value of modulus is either less than two, or larger than or equal to 2^gpd.tee.arith.maxBigIntSize.
8.5.3 TEE_BigIntInitFMM

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_BigIntInitFMM(
    [in]   TEE_BigIntFMM *bigIntFMM,
    size_t len );
```

**Description**

The `TEE_BigIntInitFMM` function initializes `bigIntFMM` and sets its represented value to zero. This function assumes that `bigIntFMM` points to a memory area of `len` `uint32_t`. This can be done for example with the following memory allocation:

```c
TEE_BigIntFMM *a;
size_t len;
len = (size_t) TEE_BigIntFMMSizeInU32(modulusSizeInBits);
a = (TEE_BigIntFMM *)TEE_Malloc(len * sizeof(TEE_BigIntFMM),
    TEE_MALLOC_NO.FILL | TEE_MALLOC_NO.SHARE );
TEE_BigIntInitFMM(a, len);
```

**Parameters**

- `bigIntFMM`: A pointer to the `TEE_BigIntFMM` to be initialized
- `len`: The size in `uint32_t` of the memory pointed to by `bigIntFMM`

**Specification Number:** 10  **Function Number:** 0x1602

**Panic Reasons**

- If the implementation detects any error.
- If the provided value of `len` is larger than the number of bytes needed to represent `gpd.tee.arith.maxBigIntSize`.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `len`. Versions of TEE Internal Core API prior to v1.2 might not panic for large values of `len`. 
8.6 Converter Functions

TEE_BigInt contains the internal representation of a multi-precision integer. However in many use cases some integer data comes from external sources or integers; for example, a local device gets an ephemeral Diffie-Hellman public key during a key agreement procedure. In this case the ephemeral key is expected to be in octet string format, which is a big-endian radix 256 representation for unsigned numbers. For example 0x123456789abcdef has the following octet string representation:

\{0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef\}

This section provides functions to convert to and from such alternative representations.

8.6.1 TEE_BigIntConvertFromOctetString

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```
TEE_Result TEE_BigIntConvertFromOctetString(
    [out] TEE_BigInt *dest,
    [inbuf] uint8_t    *buffer, size_t bufferLen,
    int32_t     sign );
```

Description

The TEE_BigIntConvertFromOctetString function converts a bufferLen byte octet string buffer into a TEE_BigInt format. The octet string is in most significant byte first representation. The input parameter sign will set the sign of dest. It will be set to negative if sign < 0 and to positive if sign >= 0.

Parameters

- dest: Pointer to a TEE_BigInt to hold the result
- buffer: Pointer to the buffer containing the octet string representation of the integer
- bufferLen: The length of *buffer in bytes
- sign: The sign of dest is set to the sign of sign.

Specification Number: 10  Function Number: 0x1701

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OVERFLOW: If memory allocation for the dest is too small

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the bufferLen.
8.6.2 TEE_BigIntConvertToOctetString

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
TEE_Result TEE_BigIntConvertToOctetString(
    [outbuf] void* buffer, size_t *bufferLen,
    [in]    TEE_BigInt  *bigInt);
```

**Description**

The `TEE_BigIntConvertToOctetString` function converts the absolute value of an integer in `TEE_BigInt` format into an octet string. The octet string is written in a most significant byte first representation.

**Parameters**

- `buffer, bufferLen`: Output buffer where converted octet string representation of the integer is written
- `bigInt`: Pointer to the integer that will be converted to an octet string

**Specification Number: 10  Function Number: 0x1703**

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is too small to contain the octet string

**Panic Reasons**

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.

**Backward Compatibility**

TEE Internal Core API v1.1 used a different type for the `bufferLen`. 
8.6.3 TEE_BigIntConvertFromS32

Since: TEE Internal API v1.0

```c
void TEE_BigIntConvertFromS32(
    [out] TEE_BigInt *dest,
    int32_t shortVal);
```

Description

The `TEE_BigIntConvertFromS32` function sets `dest` to the value `shortVal`.

Parameters

- `dest`: Pointer to the start of an array reference by `TEE_BigInt *` into which the result is stored.
- `shortVal`: Input value

Specification Number: 10  Function Number: 0x1702

Result Size

The result SHALL point to a memory allocation which is at least large enough for holding a 32-bit signed value in a `TEE_BigInt` structure.

Panic Reasons

- If the memory pointed to by `dest` has not been initialized as a `TEE_BigInt` capable of holding at least a 32-bit value.
- If the Implementation detects any error.

Backward Compatibility

Versions of TEE Internal Core API prior to v1.2 did not include the clarification of panic due to an uninitialized `dest` pointer.
8.6.4 TEE_BigIntConvertToS32

Since: TEE Internal API v1.0

TEE_Result TEE_BigIntConvertToS32(
    [out] int32_t *dest,
    [in] TEE_BigInt *src);

Description
The TEE_BigIntConvertToS32 function sets *dest to the value of src, including the sign of src. If src
does not fit within an int32_t, the value of *dest is undefined.

Parameters
- dest: Pointer to an int32_t to store the result
- src: Pointer to the input value

Specification Number: 10  Function Number: 0x1704

Return Code
- TEE_SUCCESS: In case of success.
- TEE_ERROR_OVERFLOW: If src does not fit within an int32_t

Panic Reasons
- If the implementation detects any error associated with this function which is not explicitly associated
  with a defined return code for this function.
8.7 Logical Operations

8.7.1 TEE_BigIntCmp

Since: TEE Internal API v1.0

```c
int32_t TEE_BigIntCmp(
    [in]  TEE_BigInt *op1,
    [in]  TEE_BigInt *op2 );
```

**Description**
The `TEE_BigIntCmp` function checks whether `op1 > op2`, `op1 == op2`, or `op1 < op2`.

**Parameters**
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

**Specification Number**: 10  **Function Number**: 0x1801

**Return Value**
This function returns a negative number if `op1 < op2`, 0 if `op1 == op2`, and a positive number if `op1 > op2`.

**Panic Reasons**
- If the Implementation detects any error.

8.7.2 TEE_BigIntCmpS32

Since: TEE Internal API v1.0

```c
int32_t TEE_BigIntCmpS32(  
    [in]  TEE_BigInt *op,
    int32_t shortVal );
```

**Description**
The `TEE_BigIntCmpS32` function checks whether `op > shortVal`, `op == shortVal`, or `op < shortVal`.

**Parameters**
- `op`: Pointer to the first operand
- `shortVal`: Pointer to the second operand

**Specification Number**: 10  **Function Number**: 0x1802

**Return Value**
This function returns a negative number if `op < shortVal`, 0 if `op == shortVal`, and a positive number if `op > shortVal`.

**Panic Reasons**
- If the Implementation detects any error.
8.7.3 TEE_BigIntShiftRight

Since: TEE Internal API v1.0 – See Backward Compatibility note below.

```c
void TEE_BigIntShiftRight(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op,
    size_t bits);
```

Description

The TEE_BigIntShiftRight function computes \(|\text{dest}| = \text{op} \gg \text{bits}| and dest will have the same sign as op.\(^5\) If bits is greater than the bit length of op then the result is zero. dest and op MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- dest: Pointer to TEE_BigInt to hold the shifted result
- op: Pointer to the operand to be shifted
- bits: Number of bits to shift

Specification Number: 10  Function Number: 0x1805

Panic Reasons

- If the Implementation detects any error.

Backward Compatibility

TEE Internal Core API v1.1 used a different type for the bits.

---

\(^5\) The notation \(|x|\) means the absolute value of \(x\).
8.7.4 TEE_BigIntGetBit

Since: TEE Internal API v1.0

```c
bool TEE_BigIntGetBit(
    [in] TEE_BigInt *src,
    uint32_t    bitIndex );
```

**Description**

The TEE_BigIntGetBit function returns the bitIndexth bit of the natural binary representation of |src|. A true return value indicates a “1” and a false return value indicates a “0” in the bitIndexth position. If bitIndex is larger than the number of bits in op, the return value is false, thus indicating a “0”.

**Parameters**

- src: Pointer to the integer
- bitIndex: The offset of the bit to be read, starting at offset 0 for the least significant bit

**Specification Number:** 10  **Function Number:** 0x1803

**Return Value**

The Boolean value of the bitIndexth bit in |src|. True represents a “1” and false represents a “0”.

**Panic Reasons**

- If the Implementation detects any error.

8.7.5 TEE_BigIntGetBitCount

Since: TEE Internal API v1.0

```c
uint32_t TEE_BigIntGetBitCount(
    [in] TEE_BigInt *src );
```

**Description**

The TEE_BigIntGetBitCount function returns the number of bits in the natural binary representation of |src|; that is, the magnitude of src.

**Parameters**

- src: Pointer to the integer

**Specification Number:** 10  **Function Number:** 0x1804

**Return Value**

The number of bits in the natural binary representation of |src|. If src equals zero, it will return 0.

**Panic Reasons**

- If the Implementation detects any error.
8.7.6 TEE_BigIntSetBit

Since: TEE Internal Core API v1.2

```c
TEE_Result TEE_BigIntSetBit(
    [inout] TEE_BigInt *op,
    uint32_t bitIndex,
    bool value);
```

Description

The TEE_BigIntSetBit function sets the bitIndexth bit of the natural binary representation of |op| to 1 or 0, depending on the parameter value. If value is true the bit will be set, and if value is false the bit will be cleared. If bitIndex is larger than the number of bits in op, the function will return an overflow error.

Parameters

- op: Pointer to the integer
- bitIndex: The offset of the bit to be set, starting at offset 0 for the least significant bit.
- value: The bit value to set where true represents a “1” and false represents a “0”.

Specification Number: 10   Function Number: 0x1806

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OVERFLOW: If the bitIndexth bit is larger than allocated bit length of op

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
8.7.7 TEE_BigIntAssign

Since: TEE Internal Core API v1.2

```c
TEE_Result TEE_BigIntAssign(
    [out] TEE_BigInt *dest,
    [in]  TEE_BigInt  *src);
```

**Description**

The `TEE_BigIntAssign` function assigns the value of `src` to `dest`. The parameters `src` and `dest` MAY point within the same memory region but SHALL point to the start address of a `TEE_BigInt`.

**Parameters**

- `dest`: Pointer to `TEE_BigInt` to be assigned.
- `src`: Pointer to the source operand.

**Specification Number:** 10  **Function Number:** 0x1807

**Return Code**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OVERFLOW`: In case the `dest` operand cannot hold the value of `src`

**Panic Reasons**

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
8.7.8  TEE_BigIntAbs

Since: TEE Internal Core API v1.2

```c
TEE_Result TEE_BigIntAbs(
    [out] TEE_BigInt *dest,
    [in]  TEE_BigInt *src);
```

Description

The TEE_BigIntAbs function assigns the value of \(|src|\) to dest. The parameters src and dest MAY point within the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- dest: Pointer to TEE_BigInt to be assigned.
- src: Pointer to the source operand.

Specification Number: 10  Function Number: 0x1808

Return Code

- TEE_SUCCESS: In case of success.
- TEE_ERROR_OVERFLOW: In case the dest operand cannot hold the value of \(|src|\)

Panic Reasons

- If the Implementation detects any error associated with this function which is not explicitly associated with a defined return code for this function.
8.8 Basic Arithmetic Operations

This section describes basic arithmetical operations addition, subtraction, negation, multiplication, squaring, and division.

8.8.1 TEE_BigIntAdd

Since: TEE Internal API v1.0

```c
void TEE_BigIntAdd(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2 );
```

Description

The TEE_BigIntAdd function computes dest = op1 + op2. All or some of dest, op1, and op2 MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- dest: Pointer to TEE_BigInt to store the result op1 + op2
- op1: Pointer to the first operand
- op2: Pointer to the second operand

Specification Number: 10 Function Number: 0x1901

Result Size

Depending on the sign of op1 and op2, the result may be larger or smaller than op1 and op2. For the worst case, dest SHALL have memory allocation for holding max(magnitude(op1), magnitude(op2)) + 1 bits.\(^6\)

Panic Reasons

- If the implementation detects any error.

---

\(^6\) The magnitude function is defined in section 8.7.5.
8.8.2 TEE_BigIntSub

Since: TEE Internal API v1.0

```c
void TEE_BigIntSub(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2 );
```

Description

The TEE_BigIntSub function computes dest = op1 – op2. All or some of dest, op1, and op2 MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- `dest`: Pointer to TEE_BigInt to store the result op1 – op2
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand

Specification Number: 10 Function Number: 0x1906

Result Size

Depending on the sign of op1 and op2, the result may be larger or smaller than op1 and op2. For the worst case, the result SHALL have memory allocation for holding max(magnitude(op1), magnitude(op2)) + 1 bits.

Panic Reasons

- If the Implementation detects any error.
8.8.3 TEE_BigIntNeg

Since: TEE Internal API v1.0

```c
void TEE_BigIntNeg(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op );
```

Description

The TEE_BigIntNeg function negate an operand: dest = -op. dest and op MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- dest: Pointer to TEE_BigInt to store the result -op
- op: Pointer to the operand to be negated

Specification Number: 10 Function Number: 0x1904

Result Size

The result SHALL have memory allocation for magnitude(op) bits.

Panic Reasons

- If the Implementation detects any error.
8.8.4 TEE_BigIntMul

Since: TEE Internal API v1.0

```c
void TEE_BigIntMul(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2);
```

Description

The TEE_BigIntMul function computes \( \text{dest} = \text{op1} \times \text{op2} \). All or some of \( \text{dest} \), \( \text{op1} \), and \( \text{op2} \) MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- \( \text{dest} \): Pointer to TEE_BigInt to store the result \( \text{op1} \times \text{op2} \)
- \( \text{op1} \): Pointer to the first operand
- \( \text{op2} \): Pointer to the second operand

Specification Number: 10  Function Number: 0x1903

Result Size

The result SHALL have memory allocation for \( (\text{magnitude(op1)} + \text{magnitude(op2)}) \) bits.

Panic Reasons

- If the implementation detects any error.
8.8.5 TEE_BigIntSquare

Since: TEE Internal API v1.0

```c
void TEE_BigIntSquare(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op);
```

**Description**

The TEE_BigIntSquare function computes \( \text{dest} = \text{op} \times \text{op} \). \text{dest} and \text{op} MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

**Parameters**

- \text{dest}: Pointer to TEE_BigInt to store the result \( \text{op} \times \text{op} \)
- \text{op}: Pointer to the operand to be squared

**Specification Number**: 10  **Function Number**: 0x1905

**Result Size**

The result SHALL have memory allocation for \( 2^{\text{magnitude}(	ext{op})} \) bits.

**Panic Reasons**

- If the implementation detects any error.
8.8.6 TEE_BigIntDiv

Since: TEE Internal API v1.0

```c
void TEE_BigIntDiv(
    [out] TEE_BigInt *dest_q,
    [out] TEE_BigInt *dest_r,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2 );
```

Description

The TEE_BigIntDiv function computes dest_r and dest_q such that op1 = dest_q * op2 + dest_r.
It will round dest_q towards zero and dest_r will have the same sign as op1.

Example:

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>dest_q</th>
<th>dest_r</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>53 = 7*7 + 4</td>
</tr>
<tr>
<td>-53</td>
<td>7</td>
<td>-7</td>
<td>-4</td>
<td>-53 = (-7)*7 + (-4)</td>
</tr>
<tr>
<td>53</td>
<td>-7</td>
<td>-7</td>
<td>+4</td>
<td>53 = (-7)*(-7) + 4</td>
</tr>
<tr>
<td>-53</td>
<td>-7</td>
<td>7</td>
<td>-4</td>
<td>-53 = 7*(-7) + (-4)</td>
</tr>
</tbody>
</table>

To call TEE_BigIntDiv with op2 equal to zero is considered a programming error and will cause the
Trusted Application to panic.

The memory pointed to by dest_q and dest_r SHALL NOT overlap. However it is possible that
dest_q == op1, dest_q == op2, dest_r == op1, dest_r == op2, when dest_q and dest_r do not
overlap. If a NULL pointer is passed for either dest_q or dest_r, the implementation MAY take advantage
of the fact that it is only required to calculate either dest_q or dest_r.

Parameters

- dest_q: Pointer to a TEE_BigInt to store the quotient. dest_q can be NULL.
- dest_r: Pointer to a TEE_BigInt to store the remainder. dest_r can be NULL.
- op1: Pointer to the first operand, the dividend
- op2: Pointer to the second operand, the divisor

Specification Number: 10 Function Number: 0x1902

Result Sizes

The quotient, dest_q, SHALL have memory allocation sufficient to hold a TEE_BigInt with magnitude:

- 0 if |op1| <= |op2| and
- magnitude(op1) - magnitude(op2) if |op1| > |op2|.

The remainder dest_r SHALL have memory allocation sufficient to hold a TEE_BigInt with
magnitude(op2) bits.
Panic Reasons

- If op2 == 0
- If the implementation detects any other error.
8.9 Modular Arithmetic Operations

To reduce the number of tests the modular functions needs to perform on entrance and to speed up the performance, all modular functions (except TEE_BigIntMod) assume that input operands are normalized, i.e. non-negative and smaller than the modulus, and the modulus SHALL be greater than one, otherwise it is a Programmer Error and the behavior of these functions are undefined. This normalization can be done by using the reduction function in section 8.9.1.

8.9.1 TEE_BigIntMod

Since: TEE Internal API v1.0

```c
void TEE_BigIntMod(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op,
    [in] TEE_BigInt *n );
```

**Description**

The TEE_BigIntMod function computes dest = op (mod n) such that 0 <= dest < n. dest and op MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. The value n SHALL point to a unique memory region. For negative op the function follows the normal convention that -1 = (n-1) mod n.

**Parameters**

- dest: Pointer to TEE_BigInt to hold the result op (mod n). The result dest will be in the interval [0, n-1].
- op: Pointer to the operand to be reduced mod n
- n: Pointer to the modulus. Modulus SHALL be larger than 1.

**Specification Number:** 10  **Function Number:** 0x1A03

**Result Size**

The result dest SHALL have memory allocation for magnitude(n) bits.

**Panic Reasons**

- If n < 2
- If the Implementation detects any other error.

---

7 The magnitude function is defined in section 8.7.5.
### 8.9.2 TEE_BigIntAddMod

**Since:** TEE Internal API v1.0

```c
void TEE_BigIntAddMod(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2,
    [in] TEE_BigInt *n );
```

**Description**

The `TEE_BigIntAddMod` function computes `dest = (op1 + op2) (mod n)`. All or some of `dest`, `op1`, and `op2` MAY point to the same memory region but SHALL point to the start address of a `TEE_BigInt`. The value `n` SHALL point to a unique memory region.

**Parameters**

- `dest`: Pointer to `TEE_BigInt` to hold the result `(op1 + op2) (mod n)`
- `op1`: Pointer to the first operand. Operand SHALL be in the interval `[0,n-1]`.
- `op2`: Pointer to the second operand. Operand SHALL be in the interval `[0,n-1]`.
- `n`: Pointer to the modulus. Modulus SHALL be larger than 1.

**Specification Number:** 10  **Function Number:** 0x1A01

**Result Size**

The result `dest` SHALL have memory allocation for `magnitude(n)` bits.

**Panic Reasons**

- If `n < 2`
- If the implementation detects any other error.
8.9.3 TEE_BigIntSubMod

Since: TEE Internal API v1.0

```c
void TEE_BigIntSubMod(
    [out] TEE_BigInt *dest,
    [in]  TEE_BigInt *op1,
    [in]  TEE_BigInt *op2,
    [in]  TEE_BigInt *n );
```

Description

The TEE_BigIntSubMod function computes dest = (op1 - op2) (mod n). All or some of dest, op1, and op2 MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. The value n SHALL point to a unique memory region.

Parameters

- **dest**: Pointer to TEE_BigInt to hold the result (op1 - op2) (mod n)
- **op1**: Pointer to the first operand. Operand SHALL be in the interval \([0, n-1]\).
- **op2**: Pointer to the second operand. Operand SHALL be in the interval \([0, n-1]\).
- **n**: Pointer to the modulus. Modulus SHALL be larger than 1.

Specification Number: 10 Function Number: 0x1A06

Result Size

The result dest SHALL have memory allocation for magnitude(n) bits.

Panic Reasons

- **If n < 2**
- **If the implementation detects any other error.**
8.9.4 TEE_BigIntMulMod

Since: TEE Internal API v1.0

```c
void TEE_BigIntMulMod(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2,
    [in] TEE_BigInt *n );
```

Description

The TEE_BigIntMulMod function computes \( \text{dest} = (\text{op1} \times \text{op2}) \mod \text{n} \). All or some of dest, op1, and op2 MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. The value \( n \) SHALL point to a unique memory region.

Parameters

- **dest**: Pointer to TEE_BigInt to hold the result \( (\text{op1} \times \text{op2}) \mod \text{n} \)
- **op1**: Pointer to the first operand. Operand SHALL be in the interval \([0, n-1]\).
- **op2**: Pointer to the second operand. Operand SHALL be in the interval \([0, n-1]\).
- **n**: Pointer to the modulus. Modulus SHALL be larger than 1.

Specification Number: 10  Function Number: 0x1A04

Result Size

The result \( \text{dest} \) SHALL have memory allocation for magnitude(\( n \)) bits.

Panic Reasons

- If \( n < 2 \)
- If the implementation detects any other error.
### 8.9.5 TEE_BigIntSquareMod

**Since:** TEE Internal API v1.0

```c
void TEE_BigIntSquareMod(
    [out]  TEE_BigInt *dest,
    [in]   TEE_BigInt *op,
    [in]   TEE_BigInt *n );
```

**Description**

The `TEE_BigIntSquareMod` function computes \( \text{dest} = (\text{op} \cdot \text{op}) \mod \text{n} \). `dest` and `op` MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. The value `n` SHALL point to a unique memory region.

**Parameters**

- **dest**: Pointer to TEE_BigInt to hold the result \((\text{op} \cdot \text{op}) \mod \text{n}\)
- **op**: Pointer to the operand. Operand SHALL be in the interval \([0, n-1]\).
- **n**: Pointer to the modulus. Modulus SHALL be larger than 1.

**Specification Number**: 10  **Function Number**: 0x1A05

**Result Size**

The result `dest` SHALL have memory allocation for `magnitude(n)` bits.

**Panic Reasons**

- If \( n < 2 \)
- If the Implementation detects any other error.
8.9.6 TEE_BigIntInvMod

Since: TEE Internal API v1.0

```c
void TEE_BigIntInvMod(
    [out] TEE_BigInt *dest,
    [in] TEE_BigInt *op,
    [in] TEE_BigInt *n);
```

**Description**

The TEE_BigIntInvMod function computes dest such that dest * op = 1 (mod n). dest and op MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. This function assumes that gcd(op, n) is equal to 1, which can be checked by using the function in section 8.10.1. If gcd(op, n) is greater than 1, then the result is unreliable.

**Parameters**

- **dest**: Pointer to TEE_BigInt to hold the result (op^-1) (mod n)
- **op**: Pointer to the operand. Operand SHALL be in the interval [1, n-1].
- **n**: Pointer to the modulus. Modulus SHALL be larger than 1.

**Specification Number**: 10  **Function Number**: 0x1A02

**Result Size**

The result dest SHALL have memory allocation for magnitude(n) bits.

**Panic Reasons**

- If n < 2
- If op = 0
- If the Implementation detects any other error.
8.9.7  TEE_BigIntExpMod

Since: TEE Internal Core API v1.2

```c
void TEE_BigIntExpMod(
    [out] TEE_BigInt  *dest,
    [in]  TEE_BigInt  *op1,
    [in]  TEE_BigInt  *op2,
    [in]  TEE_BigInt  *n,
    [in]  TEE_BigIntFMMContext *context );
```

Description

The `TEE_BigIntExpMod` function computes \( \text{dest} = (\text{op1} \ ^ \ \text{op2}) \pmod{n} \). All or some of `dest`, `op1`, and `op2` MAY point to the same memory region but SHALL point to the start address of a `TEE_BigInt`. The value `n` SHALL point to a unique memory region. In order to utilize the FMM capabilities, a pre-computed `TEE_BigIntFMMContext` MAY be supplied. The `context` parameter MAY be `NULL`. If it is not `NULL`, the `context` SHALL be initialized using the same modulus `n` as provided as parameter.

Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result \( (\text{op1} \ ^ \ \text{op2}) \pmod{n} \)
- `op1`: Pointer to the first operand. Operand SHALL be in the interval \([0,n-1]\).
- `op2`: Pointer to the second operand. Operand SHALL be non-negative.
- `n`: Pointer to the modulus. Modulus SHALL be an odd integer larger than 2 and less than 2 to the power of `gpd.tee.arith.maxBigIntSize`.
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`, or `NULL`.

Specification Number: 10  Function Number: 0x1A07

Result Size

The result `dest` SHALL have memory allocation for `magnitude(n)` bits.

Panic Reasons

- If `n` <= 2
- If `n` even
- If the Implementation detects any other error.
8.10 Other Arithmetic Operations

8.10.1 TEE_BigIntRelativePrime

Since: TEE Internal API v1.0

```c
bool TEE_BigIntRelativePrime(
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2 );
```

Description

The TEE_BigIntRelativePrime function determines whether \( \gcd(op1, op2) = 1 \). \( op1 \) and \( op2 \) MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt.

Parameters

- \( op1 \): Pointer to the first operand
- \( op2 \): Pointer to the second operand

Specification Number: 10    Function Number: 0x1B03

Return Value

- true if \( \gcd(op1, op2) = 1 \)
- false otherwise
8.10.2 TEE_BigIntComputeExtendedGcd

Since: TEE Internal API v1.2 – See Backward Compatibility note below.

```c
void TEE_BigIntComputeExtendedGcd(
    [out] TEE_BigInt *gcd,
    [out] TEE_BigInt *u,
    [out] TEE_BigInt *v,
    [in] TEE_BigInt *op1,
    [in] TEE_BigInt *op2);
```

Description

The TEE_BigIntComputeExtendedGcd function computes the greatest common divisor of the input parameters op1 and op2. op1 and op2 SHALL NOT both be zero. Furthermore it computes coefficients u and v such that \( u \cdot \text{op1} + v \cdot \text{op2} = \text{gcd} \). op1 and op2 MAY point to the same memory region but SHALL point to the start address of a TEE_BigInt. u, v, or both can be NULL. If both are NULL, then the function only computes the gcd of op1 and op2.

Parameters

- gcd: Pointer to TEE_BigInt to hold the greatest common divisor of op1 and op2
- u: Pointer to TEE_BigInt to hold the first coefficient
- v: Pointer to TEE_BigInt to hold the second coefficient
- op1: Pointer to the first operand
- op2: Pointer to the second operand

Specification Number: 10 Function Number: 0x1B01

Result Sizes

- The gcd result SHALL be able to hold \( \text{max}(|\text{magnitude}(\text{op1})|, |\text{magnitude}(\text{op2})|) \) bits.\(^8\)
- If \( \text{op1} \neq \text{\theta} \) and \( \text{op2} \neq \text{\theta} \), then \( |u| < |\text{op2/gcd}| \) and \( |v| < |\text{op1/gcd}| \).\(^9\)
- If \( \text{op1} \neq \text{\theta} \) and \( \text{op2} = \text{\theta} \), then \( v = \text{\theta} \).
- If \( \text{op2} \neq \text{\theta} \) and \( \text{op1} = \text{\theta} \), then \( u = \text{\theta} \).

Panic Reasons

- If op1 and op2 are both zero.
- If the implementation detects any other error.

Backward Compatibility

Versions of this specification before v1.2 did not make it explicit that setting both op1 and op2 to zero is illegal. Behavior of older versions in this case is therefore undefined.

---

\(^8\) The magnitude function is defined in section 8.7.5.

\(^9\) The notation \(|x|\) means the absolute value of \(x\).
8.10.3 TEE_BigIntIsProbablePrime

Since: TEE Internal API v1.0

```c
int32_t TEE_BigIntIsProbablePrime(
    [in] TEE_BigInt *op,
    uint32_t confidenceLevel );
```

Description
The TEE_BigIntIsProbablePrime function performs a probabilistic primality test on op. The parameter confidenceLevel is used to specify the probability of a non-conclusive answer. If the function cannot guarantee that op is prime or composite, it SHALL iterate the test until the probability that op is composite is less than $2^{-\text{confidenceLevel}}$. Values smaller than 80 for confidenceLevel will not be recognized and will default to 80. The maximum honored value of confidenceLevel is implementation-specific, but SHALL be at least 80.

The algorithm for performing the primality test is implementation-specific, but its correctness and efficiency SHALL be equal to or better than the Miller-Rabin test.

Parameters
- op: Candidate number that is tested for primality
- confidenceLevel: The desired confidence level for a non-conclusive test. This parameter (usually) maps to the number of iterations and thus to the running time of the test. Values smaller than 80 will be treated as 80.

Specification Number: 10  Function Number: 0x1B02

Return Value
- 0: If op is a composite number
- 1: If op is guaranteed to be prime
- -1: If the test is non-conclusive but the probability that op is composite is less than $2^{(-\text{confidenceLevel})}$

Panic Reasons
- If the Implementation detects any error.
8.11 Fast Modular Multiplication Operations

This part of the API allows the implementer of the TEE Internal Core API to give the TA developer access to faster modular multiplication routines, possibly hardware accelerated. These functions MAY be implemented using Montgomery or Barrett or any other suitable technique for fast modular multiplication. If no such support is possible the functions in this section MAY be implemented using regular multiplication and modular reduction. The data type `TEE_BigIntFMM` is used to represent the integers during repeated multiplications such as when calculating a modular exponentiation. The internal representation of the `TEE_BigIntFMM` is implementation-specific.

8.11.1 TEE_BigIntConvertToFMM

Since: TEE Internal API v1.0

```c
void TEE_BigIntConvertToFMM(
    [out] TEE_BigIntFMM           *dest,
    [in]  TEE_BigInt              *src,
    [in]  TEE_BigInt              *n,
    [in]  TEE_BigIntFMMContext    *context );
```

Description

The `TEE_BigIntConvertToFMM` function converts `src` into a representation suitable for doing fast modular multiplication. If the operation is successful, the result will be written in implementation-specific format into the buffer `dest`, which SHALL have been allocated by the TA and initialized using `TEE_BigIntInitFMM`.

Parameters

- `dest`: Pointer to an initialized `TEE_BigIntFMM` memory area
- `src`: Pointer to the `TEE_BigInt` to convert
- `n`: Pointer to the modulus
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`

Specification Number: 10  Function Number: 0x1C03

Panic Reasons

- If the Implementation detects any error.
8.11.2 TEE_BigIntConvertFromFMM

Since: TEE Internal API v1.0

The TEE_BigIntConvertFromFMM function converts src in the fast modular multiplication representation back to a TEE_BigInt representation.

Parameters

- dest: Pointer to an initialized TEE_BigInt memory area to hold the converted result
- src: Pointer to a TEE_BigIntFMM holding the value in the fast modular multiplication representation
- n: Pointer to the modulus
- context: Pointer to a context previously initialized using TEE_BigIntInitFMMContext

Specification Number: 10   Function Number: 0x1C02

Panic Reasons

- If the implementation detects any error.
8.11.3  TEE_BigIntComputeFMM

Since: TEE Internal API v1.0

```c
void TEE_BigIntComputeFMM(
    [out] TEE_BigIntFMM *dest,
    [in]  TEE_BigIntFMM  *op1,
    [in]  TEE_BigIntFMM  *op2,
    [in]  TEE_BigInt     *n,
    [in]  TEE_BigIntFMMContext *context );
```

Description

The `TEE_BigIntComputeFMM` function calculates `dest = op1 * op2` in the fast modular multiplication representation. The pointers `dest`, `op1`, and `op2` SHALL each point to a `TEE_BigIntFMM` which has been previously initialized with the same modulus and context as used in this function call; otherwise the result is undefined. All or some of `dest`, `op1`, and `op2` MAY point to the same memory region but SHALL point to the start address of a `TEE_BigIntFMM`.

Parameters

- `dest`: Pointer to `TEE_BigIntFMM` to hold the result `op1 * op2` in the fast modular multiplication representation
- `op1`: Pointer to the first operand
- `op2`: Pointer to the second operand
- `n`: Pointer to the modulus
- `context`: Pointer to a context previously initialized using `TEE_BigIntInitFMMContext`

Specification Number: 10  Function Number: 0x1C01

Panic Reasons

- If the implementation detects any error.
9 Peripheral and Event APIs

Since: TEE Internal Core API v1.2

Note: The Peripheral and Event APIs were originally introduced in [TEE TUI Low] v1.0. They are incorporated in this document as of TEE Internal Core API v1.2. This document supersedes the text in [TEE TUI Low] v1.0 and in the event of any discrepancy, this document SHALL prevail.

The Peripheral and Event APIs, where provided by a Trusted OS, enable interaction between Trusted Applications and peripherals.

The Peripheral and Event APIs are optional, but if one is implemented the other is also required. A sentinel TEE_CORE_API_EVENT, defined in section 2.6, is set on implementations where they are supported.

9.1 Introduction

9.1.1 Peripherals

A peripheral is an ancillary component used to interact with a system, with the software interface between peripheral and system being provided by a device driver. On a typical device that includes a TEE, there may be many peripherals. The TEE is not expected to have software drivers for interacting with every peripheral attached to the device.

There are several classes of peripheral:

- Peripherals that are temporarily or permanently isolated from non-TEE entities, managed by the TEE, and fully usable by a TA through the APIs the TEE offers. These devices are described as TEE ownable.
- Peripherals that are under the total control of the REE or other entity outside the TEE and are not usable by the TEE.
- Peripherals where the TEE cannot interpret events – because it does not have the required driver – but where the TEE can control the flow of events, for example by routing flow through the TEE or by controlling the clock on a bus. These devices are described as TEE controllable.
- Peripherals for which a TEE can parse and forward events, even though the TEE does not fully control that source; e.g. a sockets interface to the REE. As the interface is hosted by the REE, it is REE controlled, but TEE parseable.

TA and TEE implementers should be aware of potential side channel attacks and provide and/or control appropriate interfaces to restrict those attacks. For example, a TEE could be configured to stop access by entities outside the TEE to specific peripherals such as accelerometers to prevent indirect interpretation of touch screen use during a TUI session.

The TEE_Peripheral_GetPeripherals function enables the TA to discover which peripherals the TEE knows about, and their characteristics, while other functions support low-level interaction with peripherals.

When a data source (or sink) is handed back to the REE, or transferred between TA instances, any state specific to previous TA activity or TA/user interaction SHALL be removed to prevent information leakage.
9.1.1.1 Access to Peripherals from a TA

Peripherals which are under the full or partial control of the TEE (i.e. peripherals which are TEE ownable, TEE parseable, or TEE controllable) MAY support exclusive access by no more than one TA at any one time.

A Trusted OS MAY provide additional access control mechanisms which are out of scope of this specification, either because they are described in separate GlobalPlatform specifications or because they are implementation-specific. An (informative) example is a Trusted OS that limits access to a peripheral to those TAs that reside in specific security domains.

The Trusted OS SHALL recover ownership of all peripherals with open handles from a TA in the following scenarios:

- The TA Panics.
- TA_DestroyEntryPoint is called for the TA owning the peripheral.

9.1.1.1.1 Multiple Access to Peripherals (informative)

Some peripherals offer multiple channels, addressing capability, or other mechanisms which have the potential to allow access to multiple endpoints. It may be convenient in some scenarios to assign different logical endpoints to different TAs, while supporting a model of exclusive access to the peripheral per TA.

One approach, shown in Figure 9-1, is to implement a separate driver interface for each of the multiple endpoints. For example, a driver for an I2C interface may support separate endpoints for each I2C address, while itself being the exclusive owner of the I2C peripheral. Such drivers SHOULD ensure that information leakage between clients of the different endpoints is prevented.

Figure 9-1: Example of Multiple Access to Bus-oriented Peripheral (Informative)
9.1.2 Event Loop

The event loop is a mechanism by which a TA can enquire for and then process messages from types of peripherals including pseudo-peripherals. The event loop can simplify TA programming in scenarios where peripheral interaction occurs asynchronously with respect to TEE operation.

Events are polymorphic, with the ability to transport device-specific payloads.

The underlying implementation of the event loop is implementation-dependent; however, the Trusted OS SHALL ensure that:

- A TA can only successfully obtain an event source for a peripheral for which it already has an open handle. This ensures that if a peripheral supports exclusive access by a single TA, sensitive information coming from a peripheral can be consumed by only that TA, preventing opportunities for information leakage.
- Events submitted to the event queue for a given peripheral are submitted in the order in which they occur. No guarantee is made of the ordering of events from different peripherals.
- An error scenario in the Event API which results in a Panic SHALL NOT cause a Panic in TAs which are blocked waiting on synchronous operations. It will either be attributed to a TEE level problem (e.g. a corrupt library) or will occur in the TEE_Event_Wait function.

9.1.3 Peripheral State

The peripheral state API provides an abstracted interface to some of the hardware features of the underlying device. It can be desirable to enable a TA to read and/or configure the hardware in a specific way, for example it may be necessary to set data transmission rates on a serial peripheral, or to discover the manufacturer of a biometric sensor.

The Peripheral API provides a mechanism by which TAs can discover information about the peripherals they use, and by which modifiable parameters can be identified and updated. It is intended to ensure that peripherals for which GlobalPlatform specifies interfaces can be used in a portable manner by TAs.

It is expected that other GlobalPlatform specifications may define state items for peripherals.

9.1.4 Overview of Peripheral and Event APIs

Figure 9-2 shows how the functions and structures of the Peripheral API are related. The notation is an adaptation of UML in which:

- “F” denotes a function call.
- “S” denotes a C struct.
- “E” denotes an enumeration: A constrained set of values of type uint32_t.
- “H” denotes a handle type, which may be an opaque pointer or some other integer type used as a unique identifier.
- Arrows are used to denote whether a value is returned from a function call or is a parameter to a function call.
- Dashed lines indicate other types of useful relationship.

Figure 9-3 shows the Event API in a similar format. Structures that are common to the Peripheral and Event APIs are shown in both diagrams to make the relation between the API sets explicit.
Figure 9-2: Peripheral API Overview

Called with a TEE_PeripheralDescriptor with only the vsId field filled.
The vsId can be obtained from TEE_Peripheral_vsId.

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved.
The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
Figure 9-3: Event API Overview
9.2 Constants

9.2.1 Handles

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

The value \texttt{TEE_INVALID_HANDLE} is used by the peripheral subsystem to denote an invalid handle.

\begin{verbatim}
#define TEE_INVALID_HANDLE ((TEE_EventQueueHandle) (0))
\end{verbatim}

9.2.2 Maximum Sizes

Since: TEE Internal Core API v1.2

Table 9-1 defines the maximum size of structure payloads. If another specification supported by a given Trusted OS requires a larger payload to support events, these MUST be implemented using pointers or handles to other structures that fit within the defined maximum structure payloads.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Constant Name & Value  \\
\hline
TEE_MAX_EVENT_PAYLOAD_SIZE & 32 bytes  \\
\hline
\end{tabular}
\caption{Maximum Sizes of Structure Payloads}
\end{table}

Backward Compatibility:

[TEE TUI LL] v1.0 offered the option of supporting larger payloads. This option is no longer supported.

9.2.3 TEE_EVENT_TYPE

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

\texttt{TEE_EVENT_TYPE} is a value indicating the source of an event.

\begin{verbatim}
#if defined(TEE_CORE_API_EVENT)
typedef uint32_t TEE_EVENT_TYPE;
#endif
\end{verbatim}

To distinguish the event types defined in various specifications:

- GlobalPlatform event types SHALL have nibble 8 (the high nibble) = 0, and SHALL include the specification number as a 3-digit BCD (Binary Coded Decimal) value in nibbles 7 through 5.

  For example, GPD_SPE_123 may define specification unique event type codes 0x01230000 to 0x0123ffff.

  All event types defined in this specification have the high word set to 0x0010.

- Event types created by external bodies SHALL have nibble 8 = 1.

- Implementation defined event types SHALL have nibble 8 = 2.

Table 9-2 lists event types defined to date.
Implementations may not support all event types; however, it is recommended that TA developers define event handlers for all of the events defined on the peripherals they support. To determine which event types are supported by a particular peripheral, the developer can consult the documentation for that peripheral.

Table 9-2: TEE_EVENT_TYPE Values

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved for future use</td>
<td>0x00000000 – 0x0000ffff</td>
</tr>
<tr>
<td>Reserved for GlobalPlatform TEE specifications numbered 001 - 009</td>
<td>0x00010000 – 0x0009ffff</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_ALL</td>
<td>0x00100000</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_CORE_CLIENT_CANCEL</td>
<td>0x00100001</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_CORE_TIMER</td>
<td>0x00100002</td>
</tr>
<tr>
<td>Reserved for future versions of this specification</td>
<td>0x00100003 – 0x0010ffff</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_ILLEGAL_VALUE</td>
<td>0x0010ffff</td>
</tr>
<tr>
<td>Reserved for GlobalPlatform TEE specifications numbered 011 - 041</td>
<td>0x00110000 – 0x0041ffff</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_BIO</td>
<td>0x00420000</td>
</tr>
<tr>
<td>Defined in [TEE TUI Bio]; if the Biometrics API is not implemented, reserved.</td>
<td>0x00420001 – 0x0042ffff</td>
</tr>
<tr>
<td>Reserved for [TEE TUI Bio]</td>
<td>0x00430000 – 0x0054ffff</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_TUI_ALL</td>
<td>0x00550000</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_TUI_BUTTON</td>
<td>0x00550001</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_TUI_KEYBOARD</td>
<td>0x00550002</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_TUI_REE</td>
<td>0x00550003</td>
</tr>
<tr>
<td>TEE_EVENT_TYPE_TUI_TOUCH</td>
<td>0x00550004</td>
</tr>
<tr>
<td>Reserved for [TEE TUI Low]</td>
<td>0x00550005 – 0x0055ffff</td>
</tr>
<tr>
<td>Reserved for GlobalPlatform TEE specifications numbered 056 – 999</td>
<td>0x00560000 – 0x0999ffff</td>
</tr>
<tr>
<td>Reserved for external bodies; number space managed by GlobalPlatform</td>
<td>0x10000000 – 0x1fffffff</td>
</tr>
<tr>
<td>Implementation Defined</td>
<td>0x20000000 – 0x2fffffff</td>
</tr>
<tr>
<td>Reserved for future use</td>
<td>0x30000000 – 0xffffffff</td>
</tr>
</tbody>
</table>

Note: TEE_EVENT_TYPE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.
### 9.2.4 TEE_PERIPHERAL_TYPE

TEE_PERIPHERAL_TYPE is a value used to identify a peripheral attached to the device.

```c
#if defined(TEE_CORE_API_EVENT)
typedef uint32_t TEE_PERIPHERAL_TYPE;
#endif
```

The TEE_Peripheral_GetPeripherals function lists all the peripherals known to the TEE.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_OS</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_CAMERA</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_MICROPHONE</td>
<td>0x00000003</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_ACCELEROMETER</td>
<td>0x00000004</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_NFC</td>
<td>0x00000005</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_BLUETOOTH</td>
<td>0x00000006</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_USB</td>
<td>0x00000007</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_FINGERPRINT</td>
<td>0x00000008</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_KEYBOARD</td>
<td>0x00000009</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_TOUCH</td>
<td>0x0000000A</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_BIO</td>
<td>0x0000000B</td>
</tr>
<tr>
<td>Reserved for GlobalPlatform specifications</td>
<td>0x0000000C - 0x3fffffff</td>
</tr>
<tr>
<td>Reserved for other Specification Development Organizations (SDOs) under Liaison Statement (LS)</td>
<td>0x40000000 - 0x7fffffff</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_ILLEGAL_VALUE</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>Implementation Defined</td>
<td>0x80000000 - 0xffffffff</td>
</tr>
</tbody>
</table>

**Note:** TEE_PERIPHERAL_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.
### 9.2.5 TEE_PERIPHERAL_FLAGS

#### Table 9-4: TEE_PERIPHERAL_FLAGS Values

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PERIPHERAL_FLAG_REE_CONTROLLED</td>
<td>0x00000000</td>
<td>The Trusted OS does not control this peripheral. All events can be processed by the REE even during TUI sessions.</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE</td>
<td>0x00000001</td>
<td>The Trusted OS can control this peripheral. Events SHALL NOT be passed to the REE during TUI sessions.</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_FLAG_EVENT_SOURCE</td>
<td>0x00000002</td>
<td>The TEE can parse the events generated by this peripheral. The peripheral can be attached to an event queue.</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_FLAG_LOCKED</td>
<td>0x00000004</td>
<td>This peripheral has been locked for access by a TA or the REE.</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_FLAG-Owned</td>
<td>0x00000008</td>
<td>This peripheral has been locked for access by this TA instance.</td>
</tr>
</tbody>
</table>

The flags TEE_PERIPHERAL_FLAG_REE_CONTROLLED and TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE are mutually exclusive.

If an event source has the TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE flag but not the TEE_PERIPHERAL_FLAG_EVENT_SOURCE flag, the TEE can control the source, but not understand it. Any events generated while the TEE has control of the source SHALL be dropped.
9.2.6 TEE_PeripheralStateId Values

TEE_PeripheralState instances are used to provide information about peripherals to a TA. The following field values, which represent legal values of type TEE_PeripheralStateId which can be used to identify specific peripheral state items, are defined in this specification. Other specifications may define additional values for TEE_PeripheralStateId.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_NAME</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_FW_INFO</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_MANUFACTURER</td>
<td>0x00000003</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_FLAGS</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Reserved for GlobalPlatform specifications</td>
<td>0x00000005 - 0x3fffffff</td>
</tr>
<tr>
<td>Reserved for other SDOs under LS</td>
<td>0x40000000 - 0x7fffffff</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_ILLEGAL_VALUE</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>Implementation Defined</td>
<td>0x80000000 - 0xffffffff</td>
</tr>
</tbody>
</table>

Note: TEE_PERIPHERAL_STATE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL be treated as an undefined value when it is provided to an API.
9.3 Peripheral State Table

Every peripheral instance has a table of associated state information. A TA can obtain this table by calling `TEE_Peripheral_GetStateTable`. Each item in the state table is of `TEE_PeripheralState` type.

The peripheral state table can be used to retrieve standardized, and peripheral specific, information about the peripheral. It also contains identifiers that can then be used for direct get/put control of specific aspects of the peripheral.

For example, a serial interface peripheral may expose interfaces to its control registers to provide direct access to readable parity error counters and writable baud rate settings.

The state table returned by `TEE_Peripheral_GetStateTable` is a read-only snapshot of peripheral state at function call time. Some of the values in the table may support modification by the caller using the `TEE_Peripheral_SetState` function – this is indicated by the value of the `ro` field.

The following sections define the state table items which could be present in the peripheral state table. Other specifications may define additional items.

9.3.1 Peripheral Name

Peripherals SHALL provide a state table entry that defines a printable name for the peripheral.

<table>
<thead>
<tr>
<th><code>TEE_PeripheralValueType Field</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tag</code></td>
<td><code>TEE_PERIPHERAL_VALUE_STRING</code></td>
</tr>
<tr>
<td><code>id</code></td>
<td><code>TEE_PERIPHERAL_STATE_NAME</code></td>
</tr>
<tr>
<td><code>ro</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>u.stringVal</code></td>
<td>Pointer to a NULL-terminated printable string which contains a printable peripheral name; SHALL be unique among the peripherals that are presented to a given TA. Note: In [TEE TUI Low] v1.0, uniqueness was recommended but not required.</td>
</tr>
</tbody>
</table>

9.3.2 Firmware Information

Peripherals MAY provide a state table entry that identifies the firmware version executing on the peripheral. This entry is only relevant to peripherals which contain a processor.

<table>
<thead>
<tr>
<th><code>TEE_PeripheralValueType Field</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tag</code></td>
<td><code>TEE_PERIPHERAL_VALUE_STRING</code></td>
</tr>
<tr>
<td><code>id</code></td>
<td><code>TEE_PERIPHERAL_STATE_FW_INFO</code></td>
</tr>
<tr>
<td><code>ro</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>u.stringVal</code></td>
<td>Pointer to a NULL-terminated printable string which contains information about the firmware running in the peripheral</td>
</tr>
</tbody>
</table>
9.3.3 Manufacturer

Peripherals MAY provide a state table entry that identifies the manufacturer of the peripheral.

Table 9-8: TEE_PERIPHERAL_STATE_MANUFACTURER Values

<table>
<thead>
<tr>
<th>TEE_PeripheralValueType Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>TEE_PERIPHERAL_VALUE_STRING</td>
</tr>
<tr>
<td>id</td>
<td>TEE_PERIPHERAL_STATE_MANUFACTURER</td>
</tr>
<tr>
<td>ro</td>
<td>true</td>
</tr>
<tr>
<td>u.stringVal</td>
<td>Pointer to a NULL-terminated printable string which contains information about the manufacturer of the peripheral</td>
</tr>
</tbody>
</table>

9.3.4 Flags

Peripherals SHALL provide a state table entry that provides information about the way in which the Trusted OS can manage the input and output from this peripheral from the calling TA using one or more of the values defined for TEE_PERIPHERAL_FLAGS – these may be combined in a bitwise manner.

Table 9-9: TEE_PERIPHERAL_STATE_FLAGS Values

<table>
<thead>
<tr>
<th>TEE_PeripheralValueType Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>TEE_PERIPHERAL_VALUE_UINT32</td>
</tr>
<tr>
<td>id</td>
<td>TEE_PERIPHERAL_STATE_FLAGS</td>
</tr>
<tr>
<td>ro</td>
<td>true</td>
</tr>
<tr>
<td>u.uint32Val</td>
<td>A combination of zero or more of the TEE_PERIPHERAL_FLAGS values defined in section 9.2.5</td>
</tr>
</tbody>
</table>

9.3.5 Exclusive Access

Peripherals SHALL provide a state table entry that identifies whether the peripheral supports exclusive access.

Table 9-10: TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS Values

<table>
<thead>
<tr>
<th>TEE_PeripheralValueType Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>TEE_PERIPHERAL_VALUE_BOOL</td>
</tr>
<tr>
<td>id</td>
<td>TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS</td>
</tr>
<tr>
<td>ro</td>
<td>true</td>
</tr>
<tr>
<td>u.boolVal</td>
<td>Set to true if this peripheral can be opened for exclusive access.</td>
</tr>
</tbody>
</table>

Note: The value of the TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS field SHALL be set to the same value on all TAs running on a given TEE which have access to that peripheral.
9.4 Operating System Pseudo-peripheral

The Operating System pseudo-peripheral provides a mechanism by which events originating in the Trusted OS or the REE can be provided to a Trusted Application.

A single instance of the Operating System pseudo-peripheral is provided by a Trusted OS supporting the Peripheral and Event APIs. It has TEE_PERIPHERAL_TYPE set to TEE_PERIPHERAL_OS.

A Trusted Application can determine the source of an Event generated by the Operating System pseudo-peripheral by looking at the event type. This information about the event source is trustworthy because it is generated within the Trusted OS. Events originating outside the Trusted OS may be less trustworthy than those originating from within the Trusted OS, and Trusted Application developers should take account of this in their designs.

The Operating System pseudo-peripheral SHALL NOT expose a TEE_PeripheralHandle, as it supports neither the polled Peripheral API nor writeable state. It SHALL expose a TEE_EventSourceHandle.

The Operating System pseudo-peripheral SHALL NOT be lockable for exclusive access and SHALL be exposed to all TA instances. Any TA in the Trusted OS can subscribe to its event queue if it wishes to do so.

9.4.1 State Table

The peripheral state table for the Operating System pseudo-peripheral SHALL contain the values listed in Table 9-11.

<table>
<thead>
<tr>
<th>TEE_PeripheralValueType.id</th>
<th>TEE_PeripheralValueType.u</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PERIPHERAL_STATE_NAME</td>
<td>&quot;TEE&quot;</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_FLAGS</td>
<td>TEE_PERIPHERAL_FLAG_EVENT_SOURCE</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS</td>
<td>false</td>
</tr>
</tbody>
</table>

9.4.2 Events

The Operating System pseudo-peripheral, when opened, SHALL return a TEE_PeripheralDescriptor which SHALL contain a valid TEE_EventSourceHandle and an invalid TEE_PeripheralHandle because it acts only as an event source.

The Operating System pseudo-peripheral can act as a source for the event types listed in section 9.6.9.
9.5 Session Pseudo-peripheral

The Session pseudo-peripheral provides a mechanism by which the events private to a specific TA session may be provided to a Trusted Application.

An instance of the Session pseudo-peripheral is provided by a Trusted OS to each open TA session. It has TEE_PERIPHERAL_TYPE set to TEE_PERIPHERAL_SESSION.

The Session pseudo-peripheral SHALL NOT expose a TEE_PeripheralHandle, as it supports neither the polled Peripheral API nor writeable state. It SHALL expose a TEE_EventSourceHandle.

The Session pseudo-peripheral SHALL be exposed only the specific session of an executing TA instance.

9.5.1 State Table

The peripheral state table for the Operating System pseudo-peripheral SHALL contain the values listed in Table 9-11.

<table>
<thead>
<tr>
<th><strong>Table 9-12</strong>: TEE_PERIPHERAL_SESSION State Table Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TEE_PeripheralValueType.id</strong></td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_NAME</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_FLAGS</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS</td>
</tr>
</tbody>
</table>

9.5.2 Events

The Session pseudo-peripheral, when opened, SHALL return a TEE_PeripheralDescriptor which SHALL contain a valid TEE_EventSourceHandle and an invalid TEE_PeripheralHandle because it acts only as an event source.

The Session pseudo-peripheral can act as a source for the following event types:

- TEE_Event_ClientCancel (see section 9.6.9.2)
- TEE_Event_Timer (see section 9.6.9.3)
9.6 Data Structures

Several data structures defined in this specification are versioned. This allows a TA written against an earlier version of this API than that implemented by a TEE to request the version of the structure it understands.

9.6.1 TEE_Peripheral

TEE_Peripheral is a structure used to provide information about a single peripheral to a TA.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#ifdef(TEE_CORE_API_EVENT)
    typedef struct
    {
        uint32_t               version;
        union                 {
            TEE_Peripheral_V1 v1;
        } u;
    } TEE_Peripheral;

    typedef struct
    {
        TEE_PERIPHERAL_TYPE    periphType;
        TEE_PeripheralId       id;
    } TEE_Peripheral_V1;
#endif
```

- version: The version of the structure – currently always 1.
- periphType: The type of the peripheral.
- id: A unique identifier for a given peripheral on a TEE.

A TEE may have more than one peripheral of the same TEE_PERIPHERAL_TYPE. The id parameter provides a TEE-unique identifier for a specific peripheral, and the implementation SHOULD provide further information about the specific peripheral instance in the TEE_PERIPHERAL_STATE_NAME field described in section 9.3.1.

The id parameter for a given peripheral SHOULD NOT change between Trusted OS version updates on a device. The id parameter is not necessarily consistent between different examples of the same device.
9.6.2 TEE_PeripheralDescriptor

TEE_PeripheralDescriptor is a structure collecting the information required to access a peripheral.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef struct
    {
        uint32_t          version;
        union {
            TEE_PeripheralDescriptor_V1 v1;
        } u;
    } TEE_PeripheralDescriptor;
#endif
```

The structure fields have the following meanings:

- The `version` field identifies the version of the TEE_PeripheralDescriptor structure. In this version of the specification it SHALL be set to 1.
- The `id` field contains a unique identifier for the peripheral with which this TEE_PeripheralDescriptor instance is associated.
- The `peripheralHandle` field contains a TEE_PeripheralHandle which, if valid, enables an owning TA to perform API calls which might alter peripheral state.
- The `eventSourceHandle` field contains a TEE_EventSourceHandle which can be used to attach events generated by the peripheral to an event queue.

9.6.3 TEE_PeripheralHandle

A TEE_PeripheralHandle is an opaque handle used to manage direct access to a peripheral.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef struct __TEE_PeripheralHandle* TEE_PeripheralHandle;
#endif
```

TA implementations SHOULD NOT assume that the same TEE_PeripheralHandle will be returned for different sessions.

The value TEE_INVALID_HANDLE is used to indicate an invalid TEE_PeripheralHandle. All other values returned by the Trusted OS denote a valid TEE_PeripheralHandle.
9.6.4 TEE_PeripheralId

A TEE_PeripheralId is a uint32_t, used as a unique identifier for a peripheral on a given TEE.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
typedef uint32_t TEE_PeripheralId;
#endif
```

TEE_PeripheralId SHALL be unique on a given TEE, and SHALL be constant for a given peripheral between TEE reboots. If a peripheral is removed and reinserted, the same value of TEE_PeripheralId SHALL be associated with it.

9.6.5 TEE_PeripheralState

TEE_PeripheralState is a structure containing the current value of an individual peripheral state value on a given TEE.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
typedef struct
{
    uint32_t                version;
    TEE_PeripheralValueType tag;
    TEE_PeripheralStateId   id;
    bool                    ro;
    union {
        uint64_t              uint64Val;
        uint32_t              uint32Val;
        uint16_t              uint16Val;
        uint8_t               uint8Val;
        bool                  boolVal;
        const char*           stringVal;
    } u;
} TEE_PeripheralState;
#endif
```

The structure fields have the following meanings:

- The version field identifies the version of the TEE_PeripheralState structure. In this version of the specification it SHALL be set to 1.
- The tag field is a TEE_PeripheralStateValueType instance indicating which field in the union, u, should be accessed to obtain the correct configuration value.
- The id field is a unique identifier for this node in the peripheral configuration tree. It can be used in the set/get API calls to select a peripheral configuration value directly.
- The ro field is true if this configuration value cannot be updated by the calling TA. A TA SHOULD NOT call TEE_PeripheralSetState with a given TEE_PeripheralStateId if the ro field of the corresponding TEE_PeripheralState is true. An implementation MAY generate an error if this is not respected.
• The union field, \( u \), contains fields representing the different data types which can be used to store
peripheral configuration information.

A Trusted OS MAY indicate different \( TEE\_PeripheralState \) information to different TAs on the system.
Therefore a TA SHOULD NOT pass \( TEE\_PeripheralState \) to another TA as the information it contains
may not be valid for the other TA.

### 9.6.6 \( TEE\_PeripheralStateId \)

A \( TEE\_PeripheralStateId \) is an identifier for a peripheral state entry on a given TEE.

**Since:** TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef uint32_t TEE_PeripheralStateId;
#endif
```

Legal values in this specification for \( TEE\_PeripheralStateId \) are listed in section 9.2.6. Further values
may be defined in other specifications.

### 9.6.7 \( TEE\_PeripheralValueType \)

\( TEE\_PeripheralValueType \) indicates which of several types has been used to store the configuration
information in a \( TEE\_PeripheralState.tag \) field.

**Since:** TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef uint32_t TEE_PeripheralValueType;
#endif
```

Table 9-13: \( TEE\_PeripheralValueType \) Values

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_PERIPHERAL_VALUE_UINT64</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_UINT32</td>
<td>0x00000001</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_UINT16</td>
<td>0x00000002</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_UINT8</td>
<td>0x00000003</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_BOOL</td>
<td>0x00000004</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_STRING</td>
<td>0x00000005</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x00000006 – 0xFFFFFE</td>
</tr>
<tr>
<td>TEE_PERIPHERAL_VALUE_ILLEGAL_VALUE</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>Implementation defined</td>
<td>0x80000000 – 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

**Note:** \( TEE\_PERIPHERAL\_VALUE\_ILLEGAL\_VALUE \) is reserved for testing and validation. It SHALL be treated
as an undefined value when it is provided to an API.
9.6.8 TEE_Event

TEE_Event is a container for events in the event loop.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
typedef struct {
    uint32_t     version;
    union {
        TEE_Event_V1 v1;
    } u;
} TEE_Event;
#endif
```

The TEE_Event structure holds an individual event; the payload holds an array of bytes whose contents are interpreted according to the type of the event:

- version: The version of the structure – currently always 1.
- eventType: A value identifying the type of event.
- timestamp: The time the event occurred given as milliseconds since the TEE was started. The value of timestamp is guaranteed to increase monotonically so that the ordering of events in time is guaranteed. A Trusted OS SHOULD use the same underlying source of time information as used for TEE_GetSystemTime, described in section 7.2.1.
- eventSourceHandle: The handle of the specific event source that created this event.
- payload: A block of TEE_MAXEVENT_PAYLOAD_SIZE bytes. The content of payload, while defined for TEE_PERIPHERAL_OS, is not generally defined in this specification. Payloads specific to particular APIs may be defined in other specifications. Any unused trailing bytes SHALL be zero.

In general, if an event cannot be sufficiently described within the constraints of the payload field of TEE_MAX_EVENT_PAYLOAD_SIZE, the contents of the field may be data structure containing handles or pointers to further structures that together fully describe the event.
9.6.9 Generic Payloads

This section describes a generic payload field of the TEE_Event structure.

9.6.9.1 TEE_Event_AccessChange

This event is generated if the accessibility of a peripheral to this TA changes.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
typedef struct {
    uint32_t         version;
    TEE_PeripheralId id;
    uint32_t         flags;
} TEE_Event_AccessChange;
#endif
```

- **version**: The version of the structure – currently always 1.
- **id**: The TEE_PeripheralId for the peripheral for which the access change event was generated. This uniquely identifies the peripheral for which the access status has changed.
- **flags**: The new state of TEE_PERIPHERAL_STATE_FLAGS. For details of the legal values for this field, see the description of the u.uint32Val field in section 9.3.4.

This event SHALL be sent to all TAs which have registered to the TEE_PERIPHERAL_OS event queue when an access permission change occurs – including the TA which initiated the change.

A consequence of TEE_Event_AccessChange is that some of the peripheral state table information may change. As such, each TA instance SHOULD call TEE_Peripheral_GetStateTable to obtain fresh information when it receives this event.

9.6.9.2 TEE_Event_ClientCancel

When a TEE_Event_V1 with eventType of TEE_EVENT_TYPE_CORE_CLIENT_CANCEL is received, the TEE_Event_V1 payload has type TEE_Event_ClientCancel.

Since: TEE Internal Core API v1.2

```c
#if defined(TEE_CORE_API_EVENT)
typedef struct {
    uint32_t         version;
} TEE_Event_ClientCancel;
#endif
```

- **version**: The version of the structure – currently always 1.

This event SHALL be sent only to the TA session for which cancellation was requested on the appropriate TEE_PERIPHERAL_SESSION event queue when cancellation was requested.

9.6.9.3 TEE_Event_Timer

When a TEE_Event_V1 with eventType of TEE_EVENT_TYPE_CORE_CLIENT_TIMER is received in a given TA session context, the TEE_Event_V1 payload has type TEE_Event_Timer.
Since: TEE Internal Core API v1.2

```c
#if defined(TEE_CORE_API_EVENT)
    typedef struct {
        uint8_t          payload[TEE_MAX_EVENT_PAYLOAD_SIZE];
    } TEE_Event_Timer;
#endif
```

- payload: A byte array containing a payload whose contents are defined by the TA when the timer is created.

This event SHALL be sent only to the TA session for which timer event was requested on the appropriate TEE_PERIPHERAL_SESSION event queue when cancellation was requested.

### 9.6.10 TEE_EventQueueHandle

A TEE_EventQueueHandle is an opaque handle for an event queue.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef struct __TEE_EventQueueHandle* TEE_EventQueueHandle;
#endif
```

A Trusted OS SHOULD ensure that the value of TEE_EventQueueHandle returned to a TA is not predictable and SHALL ensure that it does contain all or part of a machine address.

The value TEE_INVALID_HANDLE is used to indicate an invalid TEE_EventQueueHandle. All other values returned by the Trusted OS denote a valid TEE_EventQueueHandle.

### 9.6.11 TEE_EventSourceHandle

A TEE_EventSourceHandle is an opaque handle for a specific source of events, for example a biometric sensor.

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    typedef struct __TEE_EventSourceHandle* TEE_EventSourceHandle;
#endif
```

The value TEE_INVALID_HANDLE is used to indicate an invalid TEE_EventSourceHandle. All other values returned by the Trusted OS denote a valid TEE_EventSourceHandle.
### 9.7 Peripheral API Functions

#### 9.7.1 TEE_Peripheral_Close

**Since:** TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    TEE_Result TEE_Peripheral_Close(
        TEE_PeripheralDescriptor *peripheralDescriptor
    );
#endif
```

**Description**

The `TEE_Peripheral_Close` function is used by a TA to release a single peripheral. On successful return, the `peripheralHandle` and `eventSourceHandle` values pointed to by `peripheral` **SHALL** be `TEE_INVALID_HANDLE`.

**Specification Number:** 10  **Function Number:** 0x2001

**Parameters**
- `peripheralDescriptor`: A pointer to a `TEE_PeripheralDescriptor` structure.

**Return Value**
- `TEE_SUCCESS`: In case of success. At least one of `peripheralHandle` and `eventSourceHandle` points to a valid handle.
- `TEE_ERROR_BAD_STATE`: The calling TA does not have a valid open handle to the peripheral.
- `TEE_ERROR_BAD_PARAMETERS`: `peripheral` is NULL.

**Panic Reasons**

`TEE_Peripheral_Close` **SHALL** NOT panic.
9.7.2  TEE_Peripheral_CloseMultiple

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    TEE_Result TEE_Peripheral_CloseMultiple(
        const    uint32_t                  numPeripherals,
        [inout]  TEE_PeripheralDescriptor *peripheralDescriptors
    );
#endif
```

Description

TEE_Peripheral_CloseMultiple is a convenience function which closes all the peripherals identified in the buffer pointed to by peripherals. In contrast to TEE_Peripheral_OpenMultiple, there is no guarantee of atomicity; the function simply attempts to close all the requested peripherals.

Specification Number: 10  Function Number: 0x2002

Parameters

- numPeripherals: The number of entries in the TEE_PeripheralDescriptor buffer pointed to by peripherals.
- peripheralDescriptors: A pointer to a buffer of numPeripherals instances of TEE_PeripheralDescriptor. The interpretation and treatment of each individual entry in the buffer of descriptors is as described for TEE_Peripheral_Close in section 9.7.1.

Return Value

- TEE_SUCCESS: In case of success, which is defined as all the requested TEE_PeripheralDescriptor instances having been successfully closed.
- TEE_ERROR_BAD_STATE: The calling TA does not have a valid open handle to at least one of the peripherals.
- TEE_ERROR_BAD_PARAMETERS: peripherals is NULL and/or numPeripherals is 0.

Panic Reasons

TEE_Peripheral_CloseMultiple SHALL NOT panic.
9.7.3 TEE_Peripheral_GetPeripherals

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_GetPeripherals(
    [inout] uint32_t* version,
    [outbuf] TEE_Peripheral* peripherals, size_t* size
);
#endif
```

Description

The `TEE_Peripheral_GetPeripherals` function returns information about the peripherals known to the TEE. This function MAY list all peripherals attached to the implementation and SHALL list all peripherals visible to the calling TA. The TEE may not be able to control all the peripherals. Of those that the TEE can control, it may not be able to parse the events generated, so not all can be used as event sources.

Specification Number: 10 Function Number: 0x2003

Parameters

- **version**:
  - On entry, the highest version of the `TEE_Peripheral` structure understood by the calling program.
  - On return, the actual version returned, which may be lower than the value requested.
- **peripherals**: A pointer to an array of `TEE_Peripheral` structures. This will be populated with information about the available sources on return. Each structure in the array returns information about one peripheral.
- **size**:
  - On entry, the size of `peripherals` in bytes.
  - On return, the actual size of the buffer containing the `TEE_Peripheral` structures in bytes. The combination of `peripherals` and `size` complies with the `[outbuf]` behavior specified in section 3.4.4.

Return Value

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_OLD_VERSION**: If the version of the `TEE_Peripheral` structure requested is not supported.
- **TEE_ERROR_OUT_OF_MEMORY**: If the system ran out of resources.
- **TEE_ERROR_SHORT_BUFFER**: If the output buffer is not large enough to hold all the sources.
- **TEE_ERROR_EXTERNAL_CANCEL**: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

Panic Reasons

- If `version` is NULL.
- If `peripherals` is NULL and/or `size` is not zero.
See section 3.4.4 for reasons for [outbuf] generated panic.

If the implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

9.7.4 TEE_Peripheral_GetState

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_GetState(
    const TEE_PeripheralId      id,
    const TEE_PeripheralStateId stateId,
    [out] TEE_PeripheralValueType*    periphType,
    [out] void*                       value
);
#endif
```

Description

The TEE_Peripheral_GetState function enables a TA which knows the state ID of a peripheral state item to fetch the value of this directly. A TA does not need to have an open handle to a peripheral to obtain information about its state – this allows a TA to discover information about peripherals available to it before opening a handle.

Specification Number: 10   Function Number:  0x2004

Parameters

- id: The unique peripheral identifier for the peripheral in which we are interested.
- stateID: The identifier for the state item for which the value is requested.
- periphType: On return, contains a value of TEE_PeripheralValueType which determines how the data pointed to by value should be interpreted.
- value: On return, points to the value of the requested state item.

Note: The caller SHALL ensure that the buffer pointed to by value is large enough to accommodate whichever is the larger of `uint64_t` and `char*` on a given TEE platform.

Return Value

- TEE_SUCCESS: State information has been fetched.
- TEE_ERROR_BAD_PARAMETERS: The value of one or both of id or stateId are not valid for this TA; periphType or value is NULL.

Panic Reasons

TEE_Peripheral_GetState SHALL NOT panic.
9.7.5 TEE_Peripheral_GetStateTable

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    TEE_Result TEE_Peripheral_GetStateTable(
        [in] TEE_PeripheralId id,
        [outbuf] TEE_PeripheralState* stateTable, size_t* bufSize
    );
#endif
```

Description

The TEE_Peripheral_GetStateTable function fetches a buffer containing zero or more instances of TEE_PeripheralState. These provide a snapshot of the state of a peripheral.

Specification Number: 10  Function Number: 0x2005

Parameters

- `id`: The TEE_PeripheralId for the peripheral from which the TA wishes to read data
- `stateTable`: A buffer of at least `bufSize` bytes that on successful return is overwritten with an array of TEE_PeripheralState structures.
- `bufSize`:
  - On entry, the size of `stateTable` in bytes.
  - On return, the actual number of bytes in the array. The combination of `stateTable` and `bufSize` complies with the [outbuf] behavior specified in section 3.4.4.

Return Value

- `TEE_SUCCESS`: Data has been written to the peripheral.
- `TEE_ERROR_BAD_PARAMETERS`: The value of `id` or `stateTable` is NULL and/or `bufSize` is 0.

Panic Reasons

- See section 3.4.4 for reasons for [outbuf] generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.7.6 TEE_Peripheral_Open

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_Open(
    [inout] TEE_PeripheralDescriptor *peripheralDescriptor
);
#endif
```

Description

The `TEE_Peripheral_Open` function is used by a TA to obtain descriptor(s) enabling access to a single peripheral. If the TA needs to open more than one peripheral for related activities, it MAY use `TEE_Peripheral_OpenMultiple`.

If this function executes successfully and if `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` indicates that exclusive access is supported, then the Trusted OS guarantees that neither the REE, nor any other TA, has access to the peripheral. If `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` indicates that exclusive access is not supported, the calling TA SHOULD assume that it does not have exclusive access to the peripheral.

The Trusted OS returns handles which can be used by the TA to manage interactions with the peripheral. If `TEE_Peripheral_Open` succeeds, at least one of `peripheralHandle` and `eventSourceHandle` is set to a valid handle value.

It is an error to call `TEE_Peripheral_Open` for a peripheral which is already owned by the calling TA instance.

Specification Number: 10  Function Number: 0x2006

Parameters

- `peripheralDescriptor`: A pointer to a `TEE_PeripheralDescriptor` structure. The fields of the structure pointed to are used as follows:
  - `id`: This is the unique identifier for a specific peripheral, as returned by `TEE_Peripheral_GetPeripherals`. This field SHALL be set on entry, and SHALL be unchanged on return.
  - `peripheralHandle`: On entry, the value is ignored and will be overwritten. On return, the value is set as follows:
    - `TEE_INVALID_HANDLE`: This peripheral does not support the Peripheral API.
    - Other value: An opaque handle which can be used with the Peripheral API functions.
  - `eventSourceHandle`: On entry, the value is ignored and will be overwritten. On return, the value is set as follows:
    - `TEE_INVALID_HANDLE`: This peripheral does not support the Event API.
    - Other value: An opaque handle which can be used with the Event API functions.

Return Value

- `TEE_SUCCESS`: In case of success. At least one of `peripheralHandle` and `eventSourceHandle` points to a valid handle.
- `TEE_ERROR_BAD_PARAMETERS`: peripheral is NULL.
• TEE_ERROR_ACCESS_DENIED: If the system was unable to acquire exclusive access to a peripheral for which TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS indicates exclusive access is possible.

**Panic Reasons**

- If peripheral->id is not known to the system.
- If peripheral->id is already owned by the calling TA instance.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 9.7.7 TEE_Peripheral_OpenMultiple

**Since:** TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_OpenMultiple(  
    const    uint32_t                  numPeripherals,  
    [inout] TEE_PeripheralDescriptor *peripheralDescriptors
);
#endif
```

**Description**

The TEE_Peripheral_OpenMultiple function is used by a TA to atomically obtain access to multiple peripherals.

TEE_Peripheral_OpenMultiple behaves as though a call to TEE_Peripheral_Open is made to each TEE_PeripheralDescriptor in peripherals in turn, but ensures that all or none of the peripherals have open descriptors on return. This function should be used where a TA needs simultaneous control of multiple peripherals to operate correctly.

If this function executes successfully, the Trusted OS guarantees that neither the REE, nor any other TA, has access to any requested peripheral for which exclusive access is supported (as indicated by TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS). If an error is returned, the Trusted OS guarantees that no handle is open for any of the requested peripherals.

The Trusted OS returns handles which can be used by the TA to manage interactions with the peripheral. If TEE_Peripheral_OpenMultiple succeeds, at least one of peripheralHandle and eventSourceHandle fields in each descriptor is set to a valid handle value. If an error is returned, all the peripheralHandle and eventSourceHandle fields in each descriptor SHALL contain TEE_INVALID_HANDLE.

**Specification Number:** 10    **Function Number:** 0x2007

**Parameters**

- numPeripherals: The number of entries in the TEE_PeripheralDescriptor buffer pointed to by peripherals.
- peripheralDescriptors: A pointer to a buffer of numPeripherals instances of TEE_PeripheralDescriptor. The interpretation and treatment of each individual entry in the buffer of descriptors is as described for TEE_Peripheral_Open in section 9.7.6.
Return Value

- **TEE_SUCCESS**: In case of success. At least one of peripheralHandle and eventSourceHandle points to a valid handle in every entry in peripherals.

- **TEE_ERROR_BAD_PARAMETERS**: peripherals is NULL and/or numPeripherals is 0.

- **TEE_ERROR_ACCESS_DENIED**: If the system was unable to acquire exclusive access to all the requested peripherals.

Panic Reasons

- If peripheralDescriptors[x].id for any instance, x, of TEE_PeripheralDescriptor is not known to the system.

- If peripheralDescriptors[x].id for any instance, x, of TEE_PeripheralDescriptor is already owned by the calling TA.

- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.7.8 TEE_Peripheral_Read

Since: TEE Internal Core API v1.2

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_Read(
    [in] TEE_PeripheralHandle peripheralHandle,
    [outbuf] void *buf, size_t *bufSize
);
#endif
```

Description

The TEE_Peripheral_Read function provides a low-level API to read data from the peripheral denoted by `peripheralHandle`. The `peripheralHandle` field of the peripheral descriptor must be a valid handle for this function to succeed.

The calling TA allocates a buffer of `bufSize` bytes before calling. On return, this will contain as much data as is available from the peripheral, up to the limit of `bufSize`. The `bufSize` parameter will be updated with the actual number of bytes placed into `buf`.

`TEE_Peripheral_Read` is designed to allow a TA to implement polled communication with peripherals. The function SHALL NOT wait on any hardware signal and SHALL retrieve only the data which is available at the time of calling.

While some peripherals may support both the event queue and the polling interface, it is recommended that TA implementers do not attempt to use both polling and the event queue to read data from the same peripheral. Peripheral behavior if both APIs are used on the same peripheral is undefined.

Note: depending on the use-case, polled interfaces can result in undesirable power consumption profiles.

Specification Number: 10  Function Number: 0x2008

Parameters

- `peripheralHandle`: A valid TEE_PeripheralHandle for the peripheral from which the TA wishes to read data.
- `buf`: A buffer of at least `bufSize` bytes which, on successful return, will be overwritten with data read back from the peripheral.
- `bufSize`:
  - On entry, the size of `buf` in bytes.
  - On return, the actual number of bytes read from the peripheral. The combination of `buf` and `bufSize` complies with the [outbuf] behavior specified in section 3.4.4.

Return Value

- `TEE_SUCCESS`: Data has been read from the peripheral. The value of `bufSize` indicates the number of bytes read.
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to hold all the sources.
- `TEE_ERROR_EXCESS_DATA`: Data was read successfully, but the peripheral has more data available to read. In this case, `bufSize` is the same value as was indicated when the function was called. It is recommended that the TA read back the remaining data from the peripheral before continuing.
• **TEE_ERROR_BAD_PARAMETERS**: The value of `peripheralHandle` is `TEE_INVALID_HANDLE`; or `buf` is `NULL` and `bufSize` is not zero.

### Panic Reasons

- If the calling TA does not provide a valid `peripheralHandle`.
- See section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### Backward Compatibility

[TEE TUI Low] v1.0 did not include the TEE_ERROR_SHORT_BUFFER return value.

### 9.7.9 TEE_Peripheral_SetState

**Since:** TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_SetState(
    const TEE_PeripheralHandle handle,
    const TEE_PeripheralStateId stateId,
    const TEE_PeripheralValueType periphType,
    const void* value

); #endif
```

### Description

The **TEE_Peripheral_SetState** function enables a TA to set the value of a writeable peripheral state item. Items are only writeable if the `ro` field of the **TEE_PeripheralState** for the state item is `false`. The value of the `ro` field can change between a call to **TEE_Peripheral_GetState** and a subsequent call to **TEE_Peripheral_SetState**.

TAs SHOULD call **TEE_Peripheral_GetStateTable** for the peripheral id in question to determine which state items are writeable by the TA.

Note that any previous snapshot of peripheral state will not be updated after a call to **TEE_Peripheral_SetState**.

### Specification Number: 10  Function Number: 0x2009

### Parameters

- `handle`: A valid open handle for the peripheral whose state is to be updated.
- `stateId`: The identifier for the state item for which the value is requested.
- `periphType`: A value of **TEE_PeripheralValueType** which determines how the data pointed to by `value` should be interpreted.
- `value`: The address of the value to be written to the state item.

### Return Value

- **TEE_SUCCESS**: State information has been updated.
• TEE_ERROR_BAD_PARAMETERS: The value of one or both of handle or stateId are not valid for this TA; or periphType is not a value defined in TEE_PeripheralValueType; or value is NULL; or the value which is being written is read-only.

Panic Reasons

TEE_Peripheral_SetState SHALL NOT panic.
9.7.10 TEE_Peripheral_Write

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Peripheral_Write(
    [in] TEE_PeripheralHandle peripheralHandle,
    [inbuf] void *buf, size_t bufSize
);
#endif
```

**Description**

The TEE_Peripheral_Write function provides a low-level API to write data to the peripheral denoted by peripheralHandle. The peripheralHandle field of the peripheral descriptor must be a valid handle for this function to succeed.

The calling TA allocates a buffer of bufSize bytes before calling and fills it with the data to be written.

**Specification Number:** 10  **Function Number:** 0x200A

**Parameters**

- **peripheralHandle**: A valid TEE_PeripheralHandle for the peripheral from which the TA wishes to read data.
- **buf**: A buffer of at least bufSize bytes containing data which has, on successful return, been written to the peripheral.
- **bufSize**: The size of buf in bytes.

**Return Value**

- **TEE_SUCCESS**: Data has been written to the peripheral.
- **TEE_ERROR_BAD_PARAMETERS**: buf is NULL and/or bufSize is 0.

**Panic Reasons**

- If peripheralHandle is not a valid open handle to a peripheral.
- If the implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8 Event API Functions

9.8.1 TEE_Event_AddSources

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#ifdef(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_AddSources(
    uint32_t                 numSources,
    [in] TEE_EventSourceHandle   *sources,
    [in] TEE_EventQueueHandle     *handle
);
#endif
```

Description

The TEE_Event_AddSources function atomically adds new event sources to an existing queue acquired by a call to TEE_Event_OpenQueue. If the function succeeds, events from this source are exclusively available to this queue.

If the function fails, the queue is still valid. The queue SHALL contain events from the original sources and MAY contain some of the requested sources. In case of error, the caller should use TEE_Event_ListSources to determine the current state of the queue.

It is not an error to add an event source to a queue to which it is already attached.

Specification Number: 10  Function Number: 0x2101

Parameters

- numSources: Defines how many sources are provided.
- sources: An array of TEE_EventSourceHandle that the TA wants to add to the queue.
- handle: The handle for the queue.

Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_BAD_STATE: If the handle does not represent a currently open queue.
- TEE_ERROR_BUSY: If any requested resource cannot be reserved.
- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- TEE_ERROR_OUT_OF_MEMORY: If the system ran out of resources.

Panic Reasons

- If handle is invalid.
- If the sources array does not contain numSources elements.
- If any pointer in sources is NULL.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.2 TEE_Event_CancelSources

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_CancelSources(
    uint32_t                numSources,
    [in] TEE_EventSourceHandle *sources,
    [in] TEE_EventQueueHandle   *handle
);
#endif
```

Description

The **TEE_Event_CancelSources** function drops all existing events from a set of sources from a queue previously acquired by a call to **TEE_Event_OpenQueue**.

New events from these sources will continue to be added to the queue, unless the TA has released the sources using **TEE_Event_DropSources** or **TEE_Event_CloseQueue**.

It is not an error to cancel an event source that is not currently attached to the queue.

**Specification Number:** 10  **Function Number:** 0x2102

Parameters

- **numSources**: Defines how many sources are provided.
- **sources**: An array of **TEE_EventSourceHandle**. Events from these sources are cleared from the queue.
- **handle**: The handle for the queue.

Return Value

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_OUT_OF_MEMORY**: If the system ran out of resources.
- **TEE_ERROR_BAD_STATE**: If the handle does not represent a currently open queue.
- **TEE_ERROR_EXTERNAL_CANCEL**: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

Panic Reasons

- If **handle** is invalid.
- If the **sources** array does not contain **numSources** elements.
- If any pointer in **sources** is **NULL**.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.3 TEE_Event_CloseQueue

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_CloseQueue( [in] TEE_EventQueueHandle *handle );
#endif
```

**Description**

The `TEE_Event_CloseQueue` function releases TUI resources previously acquired by a call to `TEE_Event_OpenQueue`. All outstanding events on the queue will be invalidated.

**Specification Number:** 10  **Function Number:** 0x2103

**Parameters**

- `handle`: The handle to the `TEE_EventQueueHandle` to close.

**Return Value**

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the handle does not represent a currently open queue.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

**Panic Reasons**

- If `handle` is invalid.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.4 TEE_Event_DropSources

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
   TEE_Result TEE_Event_DropSources(
       uint32_t                  numSources,
       [in] TEE_EventSourceHandle   *sources,
       [in] TEE_EventQueueHandle     *handle
   );
#endif
```

Description

The TEE_Event_DropSources function removes one or more event sources from an existing queue previously acquired by a call to TEE_Event_OpenQueue. No more events from these sources are added to the queue. Events from these sources will be available to the REE, until they are reserved by this or another TA using TEE_Event_AddSources or TEE_Event_OpenQueue.

Events from other event sources will continue to be added to the queue. It is permissible to have a queue with no current event sources attached to it. It is not an error to drop an event source that is not currently attached to the queue.

Specification Number: 10 Function Number: 0x2104

Parameters

- `numSources`: Defines how many sources are provided.
- `sources`: An array of TEE_EventSourceHandle. Events from these sources are cleared from the queue.
- `handle`: The handle for the queue.

Return Value

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_BAD_STATE**: If the handle does not represent a currently open queue.
- **TEE_ERROR_ITEM_NOT_FOUND**: If one or more sources was not attached to the queue. All other sources are dropped.
- **TEE_ERROR_EXTERNAL_CANCEL**: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

Panic Reasons

- If `handle` is invalid.
- If the `sources` array does not contain `numSources` elements.
- If any pointer in `sources` is NULL.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.5 TEE_Event_ListSources

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_ListSources(
    [in]   TEE_EventQueueHandle    *handle,
    [outbuf] TEE_EventSourceHandle   *sources, size_t* bufSize
);
#endif
```

Description

The `TEE_Event_ListSources` function returns information about sources currently attached to a queue.

**Specification Number:** 10  **Function Number:** 0x2105

Parameters

- `handle`: The handle for the queue.
- `sources`: A buffer of at least `bufSize` bytes that on successful return is overwritten with an array of `TEE_EventSourceHandle` structures.
- `bufSize`:
  - On entry, the size of `sources` in bytes.
  - On return, the actual number of bytes in the array. The combination of `sources` and `bufSize` complies with the `[outbuf]` behavior specified in section 3.4.4.

Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to hold all the sources.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

Panic Reasons

- If `handle` is invalid.
- If `bufSize` is NULL.
- If `sources` is NULL.
- See section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.6 TEE_Event_OpenQueue

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
    TEE_Result TEE_Event_OpenQueue(
        [inout] uint32_t               *version,
        uint32_t                numSources,
        uint32_t                timeout,
        [in] TEE_EventSourceHandle  *sources,
        [out] TEE_EventQueueHandle   *handle
    );
#endif
```

Description

The `TEE_Event_OpenQueue` function claims an exclusive access to TUI resources for the current TA instance.

This function allows for multiple event sources to be reserved.

It is possible for multiple TAs to open queues at the same time provided they do not try to reserve any of the same resources.

An individual TA SHALL NOT open multiple queues; instead, the TA SHOULD use `TEE_Event_AddSources` and `TEE_Event_DropSources` to add and remove event sources from the queue.

The `TEE_EventQueue` will be closed automatically if no calls to `TEE_Event_Wait` are made for timeout milliseconds. This has the same guarantees as the `TEE_Wait` function.

Specification Number: 10 Function Number: 0x2106

Parameters

- version:
  - On entry, the highest version of the `TEE_Event` structure understood by the calling program.
  - On return, the actual version of the `TEE_Event` structure that will be added to the queue, which may be lower than the value requested.

- numSources: Defines how many sources are provided.

- timeout: The timeout for this function in milliseconds.

- sources: An array of `TEE_EventSourceHandle`, as returned from `TEE_Event_ListSources`.

- handle: The handle for this session. This value SHOULD Be Zero on entry and is set if the session is successfully established and `numSources` is not zero.

Return Value

- TEE_SUCCESS: In case of success.

- TEE_ERROR_BUSY: If any requested resource cannot be reserved.

- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

- TEE_ERROR_OLD_VERSION: If the version of the `TEE_Event` structure requested is not supported.

- TEE_ERROR_OUT_OF_MEMORY: If the system ran out of resources.
Panic Reasons

- If **version** is invalid.
- If **handle** is **NULL**.
- If the **sources** array does not contain **numSources** elements.
- If any pointer in **sources** is **NULL**.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
9.8.7 TEE_Event_TimerCreate

Since: TEE Internal Core API v1.2

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_TimerCreate(
    [in] TEE_EventQueueHandle *handle,
    [in] uint64_t              period,
    [in] uint8_t               payload[TEE_MAX_EVENT_PAYLOAD_SIZE]
); #endif
```

Description

The `TEE_Event_TimerCreate` function creates a one-shot timer which, on expiry, will cause `TEE_Event_Timer` to be placed onto the event queue designated by `handle`. Although the accuracy of `period` cannot be guaranteed, events are timestamped if the TA requires an accurate measure of the time between events.

Specification Number: 10  Function Number: 0x2108

Parameters

- `handle`: The handle for the queue.
- `period`: The minimum timer period in milliseconds. The accuracy of the timer period is subject to the constraints of `TEE_Wait` (See section 7.2.2).
- `payload`: A payload chosen by the TA which is returned in the `TEE_Event_Timer` payload when the timer expires.

Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BUSY`: If any requested resource cannot be reserved.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

Panic Reasons

- If `handle` is invalid.
9.8.8 TEE_Event_Wait

Since: TEE Internal Core API v1.2 (originally defined identically in [TEE TUI Low] v1.0)

```c
#if defined(TEE_CORE_API_EVENT)
TEE_Result TEE_Event_Wait(
    [in]  TEE_EventQueue *handle,
    [inout] TEE_Event timeout,
    [inout] TEE_Event *events,
    [inout] uint32_t *numEvents,
    [out]   uint32_t *dropped
);
#endif
```

Description

The `TEE_Event_Wait` function fetches events that have been returned from a peripheral reserved by `TEE_Event_OpenQueue`. Events are not guaranteed to be delivered as the event queue has a finite size. If the event queue is full, the oldest event(s) SHALL be dropped first, and the dropped event count SHALL be updated with the number of dropped events. Events MAY also be dropped out of order for reasons outside the scope of this specification, but the dropped event count SHOULD reflect this.

The API allows one or more events to be obtained at a time to minimize any context switching overhead, and to allow a TA to process bursts of events en masse.

Obtaining events has a timeout, allowing a TA with more responsibilities than just user interaction to attend to these periodically without needing to use multi-threading.

The `TEE_Event_Wait` function opens the input event stream. If the stream is not available for exclusive access within the specified timeout, an error is returned. A zero timeout means this function returns immediately. This has the same guarantees as the `TEE_Wait` function.

Events are returned in order of decreasing age: `events[0]` is the oldest available event, `events[1]` the next oldest, etc.

On entry, `*numEvents` contains the number of events pointed to by `events`.

`*numEvents` can be 0 on entry, which allows the TA to query whether input is available. If timeout == 0, the function should return `TEE_SUCCESS` if there are pending events and `TEE_ERROR_TIMEOUT` if there is no pending event.

On return, `*numEvents` contains the actual number of events written to `events`.

If the function returns with any status other than `TEE_SUCCESS`, `*numEvents` = 0.

If there are no events available in the given timeout, `*numEvents` is set to zero and this function returns an error.

If any events occur, the function returns as soon as possible, and does not wait until `*numEvents` events have occurred.

If `dropped` is non-NULL, the current count of dropped events is written to this location.

This function is cancellable. If the cancelled flag of the current instance is set and the TA has unmasked the effects of cancellation, then this function returns earlier than the requested timeout.

- If the operation was cancelled by the client, `TEE_ERROR_CANCEL` is returned. See section 4.10 for more details about cancellations.
- If the cancellation was not sourced by the client, the TEE SHOULD cancel the function and `TEE_ERROR_EXTERNAL_CANCEL` is returned.
Specification Number: 10  Function Number: 0x2107

Parameters

- **handle**: The handle for the queue
- **timeout**: The timeout in milliseconds
- **events**: A pointer to an array of TEE_Event structures
- **numEvents**:
  - On entry, the maximum number of events to return
  - On return, the actual number of events returned
- **dropped**: A pointer to a count of dropped events

Return Value

- **TEE_SUCCESS**: In case of success.
- **TEE_ERROR_BAD_STATE**: If handle does not represent a currently open queue.
- **TEE_ERROR_TIMEOUT**: If there is no event to return within the timeout.
- **TEE_ERROR_EXTERNAL_CANCEL**: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- **TEE_ERROR_CANCEL**: If the operation was cancelled by anything other than an event in the REE.

Panic Reasons

- If handle is invalid.
- If events is NULL.
- If numEvents is NULL.
- If dropped is NULL.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.
Annex A  Panicked Function Identification

If this specification is used in conjunction with [TEE TA Debug], then the specification number is 10 and the values listed in Table A-1 SHALL be associated with the function declared.

Table A-1: Function Identification Values

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Function Number in hexadecimal</th>
<th>Function Number in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA Interface</td>
<td>TA_CloseSessionEntryPoint</td>
<td>0x101</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>TA_CreateEntryPoint</td>
<td>0x102</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>TA_DestroyEntryPoint</td>
<td>0x103</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>TA_InvokeCommandEntryPoint</td>
<td>0x104</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>TA_OpenSessionEntryPoint</td>
<td>0x105</td>
<td>261</td>
</tr>
<tr>
<td>Property Access</td>
<td>TEE_AllocatePropertyEnumerator</td>
<td>0x201</td>
<td>513</td>
</tr>
<tr>
<td></td>
<td>TEE_FreePropertyEnumerator</td>
<td>0x202</td>
<td>514</td>
</tr>
<tr>
<td></td>
<td>TEE_GetNextProperty</td>
<td>0x203</td>
<td>515</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsBinaryBlock</td>
<td>0x204</td>
<td>516</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsBool</td>
<td>0x205</td>
<td>517</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsIdentity</td>
<td>0x206</td>
<td>518</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsString</td>
<td>0x207</td>
<td>519</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsU32</td>
<td>0x208</td>
<td>520</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsU64</td>
<td>0x20D</td>
<td>525</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyAsUUID</td>
<td>0x209</td>
<td>521</td>
</tr>
<tr>
<td></td>
<td>TEE_GetPropertyName</td>
<td>0x20A</td>
<td>522</td>
</tr>
<tr>
<td></td>
<td>TEE_ResetPropertyEnumerator</td>
<td>0x20B</td>
<td>523</td>
</tr>
<tr>
<td></td>
<td>TEE_StartPropertyEnumerator</td>
<td>0x20C</td>
<td>524</td>
</tr>
<tr>
<td>Panic Function</td>
<td>TEE_Panic</td>
<td>0x301</td>
<td>769</td>
</tr>
<tr>
<td>Internal Client API</td>
<td>TEE_CloseTASession</td>
<td>0x401</td>
<td>1025</td>
</tr>
<tr>
<td></td>
<td>TEE_InvokeTACmd</td>
<td>0x402</td>
<td>1026</td>
</tr>
<tr>
<td></td>
<td>TEE_OpenTASession</td>
<td>0x403</td>
<td>1027</td>
</tr>
<tr>
<td>Cancellation</td>
<td>TEE_GetCancellationFlag</td>
<td>0x501</td>
<td>1281</td>
</tr>
<tr>
<td></td>
<td>TEE_MaskCancellation</td>
<td>0x502</td>
<td>1282</td>
</tr>
<tr>
<td></td>
<td>TEE_UnmaskCancellation</td>
<td>0x503</td>
<td>1283</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Function Number in hexadecimal</td>
<td>Function Number in decimal</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------</td>
<td>--------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Memory Management</td>
<td>TEE_CheckMemoryAccessRights</td>
<td>0x601</td>
<td>1537</td>
</tr>
<tr>
<td></td>
<td>TEE_Free</td>
<td>0x602</td>
<td>1538</td>
</tr>
<tr>
<td></td>
<td>TEE_GetInstanceData</td>
<td>0x603</td>
<td>1539</td>
</tr>
<tr>
<td></td>
<td>TEEMalloc</td>
<td>0x604</td>
<td>1540</td>
</tr>
<tr>
<td></td>
<td>TEE_MemCompare</td>
<td>0x605</td>
<td>1541</td>
</tr>
<tr>
<td></td>
<td>TEE_MemFill</td>
<td>0x606</td>
<td>1542</td>
</tr>
<tr>
<td></td>
<td>TEE_MemMove</td>
<td>0x607</td>
<td>1543</td>
</tr>
<tr>
<td></td>
<td>TEE_Realloc</td>
<td>0x608</td>
<td>1544</td>
</tr>
<tr>
<td></td>
<td>TEE_SetInstanceData</td>
<td>0x609</td>
<td>1545</td>
</tr>
<tr>
<td>Generic Object</td>
<td>TEE_CloseObject</td>
<td>0x701</td>
<td>1793</td>
</tr>
<tr>
<td></td>
<td>TEE_GetObjectBufferAttribute</td>
<td>0x702</td>
<td>1794</td>
</tr>
<tr>
<td></td>
<td>TEE_GetObjectInfo (deprecated)</td>
<td>0x703</td>
<td>1795</td>
</tr>
<tr>
<td></td>
<td>TEE_GetObjectValueAttribute</td>
<td>0x704</td>
<td>1796</td>
</tr>
<tr>
<td></td>
<td>TEE_RestrictObjectUsage (deprecated)</td>
<td>0x705</td>
<td>1797</td>
</tr>
<tr>
<td></td>
<td>TEE_GetObjectInfo1</td>
<td>0x706</td>
<td>1798</td>
</tr>
<tr>
<td></td>
<td>TEE_RestrictObjectUsage1</td>
<td>0x707</td>
<td>1799</td>
</tr>
<tr>
<td>Transient Object</td>
<td>TEE_AllocateTransientObject</td>
<td>0x801</td>
<td>2049</td>
</tr>
<tr>
<td></td>
<td>TEE_CopyObjectAttributes (deprecated)</td>
<td>0x802</td>
<td>2050</td>
</tr>
<tr>
<td></td>
<td>TEE_FreeTransientObject</td>
<td>0x803</td>
<td>2051</td>
</tr>
<tr>
<td></td>
<td>TEE_GenerateKey</td>
<td>0x804</td>
<td>2052</td>
</tr>
<tr>
<td></td>
<td>TEE_InitRefAttribute</td>
<td>0x805</td>
<td>2053</td>
</tr>
<tr>
<td></td>
<td>TEE_InitValueAttribute</td>
<td>0x806</td>
<td>2054</td>
</tr>
<tr>
<td></td>
<td>TEE_PopulateTransientObject</td>
<td>0x807</td>
<td>2055</td>
</tr>
<tr>
<td></td>
<td>TEE_ResetTransientObject</td>
<td>0x808</td>
<td>2056</td>
</tr>
<tr>
<td></td>
<td>TEE_CopyObjectAttributes1</td>
<td>0x809</td>
<td>2057</td>
</tr>
<tr>
<td>Persistent Object</td>
<td>TEE_CloseAndDeletePersistentObject</td>
<td>0x901</td>
<td>2305</td>
</tr>
<tr>
<td></td>
<td>TEE_CreatePersistentObject</td>
<td>0x902</td>
<td>2306</td>
</tr>
<tr>
<td></td>
<td>TEE_OpenPersistentObject</td>
<td>0x903</td>
<td>2307</td>
</tr>
<tr>
<td></td>
<td>TEE_RenamePersistentObject</td>
<td>0x904</td>
<td>2308</td>
</tr>
<tr>
<td></td>
<td>TEE_CloseAndDeletePersistentObject1</td>
<td>0x905</td>
<td>2309</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Function Number in hexadecimal</td>
<td>Function Number in decimal</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
<td>--------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Persistent Object</td>
<td>TEE_AllocatePersistentObjectEnumerator</td>
<td>0xA01</td>
<td>2561</td>
</tr>
<tr>
<td>Enumeration</td>
<td>TEE_FreePersistentObjectEnumerator</td>
<td>0xA02</td>
<td>2562</td>
</tr>
<tr>
<td></td>
<td>TEE_GetNextPersistentObject</td>
<td>0xA03</td>
<td>2563</td>
</tr>
<tr>
<td></td>
<td>TEE_ResetPersistentObjectEnumerator</td>
<td>0xA04</td>
<td>2564</td>
</tr>
<tr>
<td></td>
<td>TEE_StartPersistentObjectEnumerator</td>
<td>0xA05</td>
<td>2565</td>
</tr>
<tr>
<td>Data Stream Access</td>
<td>TEE_ReadObjectData</td>
<td>0xB01</td>
<td>2817</td>
</tr>
<tr>
<td></td>
<td>TEE_SeekObjectData</td>
<td>0xB02</td>
<td>2818</td>
</tr>
<tr>
<td></td>
<td>TEE_TruncateObjectData</td>
<td>0xB03</td>
<td>2819</td>
</tr>
<tr>
<td></td>
<td>TEE_WriteObjectData</td>
<td>0xB04</td>
<td>2820</td>
</tr>
<tr>
<td>Generic Operation</td>
<td>TEE_AllocateOperation</td>
<td>0xC01</td>
<td>3073</td>
</tr>
<tr>
<td></td>
<td>TEE_CopyOperation</td>
<td>0xC02</td>
<td>3074</td>
</tr>
<tr>
<td></td>
<td>TEE_FreeOperation</td>
<td>0xC03</td>
<td>3075</td>
</tr>
<tr>
<td></td>
<td>TEE_GetOperationInfo</td>
<td>0xC04</td>
<td>3076</td>
</tr>
<tr>
<td></td>
<td>TEE_ResetOperation</td>
<td>0xC05</td>
<td>3077</td>
</tr>
<tr>
<td></td>
<td>TEE_SetOperationKey</td>
<td>0xC06</td>
<td>3078</td>
</tr>
<tr>
<td></td>
<td>TEE_SetOperationKey2</td>
<td>0xC07</td>
<td>3079</td>
</tr>
<tr>
<td></td>
<td>TEE_GetOperationInfoMultiple</td>
<td>0xC08</td>
<td>3080</td>
</tr>
<tr>
<td></td>
<td>TEE_IsAlgorithmSupported</td>
<td>0xC09</td>
<td>3081</td>
</tr>
<tr>
<td>Message Digest</td>
<td>TEE_DigestDoFinal</td>
<td>0xD01</td>
<td>3329</td>
</tr>
<tr>
<td></td>
<td>TEE_DigestUpdate</td>
<td>0xD02</td>
<td>3330</td>
</tr>
<tr>
<td>Symmetric Cipher</td>
<td>TEE_CipherDoFinal</td>
<td>0xE01</td>
<td>3585</td>
</tr>
<tr>
<td></td>
<td>TEE_CipherInit</td>
<td>0xE02</td>
<td>3586</td>
</tr>
<tr>
<td></td>
<td>TEE_CipherUpdate</td>
<td>0xE03</td>
<td>3587</td>
</tr>
<tr>
<td>MAC</td>
<td>TEE_MACCompareFinal</td>
<td>0xF01</td>
<td>3841</td>
</tr>
<tr>
<td></td>
<td>TEE_MACComputeFinal</td>
<td>0xF02</td>
<td>3842</td>
</tr>
<tr>
<td></td>
<td>TEE_MACInit</td>
<td>0xF03</td>
<td>3843</td>
</tr>
<tr>
<td></td>
<td>TEE_MACUpdate</td>
<td>0xF04</td>
<td>3844</td>
</tr>
<tr>
<td>Authenticated Encryption</td>
<td>TEE_AEDecryptFinal</td>
<td>0x1001</td>
<td>4097</td>
</tr>
<tr>
<td></td>
<td>TEE_AEEncryptFinal</td>
<td>0x1002</td>
<td>4098</td>
</tr>
<tr>
<td></td>
<td>TEE_AEInit</td>
<td>0x1003</td>
<td>4099</td>
</tr>
<tr>
<td></td>
<td>TEE_AEUpdate</td>
<td>0x1004</td>
<td>4100</td>
</tr>
<tr>
<td></td>
<td>TEE_AEUpdateAAD</td>
<td>0x1005</td>
<td>4101</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Function Number in hexadecimal</td>
<td>Function Number in decimal</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Asymmetric</td>
<td>TEE_AsymmetricDecrypt</td>
<td>0x1101</td>
<td>4353</td>
</tr>
<tr>
<td></td>
<td>TEE_AsymmetricEncrypt</td>
<td>0x1102</td>
<td>4354</td>
</tr>
<tr>
<td></td>
<td>TEE_AsymmetricSignDigest</td>
<td>0x1103</td>
<td>4355</td>
</tr>
<tr>
<td></td>
<td>TEE_AsymmetricVerifyDigest</td>
<td>0x1104</td>
<td>4356</td>
</tr>
<tr>
<td>Key Derivation</td>
<td>TEE_DeriveKey</td>
<td>0x1201</td>
<td>4609</td>
</tr>
<tr>
<td>Random Data Generation</td>
<td>TEE_GenerateRandom</td>
<td>0x1301</td>
<td>4865</td>
</tr>
<tr>
<td>Time</td>
<td>TEE_GetREETime</td>
<td>0x1401</td>
<td>5121</td>
</tr>
<tr>
<td></td>
<td>TEE_GetSystemTime</td>
<td>0x1402</td>
<td>5122</td>
</tr>
<tr>
<td></td>
<td>TEE_GetTAPersistentTime</td>
<td>0x1403</td>
<td>5123</td>
</tr>
<tr>
<td></td>
<td>TEE_SetTAPersistentTime</td>
<td>0x1404</td>
<td>5124</td>
</tr>
<tr>
<td></td>
<td>TEE_Wait</td>
<td>0x1405</td>
<td>5125</td>
</tr>
<tr>
<td>Memory Allocation and Size of Objects</td>
<td>TEE_BigIntFMMContextSizeInU32</td>
<td>0x1502</td>
<td>5377</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntFMMSizeInU32</td>
<td>0x1501</td>
<td>5378</td>
</tr>
<tr>
<td>Initialization</td>
<td>TEE_BigIntInit</td>
<td>0x1601</td>
<td>5633</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntInitFMM</td>
<td>0x1602</td>
<td>5634</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntInitFMMContext</td>
<td>0x1603</td>
<td>5635</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntInitFMMContext1</td>
<td>0x1604</td>
<td>5636</td>
</tr>
<tr>
<td>Converter</td>
<td>TEE_BigIntConvertFromOctetString</td>
<td>0x1701</td>
<td>5889</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntConvertFromS32</td>
<td>0x1702</td>
<td>5890</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntConvertToOctetString</td>
<td>0x1703</td>
<td>5891</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntConvertToS32</td>
<td>0x1704</td>
<td>5892</td>
</tr>
<tr>
<td>Logical Operation</td>
<td>TEE_BigIntCmp</td>
<td>0x1801</td>
<td>6145</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntCmpS32</td>
<td>0x1802</td>
<td>6146</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntGetBit</td>
<td>0x1803</td>
<td>6147</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntGetBitCount</td>
<td>0x1804</td>
<td>6148</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntShiftRight</td>
<td>0x1805</td>
<td>6149</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSet</td>
<td>0x1806</td>
<td>6150</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSet</td>
<td>0x1807</td>
<td>6151</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntAbs</td>
<td>0x1808</td>
<td>6152</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Function Number in hexadecimal</td>
<td>Function Number in decimal</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Basic Arithmetic</td>
<td>TEE_BigIntAdd</td>
<td>0x1901</td>
<td>6401</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntDiv</td>
<td>0x1902</td>
<td>6402</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntMul</td>
<td>0x1903</td>
<td>6403</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntNeg</td>
<td>0x1904</td>
<td>6404</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSquare</td>
<td>0x1905</td>
<td>6405</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSub</td>
<td>0x1906</td>
<td>6406</td>
</tr>
<tr>
<td>Modular Arithmetic</td>
<td>TEE_BigIntAddMod</td>
<td>0x1A01</td>
<td>6657</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntInvMod</td>
<td>0x1A02</td>
<td>6658</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntMod</td>
<td>0x1A03</td>
<td>6659</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntMulMod</td>
<td>0x1A04</td>
<td>6660</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSquareMod</td>
<td>0x1A05</td>
<td>6661</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntSubMod</td>
<td>0x1A06</td>
<td>6662</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntExpMod</td>
<td>0x1A07</td>
<td>6663</td>
</tr>
<tr>
<td>Other Arithmetic</td>
<td>TEE_BigIntComputeExtendedGcd</td>
<td>0x1B01</td>
<td>6913</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntIsProbablePrime</td>
<td>0x1B02</td>
<td>6914</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntRelativePrime</td>
<td>0x1B03</td>
<td>6915</td>
</tr>
<tr>
<td>Fast Modular Multiplication</td>
<td>TEE_BigIntComputeFMM</td>
<td>0x1C01</td>
<td>7169</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntConvertFromFMM</td>
<td>0x1C02</td>
<td>7170</td>
</tr>
<tr>
<td></td>
<td>TEE_BigIntConvertToFMM</td>
<td>0x1C03</td>
<td>7171</td>
</tr>
<tr>
<td>Peripherals</td>
<td>TEE_Peripheral_Close</td>
<td>0x2001</td>
<td>8193</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_CloseMultiple</td>
<td>0x2002</td>
<td>8194</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_GetPeripherals</td>
<td>0x2003</td>
<td>8195</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_GetState</td>
<td>0x2004</td>
<td>8196</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_GetStateTable</td>
<td>0x2005</td>
<td>8197</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_Open</td>
<td>0x2006</td>
<td>8198</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_OpenMultiple</td>
<td>0x2007</td>
<td>8199</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_Read</td>
<td>0x2008</td>
<td>8200</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_SetState</td>
<td>0x2009</td>
<td>8201</td>
</tr>
<tr>
<td></td>
<td>TEE_Peripheral_Write</td>
<td>0x200A</td>
<td>8202</td>
</tr>
<tr>
<td>Category</td>
<td>Function</td>
<td>Function Number in hexadecimal</td>
<td>Function Number in decimal</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>--------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Events</td>
<td>TEE_Event_AddSources</td>
<td>0x2101</td>
<td>8449</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_CancelSources</td>
<td>0x2102</td>
<td>8450</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_CloseQueue</td>
<td>0x2103</td>
<td>8451</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_DropSources</td>
<td>0x2104</td>
<td>8452</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_ListSources</td>
<td>0x2105</td>
<td>8453</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_OpenQueue</td>
<td>0x2106</td>
<td>8454</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_TimerCreate</td>
<td>0x2108</td>
<td>8456</td>
</tr>
<tr>
<td></td>
<td>TEE_Event_Wait</td>
<td>0x2107</td>
<td>8455</td>
</tr>
</tbody>
</table>
Annex B  Deprecated Functions, Identifiers, Properties, and Values

B.1  Deprecated Functions

The functions in this section are deprecated and have been replaced by new functions as noted in their descriptions. These functions will be removed at some future major revision of this specification.

Backward Compatibility

While new TA code SHOULD use the new functions, the old functions SHALL be present in an implementation until removed from the specification.

B.1.1  TEE_GetObjectInfo – Deprecated

```c
void TEE_GetObjectInfo(
    TEE_ObjectHandle      object,
    [out] TEE_ObjectInfo*       objectInfo );
```

Description

Since: TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1

Use of this function is deprecated – new code SHOULD use the TEE_GetObjectInfo1 function instead.

The TEE_GetObjectInfo function returns the characteristics of an object. It fills in the following fields in the structure TEE_ObjectInfo:

- **objectType**: The parameter objectType passed when the object was created. If the object is corrupt then this field is set to TEE_TYPE_CORRUPTED_OBJECT and the rest of the fields are set to 0.
- **objectSize**: Set to 0 for an uninitialized object
- **maxObjectSize**
  - For a persistent object, set to keySize
  - For a transient object, set to the parameter maxKeySize passed to TEE_AllocateTransientObject
- **objectUsage**: A bit vector of the TEE_USAGE_XXX bits defined in Table 5-4. Initially set to 0xFFFFFFFF.
- **dataSize**
  - For a persistent object, set to the current size of the data associated with the object
  - For a transient object, always set to 0
- **dataPosition**
  - For a persistent object, set to the current position in the data for this handle. Data positions for different handles on the same object may differ.
  - For a transient object, set to 0
- **handleFlags**: A bit vector containing one or more of the following flags:
- TEE_HANDLE_FLAG_PERSISTENT: Set for a persistent object
- TEE_HANDLE_FLAG_INITIALIZED
  - For a persistent object, always set
  - For a transient object, initially cleared, then set when the object becomes initialized
- TEE_DATA_FLAG_XXX: Only for persistent objects, the flags used to open or create the object

**Parameters**

- object: Handle of the object
- objectInfo: Pointer to a structure filled with the object information

**Specification Number:** 10  **Function Number:** 0x703

**Panic Reasons**

- If object is not a valid opened object handle.
- If the Implementation detects any other error.
**B.1.2 TEE_RestrictObjectUsage – Deprecated**

```c
void TEE_RestrictObjectUsage(
    TEE_ObjectHandle object,
    uint32_t objectUsage);
```

**Description**

Since: TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1

Use of this function is deprecated – new code SHOULD use the `TEE_RestrictObjectUsage1` function instead.

The `TEE_RestrictObjectUsage` function restricts the object usage flags of an object handle to contain at most the flags passed in the `objectUsage` parameter.

For each bit in the parameter `objectUsage`:
- If the bit is set to 1, the corresponding usage flag in the object is left unchanged.
- If the bit is set to 0, the corresponding usage flag in the object is cleared.

For example, if the usage flags of the object are set to `TEE_USAGE_ENCRYPT | TEE_USAGE_DECRYPT` and if `objectUsage` is set to `TEE_USAGE_ENCRYPT | TEE_USAGE_EXTRACTABLE`, then the only remaining usage flag in the object after calling the function `TEE_RestrictObjectUsage` is `TEE_USAGE_ENCRYPT`.

Note that an object usage flag can only be cleared. Once it is cleared, it cannot be set to 1 again on a persistent object.

A transient object’s object usage flags are reset using the `TEE_ResetTransientObject` function. For a transient object, resetting the object also clears all the key material stored in the container.

For a persistent object, setting the object usage SHALL be an atomic operation. If the supplied object is persistent and corruption is detected then this function does nothing and returns. The object handle is not closed since the next use of the handle will return the corruption and delete it.

**Parameters**

- `object`: Handle on an object
- `objectUsage`: New object usage, an OR combination of one or more of the `TEE_USAGE_XXX` constants defined in Table 5-4

**Specification Number: 10  Function Number: 0x705**

**Panic Reasons**

- If `object` is not a valid opened object handle.
- If the implementation detects any other error.
B.1.3 TEE_CopyObjectAttributes – Deprecated

```c
void TEE_CopyObjectAttributes(
    TEE_ObjectHandle destObject,
    [in] TEE_ObjectHandle srcObject);
```

Description

Since: TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1 – See Backward Compatibility note below.

Use of this function is deprecated – new code SHOULD use the TEE_CopyObjectAttributes1 function instead.

The TEE_CopyObjectAttributes function populates an uninitialized object handle with the attributes of another object handle; that is, it populates the attributes of destObject with the attributes of srcObject.

It is most useful in the following situations:

- To extract the public key attributes from a key-pair object
- To copy the attributes from a persistent object into a transient object

destObject SHALL refer to an uninitialized object handle and SHALL therefore be a transient object.

The source and destination objects SHALL have compatible types and sizes in the following sense:

- The type of destObject SHALL be a subtype of srcObject, i.e. one of the conditions listed in Table 5-11 SHALL be true.
- The size of srcObject SHALL be less than or equal to the maximum size of destObject.

The effect of this function on destObject is identical to the function TEE_PopulateTransientObject except that the attributes are taken from srcObject instead of from parameters.

The object usage of destObject is set to the bitwise AND of the current object usage of destObject and the object usage of srcObject.

If the source object is corrupt then this function copies no attributes and leaves the target object uninitialized.

Parameters

- destObject: Handle on an uninitialized transient object
- srcObject: Handle on an initialized object

Specification Number: 10 Function Number: 0x802

Panic Reasons

- If srcObject is not initialized.
- If destObject is initialized.
- If the type and size of srcObject and destObject are not compatible.
- If the Implementation detects any other error.

Backward Compatibility

Versions of this specification prior to Internal Core v1.2 did not use the [in] annotation.
B.1.4 TEE_CloseAndDeletePersistentObject – Deprecated

```c
void TEE_CloseAndDeletePersistentObject( TEE_ObjectHandle object );
```

**Description**

**Since:** TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1

Use of this function is deprecated – new code SHOULD use the `TEE_CloseAndDeletePersistentObject1` function instead.

The `TEE_CloseAndDeletePersistentObject` function marks an object for deletion and closes the object handle. The object handle SHALL have been opened with the write-meta access right, which means access to the object is exclusive.

Deleting an object is atomic; once this function returns, the object is definitely deleted and no more open handles for the object exist. This SHALL be the case even if the object or the storage containing it have become corrupted.

If the storage containing the object is unavailable then this routine SHALL panic.

If `object` is `TEE_HANDLE_NULL`, the function does nothing.

**Parameters**

- `object`: The object handle

**Specification Number:** 10  **Function Number:** 0x901

**Panic Reasons**

- If `object` is not a valid handle on a persistent object opened with the write-meta access right.
- If the storage containing the object is now inaccessible
- If the Implementation detects any other error.

B.1.5 TEE_BigIntInitFMMContext - deprecated

```c
void TEE_BigIntInitFMMContext(
    [out] TEE_BigIntFMMContext *context,
    uint32_t len,
    [in] TEE_BigInt *modulus );
```

**Description**

The `TEE_BigIntInitFMMContext` function calculates the necessary prerequisites for the fast modular multiplication and stores them in a context. This function assumes that `context` points to a memory area of `len` uint32_t. This can be done for example with the following memory allocation:

```c
TEE_BigIntFMMContext* ctx;
uint_t len = TEE_BigIntFMMContextSizeInU32(bitsize);
ctx=(TEE_BigIntFMMContext *) TEE_Malloc(len * sizeof(TEE_BigIntFMMContext), 0);
```
Even though a fast multiplication might be mathematically defined for any modulus, normally there are restrictions in order for it to be fast on a computer. This specification mandates that all implementations SHALL work for all odd moduli larger than 2 and less than 2 to the power of the implementation defined property `gpd.tee.arith.maxBigIntSize`.

**Parameters**
- `context`: A pointer to the `TEE_BigIntFMMContext` to be initialized
- `len`: The size in `uint32_t` of the memory pointed to by `context`
- `modulus`: The modulus, an odd integer larger than 2 and less than 2 to the power of `gpd.tee.arith.maxBigIntSize`

**Specification Number:** 10  **Function Number:** 0x1603

**Panic Reasons**
- If the implementation detects any error.
B.2 Deprecated Identifiers

A typo introduced an incorrect object identifier. The deprecated identifier will be removed at some future major revision of this specification. Note that while new TA code SHOULD use the new identifier, the old identifier SHALL be recognized in an implementation until removed from the specification.

<table>
<thead>
<tr>
<th>Identifier in v1.1</th>
<th>Replacement Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_TYPE_CORRUPTED*</td>
<td>Since: TEE Internal Core API v1.1; deprecated in v1.1.1</td>
</tr>
<tr>
<td></td>
<td>TEE_TYPE_CORRUPTED_OBJECT</td>
</tr>
</tbody>
</table>

* As the value of the deprecated identifier was not previously formally defined, that value SHOULD be the same as the value of the Replacement Identifier. This value can be found in Table 6-13.

The following table lists deprecated algorithm identifiers and their replacements. The deprecated identifiers will be removed at some future major revision of this specification.

### Backward Compatibility

While new TA code SHOULD use the new identifiers, the old identifiers SHALL be recognized in an implementation until removed from the specification.

<table>
<thead>
<tr>
<th>Identifier in v1.1</th>
<th>Replacement Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA algorithm identifiers should be tied to the size of the digest, not the key. The key size information is provided with the key material.</td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_2048_SHA224*</td>
<td>TEE_ALG_DSA_SHA224</td>
</tr>
<tr>
<td>TEE_ALG_DSA_2048_SHA256*</td>
<td>TEE_ALG_DSA_SHA256</td>
</tr>
<tr>
<td>TEE_ALG_DSA_3072_SHA256*</td>
<td>TEE_ALG_DSA_SHA256</td>
</tr>
</tbody>
</table>

In some cases an incomplete identifier was used for DSA algorithms.

<table>
<thead>
<tr>
<th>Identifier in v1.1</th>
<th>Replacement Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG_DSA_SHA1*</td>
<td>TEE_ALG_DSA_SHA1</td>
</tr>
<tr>
<td>ALG_DSA_SHA224*</td>
<td>TEE_ALG_DSA_SHA224</td>
</tr>
<tr>
<td>Identifier in v1.1</td>
<td>Replacement Identifier</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>ALG_DSA_SHA256*</td>
<td>TEE_ALG_DSA_SHA256</td>
</tr>
</tbody>
</table>

In some cases the ECDSA algorithm was not sufficiently defined and did not indicate digest size.

| TEE_ALG_ECDSA*                     | TEE_ALG_ECDSA_SHA512                    |

ECDSA algorithm identifiers should be tied to the size of the digest, not the key. The key size information is provided with the key material.

| TEE_ALG_ECDSA_P192*                | TEE_ALG_ECDSA_SHA1                      |
| TEE_ALG_ECDSA_P224*                | TEE_ALG_ECDSA_SHA224                    |
| TEE_ALG_ECDSA_P256*                | TEE_ALG_ECDSA_SHA256                    |
| TEE_ALG_ECDSA_P384*                | TEE_ALG_ECDSA_SHA384                    |
| TEE_ALG_ECDSA_P521*                | TEE_ALG_ECDSA_SHA512                    |

A number of algorithm identifier declarations mistakenly included "_NIST" and/or the curve type. The curve type can be found in the key material.

| TEE_ALG_ECDH_NIST_P192_DERIVE_SHARED_SECRET* | TEE_ALG_ECDH_DERIVE_SHARED_SECRET         |
| TEE_ALG_ECDH_NIST_P224_DERIVE_SHARED_SECRET* | TEE_ALG_ECDH_DERIVE_SHARED_SECRET         |
| TEE_ALG_ECDH_NIST_P256_DERIVE_SHARED_SECRET* | TEE_ALG_ECDH_DERIVE_SHARED_SECRET         |
| TEE_ALG_ECDH_NIST_P384_DERIVE_SHARED_SECRET* | TEE_ALG_ECDH_DERIVE_SHARED_SECRET         |
| TEE_ALG_ECDH_NIST_P521_DERIVE_SHARED_SECRET* | TEE_ALG_ECDH_DERIVE_SHARED_SECRET         |

| TEE_ALG_ECDH_P192 | TEE_ALG_ECDH_DERIVE_SHARED_SECRET |
| TEE_ALG_ECDH_P224 | TEE_ALG_ECDH_DERIVE_SHARED_SECRET |
| TEE_ALG_ECDH_P256 | TEE_ALG_ECDH_DERIVE_SHARED_SECRET |
| TEE_ALG_ECDH_P384 | TEE_ALG_ECDH_DERIVE_SHARED_SECRET |
| TEE_ALG_ECDH_P521 | TEE_ALG_ECDH_DERIVE_SHARED_SECRET |
**B.3 Deprecated Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Replacement</th>
</tr>
</thead>
</table>
| `gpd.tee.apiversion`      | **Since:** TEE Internal API v1.0; deprecated in TEE Internal Core API v1.1.2  
                          | Deprecated in favor of `gpd.tee.internalCore.version`. |
| `gpd.tee.cryptography.ecc` | **Since:** TEE Internal Core API v1.1; deprecated in v1.2  
                          | No direct replacement. The function `TEE_IsAlgorithmSupported` can be used to determine which, if any  
                          | ECC curves are supported.                               |

---

*As the values of the deprecated algorithm identifiers were not previously formally defined, those values SHOULD be the same as the values of the Replacement Identifier. In each case, this value can be found in Table 6-11.*

*As the values of the deprecated algorithm identifiers were not previously formally defined, those values SHOULD be the same as the values of the deprecated `TEE_ALG_ECDH_Pxxx` equivalent. In each case, the particular value can be found in Table 6-11.*
### Annex C  Normative References for Algorithms

This annex provides normative references for the algorithms discussed earlier in this document.

#### Table C-1:  Normative References for Algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>References</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_AES_CTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_ECB_NOPAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_NOPAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES_CBC_MAC_PKCS5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_NOPAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DES3_CBC_MAC_PKCS5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>References</td>
<td>URL</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_V1_5_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_DSA_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>References</td>
<td>URL</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_ALG_SHA224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_ECDSA_SHA512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEE_ALG_SM2_DSA_SM3</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_SM2_KEP</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
</tbody>
</table>

Copyright © 2011-2018 GlobalPlatform, Inc. All Rights Reserved. The technology provided or described herein is subject to updates, revisions, and extensions by GlobalPlatform. Use of this information is governed by the GlobalPlatform license agreement and any use inconsistent with that agreement is strictly prohibited.
<table>
<thead>
<tr>
<th>Name</th>
<th>References</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ALG_SM2_PKE</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_SM3</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_HMAC_SM3</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_SM4_ECB_NOPAD</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_SM4_CBC_NOPAD</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>TEE_ALG_SM4_CTR</td>
<td>OCTA</td>
<td><a href="http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp">http://www.sca.gov.cn/app-zxfw/zxfw/bzgfxc.jsp</a>,</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>This specification follows a superset of both FIPS 186-2 and FIPS 186-4. Available key sizes are defined in this specification and so no key size exclusions in FIPS 186-2 or FIPS 186-4 apply to this specification. Otherwise, when applied to this specification, if FIPS 186-4 conflicts with FIPS 186-2, then FIPS 186-4 is taken as definitive.</td>
</tr>
</tbody>
</table>
Annex D Peripheral API Usage (Informative)

The following example code is informative, and is intended to provide basic usage information on the Peripheral API. Error handling is deliberately extremely simplistic and does not represent production quality code. No guarantee is made as to the quality and correctness of this code sample.

```c
#include "tee_internal_api.h"

if (TEE_CORE_API_MAJOR_VERSION != 1) && (TEE_CORE_API_MINOR_VERSION < 2)
#error "TEE Peripheral API not supported on TEE Internal Core API < 1.2"
#endif

if !defined(TEE_CORE_API_EVENT)
#error "TEE Peripheral API not supported on this platform"
#endif

#define MAX_BUFFER               (256)

// Define a proprietary serial peripheral (as no peripheral supporting the
// polled Peripheral API is defined in this document). This is purely to
// illustrate how the API is used where such a peripheral is invented.
#define PROP_PERIPHERAL_UART           (0x80000001)

#define PROP_PERIPHERAL_STATE_BAUDRATE (0x80000001)
#define PROP_PERIPHERAL_UART_BAUD9600  (0x80)

// Trivial error handling
#define ta_assert(cond, val) if (!(cond)) TEE_Panic(val)
#define TA_GETPERIPHERALS (1)
#define TA_VERSIONFAIL    (2)
#define TA_GETSTATETABLE  (3)
#define TA_FAILBAUDRATE   (4)
#define TA_FAILOPEN       (5)
#define TA_FAILWRITE      (6)

static TEE_Peripheral*      peripherals;
static TEE_PeripheralState* peripheral_state;
```
```c
void TestPeripherals()
{
    uint32_t                 ver;
    TEE_Result               res;
    size_t                   size;
    uint32_t                 max;
    TEE_PeripheralId         tee_id;
    TEE_EventSourceHandle    tee_e_handle;
    TEE_PeripheralDescriptor uart_descriptor;
    TEE_PeripheralId         uart_id;
    TEE_PeripheralHandle     uart_p_handle;
    bool                     supports_exclusive;
    bool                     supports_baudrate_change;
    uint8_t                  buf[MAX_BUFFER];

    // Get TEE peripherals table. Catch errors, but assert rather than handle.
    // First call with NULL fetches the size of the peripherals table
    res = TEE_Peripheral_GetPeripherals(&ver, NULL, &size);
    peripherals = (TEE_Peripheral*) TEE_Malloc(size);

    res = TEE_Peripheral_GetPeripherals(&ver, peripherals, &size);
    ta_assert((res == TEE_SUCCESS) && (size <= sizeof(peripherals)),
               TA_GETPERIPHERALS);

    //****************************************************************
    // Find Peripheral ID for OS pseudo-peripheral (there is only one)
    // and for the proprietary UART (there is also only one, for simplicity)
    //****************************************************************

    max = size / sizeof(TEE_Peripheral);
    for (uint32_t i = 0; i < max; i++) {
        ta_assert(peripherals[i].version == 1, TA_VERSIONFAIL);
        if (peripherals[i].periphType == TEE_PERIPHERAL_TEE) {
            tee_id = peripherals[i].id;
            tee_e_handle = peripherals[i].e_handle;
        } else if (peripherals[i].periphType == PROP_PERIPHERAL_UART) {
            uart_id = peripherals[i].id;
            uart_p_handle = peripherals[i].p_handle;
        }
    }

    // Get state of the OS pseudo-peripheral.
    // Catch errors, but assert rather than recover.
    size = sizeof(peripheral_state);
    res = TEE_Peripheral_GetStateTable(tee_id, peripheral_state, &size);
    ta_assert((res == TEE_SUCCESS) && (size <= sizeof(peripheral_state)),
               TA_GETSTATETABLE);
```

// Check if exclusive access is supported by OS pseudo-peripheral
supports_exclusive = false;
max = size / sizeof(TEE_PeripheralState);
for (uint32_t i = 0; i < max; i++) {
    if (peripheral_state[i].id == TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS) {
        supports_exclusive = peripheral_state[i].u.boolVal;
        break;
    }
}

// Set the baud rate on the proprietary UART pseudo-peripheral.

// Fetch the state table for the UART
size = sizeof(peripheral_state);
res = TEE_Peripheral_GetStateTable(uart_id, peripheral_state, &size);
ta_assert((res == TEE_SUCCESS) && (size <= sizeof(peripheral_state)), TA_GETSTATE TABLE);

// Find the state information and check it is writeable
max = size / sizeof(TEE_PeripheralState);
supports_baudrate_change = false;
uint32_t baudrate = PROP_PERIPHERAL_UART_BAUD9600;
for (uint32_t i = 0; i < max; i++) {
    if (peripheral_state[i].id == PROP_PERIPHERAL_STATE_BAUDRATE) {
        supports_baudrate_change = peripheral_state[i].u.boolVal;
        break;
    }
}

// If so, change the baud rate.
if (supports_baudrate_change) {
    res = TEE_PeripheralSetState(uart_id,
    PROP_PERIPHERAL_STATE_BAUDRATE,
    TEE_PERIPHERAL_VALUE_UINT32,
    baudrate);
ta_assert(res == TEE_SUCCESS, TA_FAILBAUDRATE);
}

// Open the UART
uart_descriptor.id = uart_id;
uart_descriptor.p_handle = TEE_INVALID_HANDLE;
uart_descriptor_e_handle = TEE_INVALID_HANDLE;
res = TEE_Peripheral_Open(&uart_descriptor);
ta_assert((res == TEE_SUCCESS) &&
    (uart_descriptor.p_handle != TEE_INVALID_HANDLE),
    TA_FAILOPEN);
// Write to the UART.
for (uint32_t i = 0; i < MAX_BUFFER; i++)
    buf[i] = i;

res = TEE_Peripheral_Write(uart_descriptor.p_handle, buf, MAX_BUFFER);

    ta_assert((res == TEE_SUCCESS), TA_FAILWRITE);
Functions

TA_CloseSessionEntryPoint, 61
TA_CreateEntryPoint, 58
TA_DestroyEntryPoint, 58
TA_InvokeCommandEntryPoint, 62
TA_OpenSessionEntryPoint, 59
TEE_AEDecryptFinal, 203
TEE_AEncryptFinal, 202
TEE_AEInit, 198
TEE_AEUpdate, 201
TEE_AEUpdateAAD, 200
TEE_AllocateOperation, 173
TEE_AllocatePersistentObjectEnumerator, 158
TEE_AllocatePropertyEnumerator, 76
TEE_AllocateTransientObject, 131
TEE_AsymmetricDecrypt, 204
TEE_AsymmetricEncrypt, 204
TEE_AsymmetricSignDigest, 206
TEE_AsymmetricVerifyDigest, 209
TEE_BigIntAdd, 253
TEE_BigIntAddMod, 261
TEE_BigIntCmp, 247
TEE_BigIntCmpS32, 247
TEE_BigIntComputeExtendedGcd, 268
TEE_BigIntComputeFMM, 272
TEE_BigIntConvertFromFMM, 271
TEE_BigIntConvertFromOctetString, 243
TEE_BigIntConvertFromS32, 245
TEE_BigIntConvertToFMM, 270
TEE_BigIntConvertToFMMContext, 244
TEE_BigIntConvertToS32, 246
TEE_BigIntDiv, 258
TEE_BigIntFMMContextSizeInU32, 237
TEE_BigIntFMMSizeInU32, 238
TEE_BigIntGetBit, 249
TEE_BigIntGetBitCount, 249
TEE_BigIntInit, 239
TEE_BigIntInitFMM, 242
TEE_BigIntInitFMMContext, 240, 326
TEE_BigIntInvMod, 265
TEE_BigIntIsProbablePrime, 269
TEE_BigIntMod, 260
TEE_BigIntMul, 256
TEE_BigIntMulMod, 263
TEE_BigIntNeg, 255
TEE_BigIntRelativePrime, 267
TEE_BigIntShiftRight, 248
TEE_BigIntSizeInU32 (macro), 236
TEE_BigIntSquare, 257
TEE_BigIntSquareMod, 264
TEE_BigIntSub, 254
TEE_BigIntSubMod, 262
TEE_CheckMemoryAccessRights, 104
TEE_CipherDoFinal, 193
TEE_CipherInit, 190
TEE_CipherUpdate, 192
TEE_CloseAndDeletePersistentObject (deprecated), 326
TEE_CloseAndDeletePersistentObject1, 156
TEE_CloseObject, 130
TEE_CloseTASession, 95
TEE_CopyObjectAttributes (deprecated), 325
TEE_CopyObjectAttributes1, 143
TEE_CopyOperation, 186
TEE_CreatePersistentObject, 151
TEE_DeriveKey, 212
TEE_DigestDoFinal, 189
TEE_DigestUpdate, 188
TEE_Free, 113
TEE_FreeOperation, 177
TEE_FreePersistentObjectEnumerator, 158
TEE_FreePropertyEnumerator, 77
TEE_FreeTransientObject, 135
TEE_GenerateKey, 145
TEE_GenerateRandom, 215
TEE_GetCancellationFlag, 101
TEEGetInstanceData, 108
TEE_GetNextPersistentObject, 161
TEE_GetNextProperty, 80
TEE_GetObjectBufferAttribute, 127
TEE_GetObjectInfo (deprecated), 322
TEE_GetObjectInfo1, 124
TEE_GetObjectValueAttribute, 129
TEE_GetOperationInfo, 178
TEE_GetOperationInfoMultiple, 179
TEE_GetPropertyAsBinaryBlock, 73
TEE_GetPropertyAsBool, 70
TEE_GetPropertyAsBinaryBlock, 73
TEE_GetPropertyAsBool, 70
TEE_GetPropertyAsIdentity, 75
TEE_GetPropertyAsString, 69
TEE_GetPropertyAsU32, 71
TEE_GetPropertyAsU64, 72
TEE_GetPropertyAsUUID, 74
TEE.GetPropertyByName, 79
TEE_GetREETime, 232
TEE_GetSystemTime, 227
TEE_GetTAPersistentTime, 229
TEE_InitRefAttribute, 141
TEE_InitValueAttribute, 141
TEE_InvokeTACommand, 97
TEE_IsAlgorithmSupported, 187
TEE_MACCompareFinal, 197
TEE_MACComputeFinal, 196
TEE_MACInit, 194
TEE_MACUpdate, 195
TEE_Malloc, 109
TEE_MaskCancellation, 103
TEE_MemCompare, 115
TEE_MemFill, 116
TEE_MemMove, 114
TEE_OpenPersistentObject, 149
TEE_OpenTASession, 94
TEE_Panic, 93
TEE_PopulateTransientObject, 136
TEE_ReadObjectData, 162

TEE_Realloc, 111
TEE_RenamePersistentObject, 157
TEE_ResetOperation, 181
TEE_ResetPersistentObjectEnumerator, 159
TEE_ResetPropertyEnumerator, 78
TEE_ResetTransientObject, 135
TEE_RestrictObjectUsage (deprecated), 324
TEE_RestrictObjectUsage1, 126
TEE_SeedObjectData, 167
TEE_SetInstanceData, 107
TEE_SetOperationKey, 182
TEE_SetOperationKey2, 184
TEE_SetTAPersistentTime, 231
TEE_StartPersistentObjectEnumerator, 160
TEE_StartPropertyEnumerator, 77
TEE_TruncateObjectData, 166
TEE_UnmaskCancellation, 103
TEE_Wait, 228
TEE_WriteObjectData, 164
Functions by Category

Asymmetric
TEE_AsymmetricDecrypt, 211
TEE_AsymmetricEncrypt, 211
TEE_AsymmetricSignDigest, 213
TEE_AsymmetricVerifyDigest, 216

Authenticated Encryption
TEE_AEDecryptFinal, 210
TEE_AEEncryptFinal, 209
TEE_AEInit, 205
TEE_AEUpdate, 208
TEE_AEUpdateAAD, 207

Basic Arithmetic
TEE_BigIntAdd, 261
TEE_BigIntDiv, 266
TEE_BigIntMul, 264
TEE_BigIntNeg, 263
TEE_BigIntSquare, 265
TEE_BigIntSub, 262

Cancellation
TEE_GetCancellationFlag, 103
TEE_MaskCancellation, 105
TEE_UnmaskCancellation, 105

Converter
TEE_BigIntConvertFromOctetString, 250
TEE_BigIntConvertFromS32, 252
TEE_BigIntConvertToOctetString, 251
TEE_BigIntConvertToS32, 253

Data Stream Access
TEE_ReadObjectData, 168
TEE_SeekObjectData, 173
TEE_TruncateObjectData, 172
TEE_WriteObjectData, 170

Deprecated
TEE_CloseAndDeletePersistentObject, 334
TEE_CopyObjectAttributes, 333
TEE_GetObjectInfo, 330
TEE_RestrictObjectUsage, 332

Fast Modular Multiplication
TEE_BigIntComputeFMM, 280
TEE_BigIntConvertFromFMM, 279
TEE_BigIntConvertToFMM, 278

Generic Object
TEE_CloseObject, 133
TEE_GetObjectBufferAttribute, 130
TEE_GetObjectInfo, 127
TEE_GetObjectValueAttribute, 132
TEE_RestrictObjectUsage, 129

Generic Operation
TEE_AllocateOperation, 180
TEE_CopyOperation, 193
TEE_FreeOperation, 184
TEE_GetOperationInfo, 185
TEE_GetOperationInfoMultiple, 186
TEE_IsAlgorithmSupported, 194
TEE_ResetOperation, 188
TEE_SetOperationKey, 189
TEE_SetOperationKey2, 191

Initialization
TEE_BigIntInit, 246
TEE_BigIntInitFMM, 249
TEE_BigIntInitFMMContext, 247, 335

Internal Client API
TEE_CloseTASession, 97
TEE_InvokeTACmd, 99
TEE_OpenTASession, 96

Key Derivation
TEE_DeriveKey, 219

Logical Operation
TEE_BigIntCmp, 254
TEE_BigIntCmpS32, 254
TEE_BigIntGetBit, 257
TEE_BigIntGetBitCount, 257
TEE_BigIntShiftRight, 256

MAC
TEE_MACCompareFinal, 204
TEE_MACComputeFinal, 203
TEE_MACInit, 201
TEE_MACUpdate, 202

Memory Allocation and Size of Objects
TEE_BigIntFMMContextSizeInU32, 244
TEE_BigIntFMMSizeInU32, 245
TEE_BigIntPtrSizeInU32 (macro), 243

Memory Management
TEE_CheckMemoryAccessRights, 107
TEE_Free, 116
TEE_GetInstanceData, 111
TEE_Malloc, 112
TEE_MemCompare, 118
TEE_MemFill, 119
TEE_MemMove, 117
TEE_Realloc, 114
TEE_SetInstanceData, 110

Message Digest
TEE_DigestDoFinal, 196
TEE_DigestUpdate, 195

Modular Arithmetic
TEE_BigIntAddMod, 269
TEE_BigIntInvMod, 273
TEE_BigIntMod, 268
TEE_BigIntMulMod, 271
TEE_BigIntSquareMod, 272
TEE_BigIntSubMod, 270
Other Arithmetic

- TEE_BigIntComputeExtendedGcd, 276
- TEE_BigIntIsProbablePrime, 277
- TEE_BigIntRelativePrime, 275

Panic Function

- TEE_Panic, 95

Persistent Object

- TEE_CloseAndDeletePersistentObject (deprecated), 334
- TEE_CloseAndDeletePersistentObject1, 160
- TEE_CreatePersistentObject, 155
- TEE_OpenPersistentObject, 153
- TEE_RenamePersistentObject, 161

Persistent Object Enumeration

- TEE_AllocatePersistentObjectEnumerator, 162
- TEE_FreePersistentObjectEnumerator, 162
- TEE_GetNextPersistentObject, 166
- TEE_ResetPersistentObjectEnumerator, 164
- TEE_StartPersistentObjectEnumerator, 165

Property Access

- TEE_AllocatePropertyEnumerator, 77
- TEE_FreePropertyEnumerator, 78
- TEE_GetNextProperty, 82
- TEE_GetPropertyAsBinaryBlock, 74
- TEE_GetPropertyAsBool, 71
- TEE_GetPropertyAsIdentity, 76
- TEE_GetPropertyAsString, 70
- TEE_GetPropertyAsU32, 72
- TEE_GetPropertyAsU64, 73
- TEE_GetPropertyAsUUID, 75
- TEE_GetPropertyName, 81
- TEE_ResetPropertyEnumerator, 80
- TEE_StartPropertyEnumerator, 78

Random Data Generation

- TEE_GenerateRandom, 222

Symmetric Cipher

- TEE_CipherDoFinal, 200
- TEE_CipherInit, 197
- TEE_CipherUpdate, 199

TA Interface

- TA_CloseSessionEntryPoint, 62
- TA_CreateEntryPoint, 58
- TA_DestroyEntryPoint, 59
- TA_InvokeCommandEntryPoint, 63
- TA_OpenSessionEntryPoint, 60

Time

- TEE_GetREETime, 239
- TEE_GetSystemTime, 234
- TEE_GetTAPersistentTime, 236
- TEE_SetTAPersistentTime, 238
- TEE_Wait, 235

Transient Object

- TEE_AllocateTransientObject, 134
- TEE_CopyObjectAttributes (deprecated), 333
- TEE_CopyObjectAttributes1, 147
- TEE_FreeTransientObject, 138
- TEE_GenerateKey, 149
- TEE_InitRefAttribute, 145
- TEE_InitValueAttribute, 145
- TEE_PopulateTransientObject, 140
- TEE_ResetTransientObject, 138