

---

# GlobalPlatform Device Technology TEE Internal Core API Specification

Version 1.1

Public Release

June 2014

Document Reference: GPD\_SPE\_010



**Copyright © 2011-2014 GlobalPlatform, Inc. All Rights Reserved.**

*Recipients of this document are invited to submit, with their comments, notification of any relevant patents or other intellectual property rights (collectively, "IPR") of which they may be aware which might be necessarily infringed by the implementation of the specification or other work product set forth in this document, and to provide supporting documentation. 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.*

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

<b>1</b>	<b>Introduction .....</b>	<b>11</b>
1.1	Audience .....	11
1.2	IPR Disclaimer.....	11
1.3	Normative References .....	12
1.4	Terminology and Definitions.....	12
1.5	Abbreviations and Notations .....	16
1.6	Revision History .....	17
<b>2</b>	<b>Overview of the TEE Internal Core API .....</b>	<b>18</b>
2.1	Trusted Applications.....	18
2.1.1	TA Interface.....	19
2.1.2	Instances, Sessions, Tasks, and Commands.....	20
2.1.3	Sequential Execution of Entry Points.....	20
2.1.4	Cancellations.....	20
2.1.5	Unexpected Client Termination.....	21
2.1.6	Instance Types.....	21
2.1.7	Configuration, Development, and Management .....	21
2.2	Error Handling .....	22
2.2.1	Normal Errors.....	22
2.2.2	Programmer Errors .....	22
2.2.3	Panics.....	23
2.3	Opaque Handles .....	24
2.4	Properties.....	25
2.5	Trusted Storage API for Data and Keys.....	26
2.6	Cryptographic Operations API .....	26
2.7	Time API.....	27
2.8	Arithmetical API.....	27
<b>3</b>	<b>Common Definitions .....</b>	<b>28</b>
3.1	Header File.....	28
3.2	Data Types.....	28
3.2.1	Basic Types.....	28
3.2.2	Bit Numbering.....	28
3.2.3	TEE_Result, TEEC_Result .....	28
3.2.4	TEE_UUID, TEEC_UUID .....	29
3.3	Constants .....	30
3.3.1	Return Code Ranges and Format.....	30
3.3.2	Return Codes .....	31
3.4	Parameter Annotations .....	32
3.4.1	[in], [out], and [inout].....	32
3.4.2	[outopt] .....	32
3.4.3	[inbuf].....	32
3.4.4	[outbuf] .....	33
3.4.5	[outbufopt] .....	33
3.4.6	[instring] and [instringopt] .....	34
3.4.7	[outstring] and [outstringopt].....	34
3.4.8	[ctx].....	34
<b>4</b>	<b>Trusted Core Framework API .....</b>	<b>35</b>
4.1	Data Types.....	36
4.1.1	TEE_Identity.....	36

4.1.2	TEE_Param .....	36
4.1.3	TEE_TASessionHandle .....	36
4.1.4	TEE_PropSetHandle .....	36
4.2	Constants .....	37
4.2.1	Parameter Types .....	37
4.2.2	Login Types .....	37
4.2.3	Origin Codes .....	37
4.2.4	Property Set Pseudo-Handles .....	38
4.2.5	Memory Access Rights .....	38
4.3	TA Interface .....	39
4.3.1	TA_CreateEntryPoint .....	43
4.3.2	TA_DestroyEntryPoint .....	43
4.3.3	TA_OpenSessionEntryPoint .....	44
4.3.4	TA_CloseSessionEntryPoint .....	46
4.3.5	TA_InvokeCommandEntryPoint .....	47
4.3.6	Operation Parameters in the TA Interface .....	48
4.4	Property Access Functions .....	52
4.4.1	TEE_GetPropertyAsString .....	54
4.4.2	TEE_GetPropertyAsBool .....	55
4.4.3	TEE_GetPropertyAsU32 .....	56
4.4.4	TEE_GetPropertyAsBinaryBlock .....	57
4.4.5	TEE_GetPropertyAsUUID .....	58
4.4.6	TEE_GetPropertyAsIdentity .....	59
4.4.7	TEE_AllocatePropertyEnumerator .....	60
4.4.8	TEE_FreePropertyEnumerator .....	60
4.4.9	TEE_StartPropertyEnumerator .....	61
4.4.10	TEE_ResetPropertyEnumerator .....	61
4.4.11	TEE_GetPropertyName .....	62
4.4.12	TEE_GetNextProperty .....	63
4.5	Trusted Application Configuration Properties .....	64
4.6	Client Properties .....	66
4.7	Implementation Properties .....	68
4.8	Panics .....	71
4.8.1	TEE_Panic .....	71
4.9	Internal Client API .....	72
4.9.1	TEE_OpenTASession .....	72
4.9.2	TEE_CloseTASession .....	73
4.9.3	TEE_InvokeTACommand .....	74
4.9.4	Operation Parameters in the Internal Client API .....	76
4.10	Cancellation Functions .....	77
4.10.1	TEE_GetCancellationFlag .....	77
4.10.2	TEE_UnmaskCancellation .....	78
4.10.3	TEE_MaskCancellation .....	78
4.11	Memory Management Functions .....	79
4.11.1	TEE_CheckMemoryAccessRights .....	79
4.11.2	TEE_SetInstanceData .....	82
4.11.3	TEE_GetInstanceData .....	82
4.11.4	TEE_Malloc .....	83
4.11.5	TEE_Realloc .....	85
4.11.6	TEE_Free .....	86
4.11.7	TEE_MemMove .....	86
4.11.8	TEE_MemCompare .....	87

4.11.9	TEE_MemFill .....	88
<b>5</b>	<b>Trusted Storage API for Data and Keys .....</b>	<b>89</b>
5.1	Summary of Features and Design .....	89
5.2	Trusted Storage and Rollback Detection .....	91
5.3	Data Types .....	92
5.3.1	TEE_Attribute .....	92
5.3.2	TEE_ObjectInfo .....	92
5.3.3	TEE_Whence .....	93
5.3.4	TEE_ObjectHandle .....	93
5.3.5	TEE_ObjectEnumHandle .....	93
5.4	Constants .....	94
5.5	Generic Object Functions .....	96
5.5.1	TEE_GetObjectInfo1 .....	96
5.5.2	TEE_RestrictObjectUsage1 .....	98
5.5.3	TEE_GetObjectBufferAttribute .....	99
5.5.4	TEE_GetObjectValueAttribute .....	100
5.5.5	TEE_CloseObject .....	101
5.6	Transient Object Functions .....	102
5.6.1	TEE_AllocateTransientObject .....	102
5.6.2	TEE_FreeTransientObject .....	105
5.6.3	TEE_ResetTransientObject .....	106
5.6.4	TEE_PopulateTransientObject .....	107
5.6.5	TEE_InitRefAttribute, TEE_InitValueAttribute .....	111
5.6.6	TEE_CopyObjectAttributes1 .....	112
5.6.7	TEE_GenerateKey .....	114
5.7	Persistent Object Functions .....	117
5.7.1	TEE_OpenPersistentObject .....	117
5.7.2	TEE_CreatePersistentObject .....	119
5.7.3	Persistent Object Sharing Rules .....	122
5.7.4	TEE_CloseAndDeletePersistentObject1 .....	124
5.7.5	TEE_RenamePersistentObject .....	125
5.8	Persistent Object Enumeration Functions .....	126
5.8.1	TEE_AllocatePersistentObjectEnumerator .....	126
5.8.2	TEE_FreePersistentObjectEnumerator .....	127
5.8.3	TEE_ResetPersistentObjectEnumerator .....	128
5.8.4	TEE_StartPersistentObjectEnumerator .....	129
5.8.5	TEE_GetNextPersistentObject .....	130
5.9	Data Stream Access Functions .....	131
5.9.1	TEE_ReadObjectData .....	131
5.9.2	TEE_WriteObjectData .....	133
5.9.3	TEE_TruncateObjectData .....	134
5.9.4	TEE_SeekObjectData .....	135
<b>6</b>	<b>Cryptographic Operations API .....</b>	<b>136</b>
6.1	Data Types .....	138
6.1.1	TEE_OperationMode .....	138
6.1.2	TEE_OperationInfo .....	138
6.1.3	TEE_OperationInfoMultiple .....	139
6.1.4	TEE_OperationHandle .....	139
6.2	Generic Operation Functions .....	140
6.2.1	TEE_AllocateOperation .....	140
6.2.2	TEE_FreeOperation .....	144

6.2.3	TEE_GetOperationInfo.....	145
6.2.4	TEE_GetOperationInfoMultiple .....	146
6.2.5	TEE_ResetOperation .....	148
6.2.6	TEE_SetOperationKey.....	149
6.2.7	TEE_SetOperationKey2.....	151
6.2.8	TEE_CopyOperation .....	152
6.3	Message Digest Functions.....	153
6.3.1	TEE_DigestUpdate .....	153
6.3.2	TEE_DigestDoFinal.....	154
6.4	Symmetric Cipher Functions.....	155
6.4.1	TEE_CipherInit.....	155
6.4.2	TEE_CipherUpdate .....	156
6.4.3	TEE_CipherDoFinal .....	157
6.5	MAC Functions.....	158
6.5.1	TEE_MACInit.....	158
6.5.2	TEE_MACUpdate.....	159
6.5.3	TEE_MACComputeFinal.....	160
6.5.4	TEE_MACCompareFinal.....	161
6.6	Authenticated Encryption Functions .....	162
6.6.1	TEE_AEInit.....	162
6.6.2	TEE_AEUpdateAAD .....	163
6.6.3	TEE_AEUpdate.....	164
6.6.4	TEE_AEEncryptFinal .....	165
6.6.5	TEE_AEDecryptFinal .....	166
6.7	Asymmetric Functions.....	167
6.7.1	TEE_AsymmetricEncrypt, TEE_AsymmetricDecrypt.....	167
6.7.2	TEE_AsymmetricSignDigest.....	169
6.7.3	TEE_AsymmetricVerifyDigest.....	171
6.8	Key Derivation Functions .....	173
6.8.1	TEE_DeriveKey.....	173
6.9	Random Data Generation Function .....	174
6.9.1	TEE_GenerateRandom.....	174
6.10	Cryptographic Algorithms Specification .....	175
6.10.1	List of Algorithm Identifiers.....	175
6.10.2	Object Types .....	179
6.10.3	Elliptic Curve Types .....	180
6.11	Object or Operation Attributes.....	181
<b>7</b>	<b>Time API.....</b>	<b>183</b>
7.1	Data Types.....	183
7.1.1	TEE_Time .....	183
7.2	Time Functions.....	184
7.2.1	TEE_GetSystemTime .....	184
7.2.2	TEE_Wait .....	185
7.2.3	TEE_GetTAPersistentTime.....	186
7.2.4	TEE_SetTAPersistentTime .....	188
7.2.5	TEE_GetREETime .....	189
<b>8</b>	<b>TEE Arithmetical API.....</b>	<b>190</b>
8.1	Introduction.....	190
8.2	Error Handling and Parameter Checking .....	190
8.3	Data Types.....	191
8.3.1	TEE_BigInt .....	191

8.3.2	TEE_BigIntFMMContext .....	192
8.3.3	TEE_BigIntFMM .....	192
8.4	Memory Allocation and Size of Objects .....	193
8.4.1	TEE_BigIntSizeInU32 .....	193
8.4.2	TEE_BigIntFMMContextSizeInU32 .....	194
8.4.3	TEE_BigIntFMMSizeInU32 .....	194
8.5	Initialization Functions .....	195
8.5.1	TEE_BigIntInit .....	195
8.5.2	TEE_BigIntInitFMMContext .....	196
8.5.3	TEE_BigIntInitFMM .....	197
8.6	Converter Functions .....	198
8.6.1	TEE_BigIntConvertFromOctetString .....	198
8.6.2	TEE_BigIntConvertToOctetString .....	199
8.6.3	TEE_BigIntConvertFromS32 .....	200
8.6.4	TEE_BigIntConvertToS32 .....	200
8.7	Logical Operations .....	201
8.7.1	TEE_BigIntCmp .....	201
8.7.2	TEE_BigIntCmpS32 .....	201
8.7.3	TEE_BigIntShiftRight .....	202
8.7.4	TEE_BigIntGetBit .....	202
8.7.5	TEE_BigIntGetBitCount .....	203
8.8	Basic Arithmetic Operations .....	204
8.8.1	TEE_BigIntAdd .....	204
8.8.2	TEE_BigIntSub .....	205
8.8.3	TEE_BigIntNeg .....	206
8.8.4	TEE_BigIntMul .....	207
8.8.5	TEE_BigIntSquare .....	208
8.8.6	TEE_BigIntDiv .....	209
8.9	Modular Arithmetic Operations .....	210
8.9.1	TEE_BigIntMod .....	210
8.9.2	TEE_BigIntAddMod .....	211
8.9.3	TEE_BigIntSubMod .....	212
8.9.4	TEE_BigIntMulMod .....	213
8.9.5	TEE_BigIntSquareMod .....	214
8.9.6	TEE_BigIntInvMod .....	215
8.10	Other Arithmetic Operations .....	216
8.10.1	TEE_BigIntRelativePrime .....	216
8.10.2	TEE_BigIntComputeExtendedGcd .....	217
8.10.3	TEE_BigIntIsProbablePrime .....	218
8.11	Fast Modular Multiplication Operations .....	219
8.11.1	TEE_BigIntConvertToFMM .....	219
8.11.2	TEE_BigIntConvertFromFMM .....	220
8.11.3	TEE_BigIntComputeFMM .....	221
<b>Annex A</b>	<b>Panicked Function Identification .....</b>	<b>222</b>
<b>Annex B</b>	<b>Deprecated Functions .....</b>	<b>227</b>
B.1.1	TEE_GetObjectInfo – Deprecated .....	227
B.1.2	TEE_RestrictObjectUsage – Deprecated .....	229
B.1.3	TEE_CopyObjectAttributes – Deprecated .....	230
B.1.4	TEE_CloseAndDeletePersistentObject - Deprecated .....	231
<b>Annex C</b>	<b>Normative References for Algorithms .....</b>	<b>232</b>

**Functions..... 236**

**Functions by Category ..... 238**



## Figures

Figure 2-1: Trusted Application Interactions with the Trusted OS.....	19
Figure 7-1: Persistent Time Status State Machine.....	186

## Tables

Table 1-1: Normative References.....	12
Table 1-2: Terminology and Definitions.....	12
Table 1-3: Abbreviations.....	16
Table 1-4: Revision History .....	17
Table 2-1: Handle Types .....	24
Table 3-1: UUID Usage Reservations .....	29
Table 3-2: Return Code Formats and Ranges .....	30
Table 3-3: API Return Codes .....	31
Table 4-1: Parameter Type Constants .....	37
Table 4-2: Login Type Constants .....	37
Table 4-3: Origin Code Constants .....	37
Table 4-4: Property Set Pseudo-Handle Constants .....	38
Table 4-5: Memory Access Rights Constants .....	38
Table 4-6: TA Interface Functions .....	39
Table 4-7: Effect of Client Operation on TA Interface .....	40
Table 4-8: Content of <code>params[i]</code> when Trusted Application Entry Point Is Called.....	49
Table 4-9: Interpretation of <code>params[i]</code> when Trusted Application Entry Point Returns .....	50
Table 4-10: Property Sets.....	52
Table 4-11: Trusted Application Standard Configuration Properties.....	64
Table 4-12: Standard Client Properties .....	66
Table 4-13: Client Identities.....	66
Table 4-14: Implementation Properties .....	68
Table 4-15: Interpretation of <code>params[i]</code> on Entry to Internal Client API.....	76
Table 4-16: Effects of Internal Client API on <code>params[i]</code> .....	76
Table 4-17: Valid Hint Values .....	83
Table 5-1: Values of <code>gpd.tee.trustedStorage.rollbackDetection.protectionLevel</code> .....	91
Table 5-2: Object Storage Constants .....	94
Table 5-3: Data Flag Constants.....	94
Table 5-4: Usage Constants.....	94

Table 5-5: Handle Flag Constants .....	94
Table 5-6: Operation Constants .....	95
Table 5-7: Operation States .....	95
Table 5-8: Miscellaneous Constants .....	95
Table 5-9: TEE_AllocateTransientObject Object Types and Key Sizes .....	103
Table 5-10: TEE_PopulateTransientObject Supported Attributes .....	107
Table 5-11: TEE_CopyObjectAttributes1 Parameter Types .....	112
Table 5-12: TEE_GenerateKey Parameters .....	114
Table 5-13: Effect of TEE_DATA_FLAG_OVERWRITE on Behavior of TEE_CreatePersistentObject .....	120
Table 5-14: TEE_OpenPersistentObject Sharing Rules .....	123
Table 6-1: Supported Cryptographic Algorithms .....	136
Table 6-2: ECC Cryptographic Algorithms .....	137
Table 6-3: Possible TEE_OperationMode Values .....	138
Table 6-4: TEE_AllocateOperation Allowed Modes .....	141
Table 6-5: Public Key Allowed Modes .....	149
Table 6-6: Key-Pair Parts for Operation Modes .....	150
Table 6-7: Asymmetric Encrypt/Decrypt Operation Parameters .....	167
Table 6-8: Asymmetric Sign Operation Parameters .....	169
Table 6-9: Asymmetric Verify Operation Parameters .....	171
Table 6-10: Asymmetric Derivation Operation Parameters .....	173
Table 6-11: List of Algorithm Identifiers .....	175
Table 6-12: Structure of Algorithm Identifier .....	178
Table 6-13: List of Object Types .....	179
Table 6-14: List of Supported ECC Curves .....	180
Table 6-15: Object or Operation Attributes .....	181
Table 6-16: Attribute Format Definitions .....	182
Table 6-17: Partial Structure of Attribute Identifier .....	182
Table 6-18: Attribute Identifier Flags .....	182
Table 7-1: Values of the <code>gpd.tee.systemTime.protectionLevel</code> Property .....	184
Table 7-2: Values of the <code>gpd.tee.TAPersistentTime.protectionLevel</code> Property .....	187
Table A-1: Function Identification Values .....	222
Table C-1: Normative References for Algorithms .....	233

# 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 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 [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>. 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 Normative References

See also Annex C: Normative References for Algorithms.

**Table 1-1: Normative References**

Standard / Specification	Description	Ref
GPD_SPE_007	GlobalPlatform Device Technology TEE Client API Specification	[Client API]
GPD_SPE_009	GlobalPlatform Device Technology TEE System Architecture	[Sys Arch]
GPD_SPE_025	GlobalPlatform Device Technology TEE TA Debug Specification	[Debug]
GPD_SPE_027	GlobalPlatform Device Technology TEE Administration Framework	[TEE Admin]
ISO/IEC 9899:1999	Programming languages – C	[C99]
NIST Recommended Elliptic Curves	Recommended Elliptic Curves for Federal Government Use	[NIST Re Cur]
NIST SP800-56B	Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography	[NIST SP800-56B]
RFC 2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies	[RFC 2045]
RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace	[RFC 4122]

## 1.4 Terminology and Definitions

The following meanings apply to **SHALL**, **SHALL NOT**, **MUST**, **MUST NOT**, **SHOULD**, 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** indicates a recommendation.
- **MAY** indicates an option.

**Table 1-2: Terminology and Definitions**

Term	Definition
Cancellation Flag	An indicator that a Client has requested cancellation of an operation.
Client	Either of the following: <ul style="list-style-type: none"> <li>• a Client Application using the TEE Client API</li> <li>• a Trusted Application acting as a client of another Trusted Application, using the Internal Client API</li> </ul>

Term	Definition
Client Application (CA)	An application running outside of the Trusted Execution Environment making use of the TEE Client API to access facilities provided by Trusted Applications inside the Trusted Execution Environment. <i>Contrast <b>Trusted Application (TA)</b>.</i>
Client Properties	A set of properties associated with the Client of a Trusted Application.
Command	A message (including a Command Identifier and four Operation Parameters) send by a Client to a Trusted Application to initiate an operation.
Command Identifier	A 32-bit integer identifying a Command.
Cryptographic Key Object	An object containing key material.
Cryptographic Key-Pair Object	An object containing material associated with both keys of a key-pair.
Cryptographic Operation Handle	An opaque reference that identifies a particular cryptographic operation.
Cryptographic Operation Key	The key to be used for a particular operation.
Data Object	An object containing a data stream but no key material.
Data Stream	Data associated with a persistent object (excluding Object Attributes and metadata).
Function Number	Identifies a function within a specification. With the Specification Number, forms a unique identifier for a function. May be displayed when a panic occurs or in debug messages where supported.
Implementation	A particular implementation of the Trusted OS.
Initialized	Describes a transient object whose attributes have been populated.
Instance	A particular execution of a Trusted Application, having physical memory space that is separated from the physical memory space of all other TA instances.
Key Size	The key size associated with a Cryptographic Object; values are limited by the key algorithm used.
Key Usage Flags	Indicators of the operations permitted with a Cryptographic Object.
Memory Reference Parameter	An Operation Parameter that carries a pointer to a client-owned memory buffer. <i>Contrast <b>Value Parameter</b>.</i>
Metadata	Additional data associated with a Cryptographic Object: Key Size and Key Usage Flags.
Multi Instance Trusted Application	Denotes a Trusted Application for which each session opened by a client is directed to a separate TA instance.
Object Attribute	Small amounts of data used to store key material in a structured way.
Object Handle	An opaque reference that identifies a particular object.
Object Identifier	A variable-length binary buffer identifying a persistent object.

Term	Definition
Operation Parameter	One of four data items passed in a Command, which can contain integer values or references to client-owned shared memory blocks.
Panic	An exception that kills a whole TA instance as a result of calling one of the API functions.
Parameter Annotation	Denotes the pattern of usage of a function parameter or pair of function parameters.
Persistent Object	An object identified by an Object Identifier and including a Data Stream. Contrast <i>Transient Object</i> .
Property	An immutable value identified by a name.
Property Set	Any of the following: <ul style="list-style-type: none"> <li>• The configuration properties of a Trusted Application</li> <li>• Properties associated with a Client Application by the Rich Execution Environment</li> <li>• Properties describing characteristics of a TEE Implementation</li> </ul>
REE Time	A time value that is as trusted as the REE.
Rich Execution Environment (REE)	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 un-trusted. Contrast <i>Trusted Execution Environment (TEE)</i> .
Rich OS	Typically an OS providing a much wider variety of features than that of 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 the size and needs of the Rich OS it 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 has to be considered un-trusted, though from the Rich OS point of view there may be internal trust structures. Contrast <i>Trusted OS</i> .
Session	Logically connects multiple commands invoked on a Trusted Application.
Single Instance Trusted Application	Denotes a Trusted Application for which all sessions opened by clients are directed to a single TA instance.
Specification Number	Identifies the specification within which a function is defined. May be displayed when a panic occurs or in debug messages where supported.
Storage Identifier	A 32-bit identifier for a Trusted Storage Space that can be accessed by a Trusted Application.
System Time	A time value that can be used to compute time differences and operation deadlines.
TA Persistent Time	A time value set by the Trusted Application that persists across platform reboots and whose level of trust can be queried.

Term	Definition
Task	The entity that executes any code executed in a Trusted Application.
Transient Object	An object containing attributes but no data stream, which is reclaimed when closed or when the TA instance is destroyed. <i>Contrast <b>Persistent Object</b>.</i>
Trusted Application (TA)	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. <i>Contrast <b>Client Application (CA)</b>.</i>
Trusted Application Configuration Properties	A set of properties associated with the installation of a Trusted Application.
Trusted Core Framework or “Framework”	The part of the Trusted OS responsible for implementing the Trusted Core Framework API <sup>1</sup> that provides OS-like facilities to Trusted Applications and a way for the Trusted OS to interact with the Trusted Applications.
Trusted Execution Environment (TEE)	An execution environment that runs alongside but isolated from an 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. <i>Contrast <b>Rich Execution Environment (REE)</b>.</i>
Trusted OS	An operating system running in the TEE providing the TEE Internal Core API to Trusted Applications.
Trusted Storage Spaces	Storage spaces accessible only to Trusted Applications.
Uninitialized	Describes a transient object allocated with a certain object type and maximum size but with no attributes.
Universally Unique Identifier (UUID)	An identifier as specified in RFC 4122 [RFC 4122].
Value Parameter	An Operation Parameter that carries two 32-bit integers. <i>Contrast <b>Memory Reference Parameter</b>.</i>

<sup>1</sup> The Trusted Core Framework API is described in Chapter 4.

## 1.5 Abbreviations and Notations

**Table 1-3: Abbreviations**

Term	Definition
AAD	Additional Authenticated Data
AE	Authenticated Encryption
AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Client Application
CMAC	Cipher-based MAC
CRT	Chinese Remainder Theorem
CTS	CipherText Stealing
DES	Data Encryption Standard
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
gcd	Greatest Common Divisor
HMAC	Hash-based Message Authentication Code
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IPR	Intellectual Property Rights
ISO	International Organization for Standardization
IV	Initialization Vector
MAC	Message Authentication Code
MD5	Message Digest 5
MGF	Mask Generating Function
NIST	National Institute of Standards and Technology
OAEP	Optimal Asymmetric Encryption Padding
OS	Operating System
PKCS	Public Key Cryptography Standards
PSS	Probabilistic Signature Scheme
REE	Rich Execution Environment
RFC	Request For Comments; may denote a memorandum published by the IETF



Term	Definition
RSA	Rivest, Shamir, Adleman asymmetric algorithm
SHA	Secure Hash Algorithm
TA	Trusted Application
TEE	Trusted Execution Environment
UTC	Coordinated Universal Time
UTF	Unicode Transformation Format
UUID	Universally Unique Identifier
XTS	XEX-based Tweaked Codebook mode with ciphertext stealing (CTS)

## 1.6 Revision History

**Table 1-4: Revision History**

Date	Version	Description
December 2011	1.0	Initial Public Release, as “TEE Internal API Specification”.
June 2014	1.1	Public Release, as “TEE Internal Core API Specification”.

## 2 Overview of the TEE Internal Core API

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 and does not cover other language bindings.

### 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.

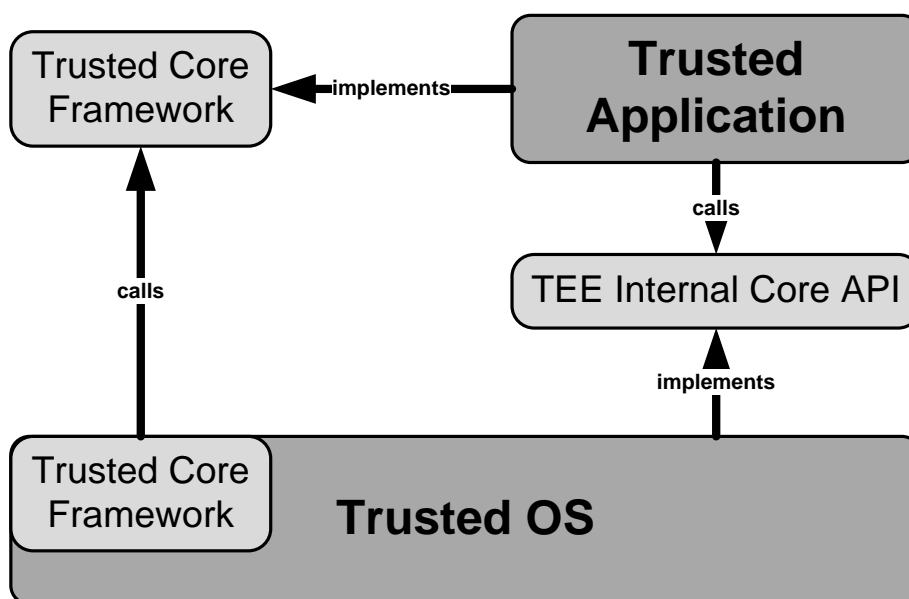
A Client typically runs in the Rich Execution Environment and communicates with a Trusted Application using the TEE Client API [Client API]. It is then called a “**Client Application**”. It is also possible for a Trusted Application to act as a client of another Trusted Application, using the Internal 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



### 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 **MUST** 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 **MUST** 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 to define the combinations of commands and their parameters that are valid to execute. 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 **MUST** 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 **MUST** 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 MUST appear to the Trusted Application as if the client requests cancellation of all pending operations and gracefully closes all its client sessions. It MUST 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 MUST initiate a termination process that MUST 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 MUST appear as if the client had requested its cancellation.
- When no operation remains pending in the session, the session MUST be closed.

If a TA client is a TA itself, this sequence of events MUST happen when the client TA panics or exits due to the termination of its own Client Application.<sup>2</sup>

### 2.1.6 Instance Types

At least two Trusted Application instance types MUST be supported: Multi Instance and Single Instance. Whether a Trusted Application is Multi Instance or Single Instance is part of its configuration properties and MUST 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 Remote Administration specification [TEE Admin] 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.

---

<sup>2</sup> Panics are discussed in section 2.2.3.

## 2.2 Error Handling

### 2.2.1 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.

### 2.2.2 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 **MUST** 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 **MUST 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 **MUST 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.

### 2.2.3 Panics

A **Panic** is an instance-wide uncatchable exception that kills a whole TA instance as a result of calling one of the API functions. It SHOULD happen when the Implementation detects an avoidable Programmer Error and there is no specifically defined error code which covers the problem. In addition the Trusted Application itself may request a panic by calling the function `TEE_Panic`.

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 MUST return the error `TEE_ERROR_TARGET_DEAD` with the origin `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], §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 [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 [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.

## 2.3 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 **MUST** always be associated with a type.

The special value `TEE_HANDLE_NULL`, which **MUST** 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 **MUST** 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 2-1: Handle Types**

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

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.



## 2.4 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 MUST 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 TEE Implementation 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 organization, 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 MUST refuse to define properties in this namespace unless they meet this specification.

## 2.5 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 **MUST** be as strong as the means used to protect the TEE code and data itself.
  - The Trusted Storage **MUST** be bound to a particular device, which means that it **MUST** 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] §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.

## 2.6 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

## 2.7 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`.

## 2.8 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.

## 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 MUST have the name “tee\_internal\_api.h”.

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

### 3.2 Data Types

#### 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]. The following basic types are used:

- `uint32_t`: Unsigned 32-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 7, 15, or 31 (most-significant bit), depending on the size of the integer.

#### 3.2.3 TEE\_Result, TEEC\_Result

```
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:

```
typedef TEE_Result TEEC_Result;
```

### 3.2.4 TEE\_UUID, TEEC\_UUID

```
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:

```
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:

```
typedef TEE_UUID TEEC_UUID;
```

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

**Table 3-1: UUID Usage Reservations**

Version	Reservation
UUID v5	When the GPD TEE Administration Framework [TEE Admin] is supported by a TEE, then TA and Security Domain (SD) UUID's using version 5 must conform to the requirements of that specification.

### 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**

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

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 3-3: API Return Codes**

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

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

The annotation `[out]` and `[inout]` are equivalent to `[in]` but for write access and read-and-write access respectively.

Note that, as described in section 4.11.1, the NULL pointer MUST never be accessible to a Trusted Application. This means that a Trusted Application MUST 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 MUST be discarded.

### 3.4.3 `[inbuf]`

The `[inbuf]` annotation applies to a pair of parameters of type `void*` and `uint32_t`. It means that the parameters describe an input data buffer. The entire buffer MUST 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 MUST check that the buffer is entirely readable and MUST 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 `void*` parameter unless the second `uint32_t` parameter is set to 0.



### 3.4.4 [outbuf]

The `[outbuf]` annotation applies to a pair of parameters of type `void*` and `uint32_t*`, herein referenced with the names `buffer` and `size`. It is used by API functions to return an output data buffer. The data buffer **MUST** 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 **MUST** be entirely writable by the Trusted Application, otherwise the Implementation **MUST** panic the calling Trusted Application instance. In any case, the implementation **MUST NOT** write beyond this limit.
- On exit:
  - If the output fits in the output buffer, then the Implementation **MUST** write the output in `buffer` and **MUST** update `*size` with the actual size of the output in bytes.
  - If the output does not fit in the output buffer, then the implementation **MUST** update `*size` with the required number of bytes and **MUST** 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 **MUST NOT** have performed the actual requested operation. It **MUST** 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` **MUST** be considered as `[inout]`. That is, `size` **MUST** be readable and writable by the Trusted Application. The parameter `size` **MUST NOT** be `NULL` and **MUST NOT** reside in shared memory. The Implementation **MUST** 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 **MUST** behave as if the output buffer was not large enough to hold the entire output data and the output data **MUST** 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 MUST 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 MUST 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.

## 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 MUST 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

```
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

```
typedef union
{
    struct
    {
        void*    buffer; uint32_t    size;
    } memref;
    struct
    {
        uint32_t a, 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_InvokeTACommand`.

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.

### 4.1.3 TEE\_TASessionHandle

```
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

```
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.

## 4.2 Constants

### 4.2.1 Parameter Types

Table 4-1: Parameter Type Constants

Constant Name	Constant Value
TEE_PARAM_TYPE_NONE	0
TEE_PARAM_TYPE_VALUE_INPUT	1
TEE_PARAM_TYPE_VALUE_OUTPUT	2
TEE_PARAM_TYPE_VALUE_INOUT	3
TEE_PARAM_TYPE_MEMREF_INPUT	5
TEE_PARAM_TYPE_MEMREF_OUTPUT	6
TEE_PARAM_TYPE_MEMREF_INOUT	7

### 4.2.2 Login Types

Table 4-2: Login Type Constants

Constant Name	Constant Value
TEE_LOGIN_PUBLIC	0x00000000
TEE_LOGIN_USER	0x00000001
TEE_LOGIN_GROUP	0x00000002
TEE_LOGIN_APPLICATION	0x00000004
TEE_LOGIN_APPLICATION_USER	0x00000005
TEE_LOGIN_APPLICATION_GROUP	0x00000006
Reserved for future GlobalPlatform defined login types	0x00000007 – 0x7FFFFFFF
Reserved for implementation-specific login types	0x80000000 – 0xFFFFFFFF
TEE_LOGIN_TRUSTED_APP	0xF0000000
Reserved for future GlobalPlatform defined login types	0xF0000001 – 0xFFFFFFFF

### 4.2.3 Origin Codes

Table 4-3: Origin Code Constants

Constant Names	Constant Value
TEE_ORIGIN_API      TEEC_ORIGIN_API	0x00000001
TEE_ORIGIN_COMMS      TEEC_ORIGIN_COMMS	0x00000002
TEE_ORIGIN_TEE      TEEC_ORIGIN_TEE	0x00000003
TEE_ORIGIN_TRUSTED_APP      TEEC_ORIGIN_TRUSTED_APP	0x00000004

## 4.2.4 Property Set Pseudo-Handles

Table 4-4: Property Set Pseudo-Handle Constants

Constant Name	Constant Value
Reserved	0x00000000 – 0xFFFFFFFF
Reserved for implementation-specific property sets	0xF0000000 – 0xFFFEFFFF
Reserved for future GlobalPlatform use	0xFFFF0000 – 0xFFFFFFFFC
TEE_PROPSET_TEE_IMPLEMENTATION	(TEE_PropSetHandle)0xFFFFFFFFD
TEE_PROPSET_CURRENT_CLIENT	(TEE_PropSetHandle)0xFFFFFFFFE
TEE_PROPSET_CURRENT_TA	(TEE_PropSetHandle)0xFFFFFFFFF

## 4.2.5 Memory Access Rights

Table 4-5: Memory Access Rights Constants

Constant Name	Constant Value
TEE_MEMORY_ACCESS_READ	0x00000001
TEE_MEMORY_ACCESS_WRITE	0x00000002
TEE_MEMORY_ACCESS_ANY_OWNER	0x00000004

## 4.3 TA Interface

Each Trusted Application MUST 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 MUST be bound to the framework before the Trusted Application code is started.

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

**Table 4-6: TA Interface Functions**

TA Interface Function (Entry Point)	Description
TA_CreateEntryPoint	This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created.
TA_DestroyEntryPoint	This is the Trusted Application destructor. The Trusted Core Framework calls this function just before the Trusted Application instance is terminated. The Framework MUST guarantee that no sessions are open when this function is called. When TA_DestroyEntryPoint returns, the Framework MUST collect all resources claimed by the Trusted Application instance.
TA_OpenSessionEntryPoint	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.
TA_CloseSessionEntryPoint	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.
TA_InvokeCommandEntryPoint	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.

Table 4-7 summarizes client operations and the resulting Trusted Application effect.

**Table 4-7: Effect of Client Operation on TA Interface**

Client Operation	Trusted Application Effect
TEEC_OpenSession or TEE_OpenTASession	If a new Trusted Application instance is needed to handle the session, TA_CreateEntryPoint is called. Then, TA_OpenSessionEntryPoint is called.
TEEC_InvokeCommand or TEE_InvokeTACommand	TA_InvokeCommandEntryPoint is called.
TEEC_CloseSession or TEE_CloseTASession	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.
TEEC_RequestCancellation or The function TEE_OpenTASession or TEE_InvokeTACommand is cancelled.	See section 4.10 for details on the effect of cancellation requests.
Client terminates unexpectedly	From the point of view of the TA instance, the behavior MUST be identical to the situation where the client does not terminate unexpectedly but, for all opened sessions: <ul style="list-style-type: none"> <li>• requests the cancellation of all pending operations in that session,</li> <li>• waits for the completion of all these operations in that session,</li> <li>• and finally closes that session.</li> </ul> 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.



## 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 MUST 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 **MUST** 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

```
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 **MUST** 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 **MUST** 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

```
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 **MUST** start automatically reclaiming resources left open.

After this function returns, the Implementation **MUST** consider the instance destroyed and **MUST** 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

```
TEE_Result TA_EXPORT 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 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 **MUST** 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

**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
  - 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 **MUST** 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

```
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

```
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.

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

#### Return Value

- `TEE_SUCCESS`: If the command is successfully executed, the function **MUST** 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 **MUST** 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 37. 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 following macros are available to decode `paramTypes`:

```
#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:

```
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 4-8: Content of `params[i]` when Trusted Application Entry Point Is Called**

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

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 MUST 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**

Value of <code>type[i]</code>	Behavior of the Framework when Entry Point Returns
TEE_PARAM_TYPE_NONE TEE_PARAM_TYPE_VALUE_INPUT TEE_PARAM_TYPE_MEMREF_INPUT	The content of <code>params[i]</code> is ignored.
TEE_PARAM_TYPE_VALUE_OUTPUT TEE_PARAM_TYPE_VALUE_INOUT	<code>params[i].value.a</code> and <code>params[i].value.b</code> contain the two integers sent to the client.
TEE_PARAM_TYPE_MEMREF_OUTPUT TEE_PARAM_TYPE_MEMREF_INOUT	The Framework reads <code>params[i].memref.size</code> : <ul style="list-style-type: none"> <li>• If it is equal or less than the original value of <code>size</code>, it is considered as the actual size of the memory buffer. 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.</li> <li>• If it is greater than the original value of <code>size</code>, 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.</li> </ul>

#### 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 **MUST 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 **MUST** be accessed only during the lifetime of the operation.
- The Trusted Application **MUST 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 **MUST** 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 4-10: Property Sets**

Pseudo-Handle	Meaning
TEE_PROPSET_CURRENT_TA	The configuration properties for the current Trusted Application. See section 4.5 for a definition of these properties.
TEE_PROPSET_CURRENT_CLIENT	The properties of the current client. This pseudo-handle is valid only in the context of the following entry points: <ul style="list-style-type: none"> <li>• TA_OpenSessionEntryPoint</li> <li>• TA_InvokeCommandEntryPoint</li> <li>• TA_CloseSessionEntryPoint</li> </ul> See section 4.6 for a definition of these properties.
TEE_PROPSET_TEE_IMPLEMENTATION	The properties of the TEE Implementation itself. See section 4.7.

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

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

Additionally, a property may be retrieved and converted into one and only one of the following types:

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

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:

- 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 unsigned integer

is converted into a string that is 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

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

```
TEE_Result TEE_GetPropertyAsString(  
    TEE_PropSetHandle propsetOrEnumerator,  
    [instringopt] char* name,  
    [outstring] char* valueBuffer, uint32_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 **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST 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 which is not explicitly associated with a defined return code for this function.

## 4.4.2 TEE\_GetPropertyAsBool

```
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 **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST 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 cannot be converted to a Boolean

### Panic Reasons

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

### 4.4.3 TEE\_GetPropertyAsU32

```
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.

If a property cannot be viewed as a 32-bit unsigned integer, this function **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST 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 cannot be converted to an unsigned 32-bit integer

#### Panic Reasons

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



#### 4.4.4 TEE\_GetPropertyAsBinaryBlock

```
TEE_Result TEE_GetPropertyAsBinaryBlock(  
    TEE_PropSetHandle propsetOrEnumerator,  
    [instringopt] char* name,  
    [outbuf] void* valueBuffer, uint32_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 **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST 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 which is not explicitly associated with a defined return code for this function.

#### 4.4.5 TEE\_GetPropertyAsUUID

```

TEE_Result TEE_GetPropertyAsUUID(
    TEE_PropSetHandle propsetOrEnumerator,
    [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 **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST NOT** be `NULL`.
- `value`: A pointer filled with the UUID. **MUST 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 which is not explicitly associated with a defined return code for this function.

#### 4.4.6 TEE\_GetPropertyAsIdentity

```
TEE_Result TEE_GetPropertyAsIdentity(  
    TEE_PropSetHandle propsetOrEnumerator,  
    [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 **MUST** 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 **MUST** be encoded in UTF-8.
  - If `propsetOrEnumerator` is a property enumerator handle, `name` is ignored and can be `NULL`.
  - Otherwise, `name` **MUST NOT** be `NULL`.
- `value`: A pointer filled with the identity. **MUST 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 which is not explicitly associated with a defined return code for this function.

#### 4.4.7 TEE\_AllocatePropertyEnumerator

```
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 which is not explicitly associated with a defined return code for this function.

#### 4.4.8 TEE\_FreePropertyEnumerator

```
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

```
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. MUST 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

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

##### Description

The function `TEE_ResetPropertyEnumerator` resets a property enumerate 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

```
TEE_Result TEE_GetPropertyName(  
    TEE_PropSetHandle enumerator,  
    [outstring] void* nameBuffer, uint32_t* nameBufferLen );
```

##### Description

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

The property name MUST be the valid UTF-8 encoding of a Unicode string containing no 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 which is not explicitly associated with a defined return code for this function.

#### 4.4.12 TEE\_GetNextProperty

```
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 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 4-11: Trusted Application Standard Configuration Properties**

Property Name	Type	Meaning
<code>gpd.ta.appID</code>	UUID	The identifier of the Trusted Application.
<code>gpd.ta.singleInstance</code>	Boolean	Whether the Implementation SHALL create a single TA instance for all the client sessions (if <code>true</code> ) or SHALL create a separate instance for each client session (if <code>false</code> ).
<code>gpd.ta.multiSession</code>	Boolean	Whether the Trusted Application instance supports multiple sessions.  This property is ignored for multi-instance Trusted Applications.
<code>gpd.ta.instanceKeepAlive</code>	Boolean	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 <code>gpd.ta.singleInstance</code> is set to <code>true</code> .  When this property is set to <code>false</code> , then the TA instance MUST be created when one or more sessions are opened on the TA and it MUST be destroyed when there are no more sessions opened on the instance.  When this property is set to <code>true</code> , 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 MUST 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 MUST be no later than the first session opening.



Property Name	Type	Meaning
<code>gpd.ta.dataSize</code>	Integer	Maximum estimated amount of dynamic data in bytes configured for the Trusted Application. The memory blocks allocated through <code>TEE_Malloc</code> 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.
<code>gpd.ta.stackSize</code>	Integer	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 <b>MUST</b> 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.
<code>gpd.ta.version</code>	String	Version number of this Trusted Application.
<code>gpd.ta.description</code>	String	Optional description of the Trusted Application

## 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 MUST 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**

Property Name	Type	Meaning
gpd.client.identity	Identity	Identity of the current client. This can be conveniently retrieved using the function TEE_GetPropertyAsIdentity (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.

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.

**Table 4-13: Client Identities**

Login Method	Meaning
TEE_LOGIN_PUBLIC	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].
TEE_LOGIN_APPLICATION	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.
TEE_LOGIN_USER	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.
TEE_LOGIN_GROUP	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.
TEE_LOGIN_APPLICATION_USER	The client UUID identifies both the calling application and the user that is executing it.
TEE_LOGIN_APPLICATION_GROUP	The client UUID identifies both the calling application and a group that is executing it.

Login Method	Meaning
TEE_LOGIN_TRUSTED_APP	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.
The range 0x80000000-0xFFFFFFFF is reserved for <i>implementation-defined</i> login methods.	The meaning of the Client UUID and the associated client properties are <i>implementation-defined</i> . 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.
The ranges 0x00000000-0x7FFFFFFF and 0xF0000000-0xFFFFFFFF are reserved for standard login methods defined by GlobalPlatform.	

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

- The property `gpd.client.identity` MUST always be determined by the Trusted OS and the determination of whether it is equal to `TEE_LOGIN_TRUSTED_APP` or not MUST be as trustworthy as the Trusted OS itself.
- When `gpd.client.identity` is equal to `TEE_LOGIN_TRUSTED_APP` then the Trusted OS MUST 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 `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 MUST 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 4-14: Implementation Properties**

Property Name	Type	Meaning
gpd.tee.apiversion	String	The version number of the API implementation. Its value for this version of the specification is the string “1.1”.
gpd.tee.description	String	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.  Note that implementations are free to define their own non-standard identification property names, provided they are not in the “gpd.” namespace
gpd.tee.deviceID	UUID	A device identifier that <b>MUST</b> be globally unique among all GlobalPlatform TEEs whatever the manufacturer, vendor, or integration.  <b>Implementer’s Note</b>  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 <b>MUST</b> therefore be carefully designed so that it does not compromise secret information.
gpd.tee.systemTime.protectionLevel	Integer	The protection level provided by the system time implementation. See the function TEE_GetSystemTime in section 7.2.1 for more details.
gpd.tee.TAPersistentTime.protectionLevel	Integer	The protection level provided for the TA Persistent Time. See the function TEE_GetTAPersistentTime in section 7.2.3 for more details.

Property Name	Type	Meaning
<code>gpd.tee.arith.maxBigIntSize</code>	Integer	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.
<code>gpd.tee.cryptography.ecc</code>	Boolean	If set to True, then the Elliptic Curve Cryptographic (ECC) algorithms shown in Table 6-2 are supported.
<code>gpd.tee.trustedStorage. antiRollback.protectionLevel</code>	Integer	Indicates the level of protection from rollback of Trusted Storage supplied by the implementation:  <b>0 (or missing):</b> No anti rollback protection <b>100:</b> Anti rollback mechanism for the Trusted Storage is enforced at the REE level. <b>1000:</b> Anti rollback mechanism for the Trusted Storage is based on TEE-controlled hardware. This hardware MUST be out of reach of software attacks from the REE. Users may still be able to roll back the Trusted Storage but this MUST be detected by the Implementation
<code>gpd.tee.trustedos.implementation. version</code>	String	The detailed version number of the TEE implementation.  The value of this property MUST change whenever anything changes in the code forming the Trusted OS which provides the TEE, i.e. any patch MUST change this string.
<code>gpd.tee.trustedos.implementation. binaryversion</code>	binary	A binary value which is equivalent to <code>gpd.tee.trustedos.implementation.version</code> . 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 <code>gpd.tee.trustedos.implementation.version</code> , 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 MUST change whenever anything changes in the code forming the Trusted OS which provides the TEE, i.e. any patch MUST change this string.
<code>gpd.tee.trustedos.manufacturer</code>	String	Name of the manufacturer of the Trusted OS.

Property Name	Type	Meaning
<code>gpd.tee.firmware.implementation.version</code>	String	<p>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.</p> <p>The value of this property MUST change whenever anything changes in this code, i.e. any patch MUST change this string. The value of this property MAY be the empty string if there is no such software.</p>
<code>gpd.tee.firmware.implementation.binaryversion</code>	Binary	<p>A binary value which is equivalent to <code>gpd.tee.firmware.implementation.version</code>. 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 <code>gpd.tee.firmware.implementation.version</code>, 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.</p> <p>The value of this property MUST change whenever anything changes in this code, i.e. any patch MUST change this string. The value of this property MAY be a zero length value if there is no such firmware.</p>
<code>gpd.tee.firmware.manufacturer</code>	String	<p>Name of the manufacturer of the firmware which supports the Trusted OS or the empty string if there is no such firmware.</p>

## 4.8 Panics

### 4.8.1 TEE\_Panic

```
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 **MUST** be destroyed and all the resources opened by the instance **MUST** be reclaimed. All sessions opened from the panicking instance on another TA **MUST** be gracefully closed and all cryptographic objects and operations **MUST** 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 [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

```
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`. This UUID can be hardcoded in the caller code.

An initial 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 Value” 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 MUST 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 MUST 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.



**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

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.

## 4.9.2 TEE\_CloseTASession

```
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

```

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` MUST 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 Value” 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 MUST 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 MUST 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.

**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
- 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 is not explicitly associated with a defined return code for this function.

#### 4.9.4 Operation Parameters in the Internal Client API

The functions `TEE_OpenTASession` and `TEE_InvokeTACommand` 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 37. 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` MUST NOT be NULL and MUST 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**

Parameter Type	Interpretation of <code>params[i]</code>
<code>TEE_PARAM_TYPE_NONE</code> <code>TEE_PARAM_TYPE_VALUE_OUTPUT</code>	Ignored.
<code>TEE_PARAM_TYPE_VALUE_INPUT</code> <code>TEE_PARAM_TYPE_VALUE_INOUT</code>	Contains two integers in <code>params[i].value.a</code> and <code>params[i].value.b</code> .
<code>TEE_PARAM_TYPE_MEMREF_INPUT</code> <code>TEE_PARAM_TYPE_MEMREF_OUTPUT</code> <code>TEE_PARAM_TYPE_MEMREF_INOUT</code>	<code>params[i].memref.buffer</code> and <code>params[i].memref.size</code> MUST be initialized with a memory buffer that is accessible with the access rights described in the type. The buffer can be NULL, in which case <code>size</code> MUST be set to 0.

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]`**

Parameter Type	Effects on <code>params[i]</code>
<code>TEE_PARAM_TYPE_NONE</code> <code>TEE_PARAM_TYPE_VALUE_INPUT</code> <code>TEE_PARAM_TYPE_MEMREF_INPUT</code>	Unchanged.
<code>TEE_PARAM_TYPE_VALUE_OUTPUT</code> <code>TEE_PARAM_TYPE_VALUE_INOUT</code>	<code>params[i].value.a</code> and <code>params[i].value.b</code> are updated with the value sent by the destination Trusted Application.
<code>TEE_PARAM_TYPE_MEMREF_OUTPUT</code> <code>TEE_PARAM_TYPE_MEMREF_INOUT</code>	<code>params[i].memref.size</code> is updated to reflect the actual or requested size of the buffer.

## 4.10 Cancellation Functions

This section defines functions for Trusted Applications to handle cancellation requested by a Client Application.

When a Client Application requests cancellation using the function `TEEC_RequestCancellation` of [Client API], the implementation **MUST** do the following:

- If the operation has not reached the TA yet but has been queued in the TEE, then it **MUST** be retired from the queue and fail with the return code `TEEC_ERROR_CANCEL` and the origin `TEEC_ORIGIN_TEE`.
- If the operation has been transmitted to the Trusted Application, the implementation **MUST** set the **Cancellation Flag** of the task executing the command.
- 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 a single cancellable function, which is the `TEE_Wait` function. Future versions may define other cancellable functions, in particular in the domain of user interactions.
  - The functions `TEE_OpenTASession` and `TEE_InvokeTACmd`, while not cancellable per se, **MUST** transmit cancellation requests: If the Cancellation Flag is set and the effects of cancellation are not masked, then the Trusted Core Framework **MUST** consider that the cancellation of the corresponding operation is requested.
- When the Cancellation Flag is set for a given task, the function `TEE_GetCancellationFlag` **MUST** 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

```
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 **MUST** 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

```
bool TEE_UnmaskCancellation( void );
```

#### Description

The `TEE_UnmaskCancellation` function unmask 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

```
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

```
TEE_Result TEE_CheckMemoryAccessRights(  
    uint32_t accessFlags,  
    void*    buffer, uint32_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.

In these 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 **MUST** guarantee the following properties:

- For the flag `TEE_MEMORY_ACCESS_READ` and `TEE_MEMORY_ACCESS_WRITE`, the Implementation **MUST** 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 **MUST** 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 **MUST** 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 **MUST** return the data.
  - **Non-observability:** If the Trusted Application writes some data in the buffer, then the data **MUST 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` **MUST** 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 **MUST** guarantee that some types of memory blocks have a minimum set of access rights:

- The following blocks **MUST** allow read and write accesses, **MUST** 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 **MUST 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 **MUST** perform a copy into a safe TA-owned memory buffer before passing the structures to the entry points.
- The following blocks **MUST** allow read accesses, **MUST** 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, **MUST** be owned by the Trusted Application instance, and **MUST** 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, **MUST** 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 **MUST** also guarantee that the `NULL` pointer cannot be dereferenced. If a Trusted Application attempts to read one byte at the address `NULL`, it **MUST** panic. This guarantee **MUST** extend to a segment of addresses starting at `NULL`, but the size of this segment is implementation-dependent.

### Parameters

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

This function **MUST NOT** panic for any reason.

### 4.11.2 TEE\_SetInstanceData

```
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 MUST 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

```
[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 TEE\_Malloc

```
void* TEE_Malloc(
    uint32_t size,
    uint32_t hint );
```

##### Description

The `TEE_Malloc` 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 of the basic C types.

The parameter `hint` is a hint to the allocator. The valid values for the hint are defined in Table 4-17. 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.

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

Name	Hint Value	Meaning
TEE_MALLOC_FILL_ZERO	0x00000000	Guarantees that the returned block of memory is filled with zeroes
Reserved	0x00000001- 0x7FFFFFFF	Reserved for future versions of this specification.
Implementation defined	0x80000000- 0xFFFFFFFF	Reserved for implementation-defined hints.

The hint **MUST** be attached to the allocated block and **SHOULD** 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.

##### 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 **MUST 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.

### 4.11.5 TEE\_Realloc

```
void* TEE_Realloc(  
    [in] void*    buffer,  
    uint32_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 MUST 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.

### 4.11.6 TEE\_Free

```
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`, or 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

```
void TEE_MemMove(  
    [outbuf(size)] void* dest,  
    [inbuf(size)] void* src,  
    uint32_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.

### 4.11.8 TEE\_MemCompare

```
int32_t TEE_MemCompare(  
    [inbuf(size)] void*    buffer1,  
    [inbuf(size)] void*    buffer2,  
    uint32_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.

### 4.11.9 TEE\_MemFill

```
void TEE_MemFill(  
    [outbuf(size)] void*    buffer,  
    uint32_t x,  
    uint32_t size);
```

#### Description

The TEE\_MemFill function writes the byte `x` (converted to a `uint8_t`) 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.



## 5 Trusted Storage API for Data and Keys

This chapter includes the following sections:

5.1	Summary of Features and Design .....	89
5.2	Trusted Storage and Rollback Detection .....	91
5.3	Data Types .....	92
5.4	Constants .....	94
5.5	Generic Object Functions.....	96
5.6	Transient Object Functions .....	102
5.7	Persistent Object Functions .....	117
5.8	Persistent Object Enumeration Functions.....	126
5.9	Data Stream Access Functions.....	131

### 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**.
  - The current version of this specification defines a single Trusted Storage Space for each TA, which is its own private storage space. The objects in this storage space are accessible only to the TA that created them and are not visible to other TAs.
  - 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**. Data objects have only a data stream. 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 amount 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 **MUST** 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 MUST 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 trust that a Trusted Application can put on 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`.

Typically, an implementation may rely on the REE for that purpose (protection level 100) or on hardware assets controlled by the TEE (protection level 1000).

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

Property Value	Meaning
100	Rollback detection mechanism for the Trusted Storage is enforced at the REE level.
1000	Rollback detection mechanism for the Trusted Storage is based on TEE-controlled hardware. This hardware MUST be out of reach of software attacks from the REE. Users may still be able to roll back the Trusted Storage but this MUST be detected by the Implementation.

## 5.3 Data Types

### 5.3.1 TEE\_Attribute

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

```
typedef struct {
    uint32_t attributeID;
    union
    {
        struct
        {
            [inbuf] void* buffer; uint32_t length;
        } ref;
        struct
        {
            uint32_t a, 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` MUST be selected. If the bit is set to 1, then it is a value attribute and the field `value` MUST 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.

### 5.3.2 TEE\_ObjectInfo

```
typedef struct {
    uint32_t objectType;
    uint32_t keySize;
    uint32_t maxKeySize;
    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

```
typedef enum
{
    TEE_DATA_SEEK_SET = 0,
    TEE_DATA_SEEK_CUR = 1,
    TEE_DATA_SEEK_END = 2
} TEE_Whence;
```

This structure enumerates the possible start offset when moving a data position in the data stream associated with a persistent object.

### 5.3.4 TEE\_ObjectHandle

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

TEE\_ObjectHandle is an opaque handle on a cryptographic object. TEE\_ObjectHandle is an opaque handle on a cryptographic 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

```
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

**Table 5-2: Object Storage Constants**

Constant Name	Value
Reserved	0x00000000
TEE_STORAGE_PRIVATE	0x00000001
Reserved for future use	0x00000002-0x7FFFFFFF
Reserved for implementation defined storage	0x80000000-0xFFFFFFFF

**Table 5-3: Data Flag Constants**

Constant Name	Value
TEE_DATA_FLAG_ACCESS_READ	0x00000001
TEE_DATA_FLAG_ACCESS_WRITE	0x00000002
TEE_DATA_FLAG_ACCESS_WRITE_META	0x00000004
TEE_DATA_FLAG_SHARE_READ	0x00000010
TEE_DATA_FLAG_SHARE_WRITE	0x00000020
TEE_DATA_FLAG_OVERWRITE	0x00000400
TEE_DATA_FLAG_EXCLUSIVE (deprecated, replace with TEE_DATA_FLAG_OVERWRITE)	0x00000400

**Table 5-4: Usage Constants**

Constant Name	Value
TEE_USAGE_EXTRACTABLE	0x00000001
TEE_USAGE_ENCRYPT	0x00000002
TEE_USAGE_DECRYPT	0x00000004
TEE_USAGE_MAC	0x00000008
TEE_USAGE_SIGN	0x00000010
TEE_USAGE_VERIFY	0x00000020
TEE_USAGE_DERIVE	0x00000040

**Table 5-5: Handle Flag Constants**

Constant Name	Value
TEE_HANDLE_FLAG_PERSISTENT	0x00010000
TEE_HANDLE_FLAG_INITIALIZED	0x00020000
TEE_HANDLE_FLAG_KEY_SET	0x00040000
TEE_HANDLE_FLAG_EXPECT_TWO_KEYS	0x00080000

**Table 5-6: Operation Constants**

Constant Name	Value
TEE_OPERATION_CIPHER	1
TEE_OPERATION_MAC	3
TEE_OPERATION_AE	4
TEE_OPERATION_DIGEST	5
TEE_OPERATION_ASYMMETRIC_CIPHER	6
TEE_OPERATION_ASYMMETRIC_SIGNATURE	7
TEE_OPERATION_KEY_DERIVATION	8
Reserved for future use	0x00000009-0x7FFFFFFF
Implementation defined	0x80000000-0xFFFFFFFF

**Table 5-7: Operation States**

Constant Name	Value
TEE_OPERATION_STATE_INITIAL	0x00000000
TEE_OPERATION_STATE_ACTIVE	0x00000001
Reserved for future use	0x00000002-0x7FFFFFFF
Implementation defined	0x80000000-0xFFFFFFFF

**Table 5-8: Miscellaneous Constants**

Constant Name	Value
TEE_DATA_MAX_POSITION	0xFFFFFFFF
TEE_OBJECT_ID_MAX_LEN	64

## 5.5 Generic Object Functions

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

### 5.5.1 TEE\_GetObjectInfo1

```
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
- keySize: The current size in bits of the object as determined by its attributes. This will always be less than or equal to maxKeySize. Set to 0 for uninitialized and data only objects.
- maxKeySize: The maximum keySize which this object can represent.
  - 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.
- 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**

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

## 5.5.2 TEE\_RestrictObjectUsage1

```
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 **MUST** 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

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

### 5.5.3 TEE\_GetObjectBufferAttribute

```

TEE_Result TEE_GetObjectBufferAttribute(
    TEE_ObjectHandle object,
    uint32_t attributeID,
    [outbuf] void* buffer, uint32_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 6.11.

Bit [29] of the attribute identifier **MUST** be set to 0, i.e. it **MUST** denote a buffer 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 6.11 for a definition of all available object attributes, their formats, and their level of protection.

#### 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

- `object` is not a valid opened object handle.
- The object is not initialized.
- Bit [29] of `attributeID` is not set to 0, so the attribute is not a buffer attribute.
- 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 which is not explicitly associated with a defined return code for this function.

### 5.5.4 TEE\_GetObjectValueAttribute

```

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 6.11.

Bit [29] of the attribute identifier MUST be set to 1, i.e. it MUST 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 6.11 for a definition of all available object attributes and their level of protection.

#### 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

- `object` is not a valid opened object handle.
- The object is not initialized.
- Bit [29] of `attributeID` is not set to 1, so the attribute is not a value attribute.
- 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 which is not explicitly associated with a defined return code for this function.

### 5.5.5 TEE\_CloseObject

```
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

- `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

```
TEE_Result TEE_AllocateTransientObject(  
    uint32_t      objectType,  
    uint32_t      maxKeySize,  
    [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 and the maximum key size MUST be specified so that all the container resources can be pre-allocated.

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 the one of the appropriate key sizes described in Table 5-9.

Note that a compliant Implementation MUST 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 5-9: TEE\_AllocateTransientObject Object Types and Key Sizes**

Object Type	Possible Key Sizes
TEE_TYPE_AES	128, 192, or 256 bits
TEE_TYPE_DES	Always 64 bits including the parity bits. This gives an effective key size of 56 bits
TEE_TYPE_DES3	128 or 192 bits including the parity bits. This gives effective key sizes of 112 or 168 bits
TEE_TYPE_HMAC_MD5	Between 64 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA1	Between 80 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA224	Between 112 and 512 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA256	Between 192 and 1024 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA384	Between 256 and 1024 bits, multiple of 8 bits
TEE_TYPE_HMAC_SHA512	Between 256 and 1024 bits, multiple of 8 bits
TEE_TYPE_RSA_PUBLIC_KEY	The number of bits in the modulus. 256, 512, 768, 1024, 1536 and 2048 bit keys MUST be supported. Support for other key sizes including bigger key sizes is implementation-dependent. Minimum key size is 256 bits. <sup>3</sup>
TEE_TYPE_RSA_KEYPAIR	Same as for RSA public key size.
TEE_TYPE_DSA_PUBLIC_KEY	Depends on Algorithm: ALG_DSA_SHA1: Between 512 and 1024 bits, multiple of 64 bits ALG_DSA_SHA224: 2048 bits 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
TEE_TYPE_ECDSA_PUBLIC_KEY	Conditional: If ECC is supported, then all the curve sizes defined in Table 6-14 MUST be supported.
TEE_TYPE_ECDSA_KEYPAIR	Conditional: If ECC is supported, then MUST be same value as for ECDSA public key size.
TEE_TYPE_ECDH_PUBLIC_KEY	Conditional: If ECC is supported, then all the curve sizes defined in Table 6-14 MUST be supported.
TEE_TYPE_ECDH_KEYPAIR	Conditional: If ECC is supported, then MUST be same value as for ECH public key size.
TEE_TYPE_GENERIC_SECRET	Multiple of 8 bits, up to 4096 bits. This type is intended for secret data that is not directly used as a key in a cryptographic operation, but participates in a key derivation.
TEE_TYPE_DATA	0 – All data is in the associated data stream.

<sup>3</sup> Note that using RSA keys smaller than 1024 bits is nowadays considered insecure but may be required for legacy protocols.

**Parameters**

- `objectType`: Type of uninitialized object container to be created (see Table 6-13).
- `maxKeySize`: 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 which is not explicitly associated with a defined return code for this function.



## 5.6.2 TEE\_FreeTransientObject

```
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 **MUST** 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

- `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

```
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 `0xFFFFFFFF`.

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

- `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

```
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 **MUST** be uninitialized. If the object is initialized, the caller **MUST** 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 parts 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 5-10: TEE\_PopulateTransientObject Supported Attributes**

Object Type	Attributes
TEE_TYPE_AES	For all secret key objects, the <code>TEE_ATTR_SECRET_VALUE</code> <b>MUST</b> be provided.  For <code>TEE_TYPE_DES</code> and <code>TEE_TYPE_DES3</code> , the buffer associated with this attribute <b>MUST</b> include parity bits.
TEE_TYPE_DES	
TEE_TYPE_DES3	
TEE_TYPE_HMAC_MD5	
TEE_TYPE_HMAC_SHA1	
TEE_TYPE_HMAC_SHA224	
TEE_TYPE_HMAC_SHA256	
TEE_TYPE_HMAC_SHA384	
TEE_TYPE_HMAC_SHA512	
TEE_TYPE_GENERIC_SECRET	
TEE_TYPE_RSA_PUBLIC_KEY	The following parts <b>MUST</b> be provided: <code>TEE_ATTR_RSA_MODULUS</code> <code>TEE_ATTR_RSA_PUBLIC_EXPONENT</code>

Object Type	Attributes
TEE_TYPE_RSA_KEYPAIR	<p>The following parts MUST be provided:</p> <p>TEE_ATTR_RSA_MODULUS</p> <p>TEE_ATTR_RSA_PUBLIC_EXPONENT</p> <p>TEE_ATTR_RSA_PRIVATE_EXPONENT</p> <p>The CRT parameters are optional. If any of these parts is provided, then all of them MUST be provided:</p> <p>TEE_ATTR_RSA_PRIME1</p> <p>TEE_ATTR_RSA_PRIME2</p> <p>TEE_ATTR_RSA_EXPONENT1</p> <p>TEE_ATTR_RSA_EXPONENT2</p> <p>TEE_ATTR_RSA_COEFFICIENT</p>
TEE_TYPE_ECDSA_PUBLIC_KEY	<p>Conditional: If ECC is supported, then the following parts MUST be provided:</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_Y</p> <p>TEE_ATTR_ECC_CURVE</p>
TEE_TYPE_ECDSA_KEYPAIR	<p>Conditional: If ECC is supported, then the following parts MUST be provided:</p> <p>TEE_ATTR_ECC_PRIVATE_VALUE</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_Y</p> <p>TEE_ATTR_ECC_CURVE</p>
TEE_TYPE_ECDH_PUBLIC_KEY	<p>Conditional: If ECC is supported, then the following parts MUST be provided:</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_Y</p> <p>TEE_ATTR_ECC_CURVE</p>
TEE_TYPE_ECDH_KEYPAIR	<p>Conditional: If ECC is supported, then the following parts MUST be provided:</p> <p>TEE_ATTR_ECC_PRIVATE_VALUE</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_Y</p> <p>TEE_ATTR_ECC_CURVE</p>
TEE_TYPE_DSA_PUBLIC_KEY	<p>The following parts MUST be provided:</p> <p>TEE_ATTR_DSA_PRIME</p> <p>TEE_ATTR_DSA_SUBPRIME</p> <p>TEE_ATTR_DSA_BASE</p> <p>TEE_ATTR_DSA_PUBLIC_VALUE</p>

Object Type	Attributes
TEE_TYPE_DSA_KEYPAIR	The following parts MUST be provided: TEE_ATTR_DSA_PRIME TEE_ATTR_DSA_SUBPRIME TEE_ATTR_DSA_BASE TEE_ATTR_DSA_PRIVATE_VALUE TEE_ATTR_DSA_PUBLIC_VALUE
TEE_TYPE_DH_KEYPAIR	The following parts MUST be provided: TEE_ATTR_DH_PRIME TEE_ATTR_DH_BASE TEE_ATTR_DH_PUBLIC_VALUE TEE_ATTR_DH_PRIVATE_VALUE Optionally, TEE_ATTR_DH_SUBPRIME may be provided, too.

All mandatory attributes MUST 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 MUST return the error code TEE\_ERROR\_BAD\_PARAMETERS and MUST NOT panic. If it does not detect any inconsistencies, it MUST 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 MUST 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 MUST be initialized.
- **TEE\_ERROR\_BAD\_PARAMETERS**: If an incorrect or inconsistent attribute value is detected. In this case, the content of the object MUST remain uninitialized.

### Panic Reasons

- **object** is not a valid opened object handle that is transient and uninitialized.
- Some mandatory attribute is missing.
- An attribute which is not defined for the object's type is present in **attrs**
- 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 which is not explicitly associated with a defined return code for this function.

### 5.6.5 TEE\_InitRefAttribute, TEE\_InitValueAttribute

```
void TEE_InitRefAttribute(
    [out] TEE_Attribute* attr,
          uint32_t      attributeID,
    [inbuf] void*       buffer, uint32_t length );
```

```
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:

```
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 6.11
- `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

- 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.

## 5.6.6 TEE\_CopyObjectAttributes1

```
TEE_Result TEE_CopyObjectAttributes1(
    TEE_ObjectHandle destObject,
    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` MUST refer to an uninitialized object handle and MUST therefore be a transient object.

The source and destination objects MUST have compatible types and sizes in the following sense:

- The type of `destObject` MUST be a subtype of `srcObject`, i.e. one of the conditions listed in Table 5-11 MUST be true.

**Table 5-11: TEE\_CopyObjectAttributes1 Parameter Types**

Type of <code>srcObject</code>	Type of <code>destObject</code>
Any	Equal to type of <code>srcObject</code>
TEE_TYPE_RSA_KEYPAIR	TEE_TYPE_RSA_PUBLIC_KEY
TEE_TYPE_DSA_KEYPAIR	TEE_TYPE_DSA_PUBLIC_KEY
TEE_TYPE_ECDSA_KEYPAIR (optional)	TEE_TYPE_ECDSA_PUBLIC_KEY (optional)
TEE_TYPE_ECDH_KEYPAIR (optional)	TEE_TYPE_ECDH_PUBLIC_KEY (optional)

- The size of `srcObject` MUST 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:** 0x802

### 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**

- `srcObject` is not initialized.
- `destObject` is initialized.
- The type and size of `srcObject` and `destObject` are not compatible.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

### 5.6.7 TEE\_GenerateKey

```
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 MUST 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**

Object Type	Details
TEE_TYPE_AES	No parameter is necessary. The function generates the attribute TEE_ATTR_SECRET_VALUE. The generated value SHALL be the full key size.
TEE_TYPE_DES	
TEE_TYPE_DES3	
TEE_TYPE_HMAC_MD5	
TEE_TYPE_HMAC_SHA1	
TEE_TYPE_HMAC_SHA224	
TEE_TYPE_HMAC_SHA256	
TEE_TYPE_HMAC_SHA384	
TEE_TYPE_HMAC_SHA512	
TEE_TYPE_GENERIC_SECRET	
TEE_TYPE_RSA_KEYPAIR	<p>No parameter is required.</p> <p>The TEE_ATTR_RSA_PUBLIC_EXPONENT attribute may be specified; if omitted, the default value is 65537.</p> <p>Key generation SHALL follow the rules defined in [NIST SP800-56B].</p> <p>The function generates and populates the following attributes:</p> <ul style="list-style-type: none"> <li>TEE_ATTR_RSA_MODULUS</li> <li>TEE_ATTR_RSA_PUBLIC_EXPONENT (if not specified)</li> <li>TEE_ATTR_RSA_PRIVATE_EXPONENT</li> <li>TEE_ATTR_RSA_PRIME1</li> <li>TEE_ATTR_RSA_PRIME2</li> <li>TEE_ATTR_RSA_EXPONENT1</li> <li>TEE_ATTR_RSA_EXPONENT2</li> <li>TEE_ATTR_RSA_COEFFICIENT</li> </ul>

Object Type	Details
TEE_TYPE_DSA_KEYPAIR	<p>The following domain parameter MUST be passed to the function:</p> <p>TEE_ATTR_DSA_PRIME TEE_ATTR_DSA_SUBPRIME TEE_ATTR_DSA_BASE</p> <p>The function generates and populates the following attributes:</p> <p>TEE_ATTR_DSA_PUBLIC_VALUE TEE_ATTR_DSA_PRIVATE_VALUE</p>
TEE_TYPE_DH_KEYPAIR	<p>The following domain parameters MUST be passed to the function:</p> <p>TEE_ATTR_DH_PRIME TEE_ATTR_DH_BASE</p> <p>The following parameters can optionally be passed:</p> <p>TEE_ATTR_DH_SUBPRIME (<math>q</math>): If present, constrains the private value <math>x</math> to be in the range <math>[2, q-2]</math></p> <p>TEE_ATTR_DH_X_BITS (<math>\ell</math>): If present, constrains the private value <math>x</math> to have <math>\ell</math> bits</p> <p>If neither of these optional parts is specified, then the only constraint on <math>x</math> is that it is less than <math>p-1</math>.</p> <p>The function generates and populates the following attributes:</p> <p>TEE_ATTR_DH_PUBLIC_VALUE TEE_ATTR_DH_PRIVATE_VALUE TEE_ATTR_DH_X_BITS (number of bits in <math>x</math>)</p>
TEE_TYPE_ECDSA_KEYPAIR	<p>The following domain parameters MUST be passed to the function:</p> <p>TEE_ATTR_ECC_CURVE</p> <p>The function generates and populates the following attributes:</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X TEE_ATTR_ECC_PUBLIC_VALUE_Y TEE_ATTR_ECC_PRIVATE_VALUE</p>
TEE_TYPE_ECDH_KEYPAIR	<p>The following domain parameters MUST be passed to the function:</p> <p>TEE_ATTR_ECC_CURVE</p> <p>The function generates and populates the following attributes:</p> <p>TEE_ATTR_ECC_PUBLIC_VALUE_X TEE_ATTR_ECC_PUBLIC_VALUE_Y TEE_ATTR_ECC_PRIVATE_VALUE</p>

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. MUST be less than or equal to the maximum key size specified when the object container was created. MUST 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.

**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

- `object` is not a valid opened object handle that is transient and uninitialized.
- `keySize` is not supported or is too large.
- A mandatory parameter is missing.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 5.7 Persistent Object Functions

### 5.7.1 TEE\_OpenPersistentObject

```

TEE_Result TEE_OpenPersistentObject(
    [in(objectIDLength)] uint32_t      storageID,
    void*            objectID, uint32_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 **MUST** 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 MUST 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 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.

## Panic Reasons

- `objectIDLen` is greater than `TEE_OBJECT_ID_MAX_LEN`.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 5.7.2 TEE\_CreatePersistentObject

```

TEE_Result TEE_CreatePersistentObject(
    uint32_t          storageID,
    [in(objectIDLength)] void*      objectID, uint32_t objectIDLen,
    uint32_t          flags,
    TEE_ObjectHandle  attributes,
    [inbuf]           void*      initialData, uint32_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 `attributes` argument can also be `NULL` for a pure data object. The object type, size, and usage are copied from `attributes`. 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 **MUST** 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**

TEE_DATA_FLAG_OVERWRITE in flags	Object Exists	Object Created?	Return Code
Absent	No	Yes	TEE_SUCCESS
Absent	Yes	No	TEE_ERROR_ACCESS_CONFLICT
Present	No	Yes	TEE_SUCCESS
Present	Yes	Deleted and re-created as an atomic operation	TEE_SUCCESS

## 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 **MUST** 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 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.

## Panic Reasons

- **objectIDLen** is greater than `TEE_OBJECT_ID_MAX_LEN`.
- **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.



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

### 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 **MUST** 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 **MUST** 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 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 section 5.4 are reserved for future use and **MUST** 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.

**Table 5-14: TEE\_OpenPersistentObject Sharing Rules**

Value of flags for First Open/Create	Value of flags for Second Open/Create	Return Code of Second Open/Create	Comments
ACCESS_READ	ACCESS_READ	ACCESS_CONFLICT	The object handles have not been opened with the flag SHARE_READ. Only the first call will succeed.
ACCESS_READ   SHARE_READ	ACCESS_READ	ACCESS_CONFLICT	Not all the object handles have been opened with the flag SHARE_READ. Only the first call will succeed.
ACCESS_READ   SHARE_READ	ACCESS_READ   SHARE_READ	TEE_SUCCESS	All the object handles have been opened with the flag SHARE_READ.
ACCESS_READ	ACCESS_WRITE	ACCESS_CONFLICT	Objects are not opened with share flags. Only the first call will succeed.
ACCESS_READ   SHARE_READ   SHARE_WRITE	ACCESS_WRITE   SHARE_READ   SHARE_WRITE	TEE_SUCCESS	All the object handles have been opened with the share flags.
ACCESS_READ   SHARE_READ   ACCESS_WRITE   SHARE_WRITE	ACCESS_WRITE_META	ACCESS_CONFLICT	The write-meta flag indicates an exclusive access to the object. Only the first call will succeed.
SHARE_READ	ACCESS_WRITE   SHARE_WRITE	ACCESS_CONFLICT	An object can be opened with only share flags, which locks the access to an object against a given mode. Here the first call prevents subsequent accesses in write mode.
0	ACCESS_READ   SHARE_READ	ACCESS_CONFLICT	An object can be opened with no flag set, which completely locks all subsequent attempts to access the object. Only the first call will succeed.

### 5.7.4 TEE\_CloseAndDeletePersistentObject1

```
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 **MUST** 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

- `object` is not a valid handle on a persistent object opened with the write-meta access right.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 5.7.5 TEE\_RenamePersistentObject

```
TEE_Result TEE_RenamePersistentObject(  
    TEE_ObjectHandle object,  
    [in(newObjectIDLen)] void* newObjectID, uint32_t newObjectIDLen );
```

### Description

The function `TEE_RenamePersistentObject` changes the identifier of an object. The object handle MUST 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 MUST 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

- `object` is not a valid handle on a persistent object that has been opened with the write-meta access right.
- `newObjectID` resides in shared memory.
- `newObjectIDLen` is more than `TEE_OBJECT_ID_MAX_LEN`.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 5.8 Persistent Object Enumeration Functions

### 5.8.1 TEE\_AllocatePersistentObjectEnumerator

```
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 which is not explicitly associated with a defined return code for this function.

## 5.8.2 TEE\_FreePersistentObjectEnumerator

```
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

- `objectEnumerator` is not a valid handle on an object enumerator.
- If the Implementation detects any other error.

### 5.8.3 TEE\_ResetPersistentObjectEnumerator

```
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

- `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

```
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 MUST 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

- `objectEnumerator` is not a valid handle on an object enumerator.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

### 5.8.5 TEE\_GetNextPersistentObject

```

TEE_Result TEE_GetNextPersistentObject(
    [out] TEE_ObjectEnumHandle    objectEnumerator,
    [out] TEE_ObjectInfo*         objectInfo,
    [out(TEE_OBJECT_ID_MAX_LEN)] void*    objectID,
    [out] uint32_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 exit 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

- `objectEnumerator` is not a valid handle on an object enumerator.
- `objectID` is `NULL`.
- `objectIDLen` is `NULL`.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 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

```
TEE_Result TEE_ReadObjectData(  
    TEE_ObjectHandle object,  
    [out] void*        buffer,  
    uint32_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 **MUST** 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

- `object` is not a valid handle on a persistent object opened with the read access right.

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

## 5.9.2 TEE\_WriteObjectData

```
TEE_Result TEE_WriteObjectData(  
    TEE_ObjectHandle object,  
    [in] void*        buffer, uint32_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 **MUST** 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

- `object` is not a valid handle on a persistent object opened with the write access right.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

### 5.9.3 TEE\_TruncateObjectData

```
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 **MUST** 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

- `object` is not a valid handle on a persistent object opened with the write access right.
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 5.9.4 TEE\_SeekObjectData

```
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

- `object` is not a valid handle on a persistent object.
- If the Implementation detects any other error 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**

Algorithm Type	Supported Algorithm
Digests	MD5 SHA-1 SHA-256 SHA-224 SHA-384 SHA-512
Symmetric ciphers	DES Triple-DES with double-length and triple-length keys AES
Message Authentication Codes (MACs)	DES-MAC AES-MAC AES-CMAC HMAC with one of the supported digests
Authenticated Encryption (AE)	AES-CCM with support for Additional Authenticated Data (AAD) AES-GCM with support for Additional Authenticated Data (AAD)
Asymmetric Encryption Schemes	RSA PKCS1-V1.5 RSA OAEP
Asymmetric Signature Schemes	DSA RSA PKCS1-V1.5 RSA PSS
Key Exchange Algorithms	Diffie-Hellman

- In addition, the algorithms in Table 6-2 are supported if the `tee.cryptography.ecc` property is set to `True`.



**Table 6-2: ECC Cryptographic Algorithms**

Algorithm Type	Supported Algorithm
Asymmetric Signature Schemes	ECDSA
Key Exchange Algorithms	ECDH

- 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

The enumeration `TEE_OperationMode` lists the modes for all the cryptographic operations.

```
typedef enum {
    TEE_MODE_ENCRYPT    = 0,
    TEE_MODE_DECRYPT    = 1,
    TEE_MODE_SIGN       = 2,
    TEE_MODE_VERIFY     = 3,
    TEE_MODE_MAC        = 4,
    TEE_MODE_DIGEST     = 5,
    TEE_MODE_DERIVE     = 6
} TEE_OperationMode;
```

**Table 6-3: Possible TEE\_OperationMode Values**

Name	Comment
TEE_MODE_ENCRYPT	Encryption mode
TEE_MODE_DECRYPT	Decryption mode
TEE_MODE_SIGN	Signature generation mode
TEE_MODE_VERIFY	Signature verification mode
TEE_MODE_MAC	MAC mode
TEE_MODE_DIGEST	Digest mode
TEE_MODE_DERIVE	Key derivation mode

### 6.1.2 TEE\_OperationInfo

```
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

```
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

$$\text{sizeof(TEE\_OperationInfoMultiple)} + N * \text{sizeof(TEE\_OperationInfoKey)}$$

### 6.1.4 TEE\_OperationHandle

```
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

```
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 **MUST** 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` **MUST** be one of the constants defined in section 6.10.1.

The parameter `mode` **MUST** be one of the constants defined in section 6.1.1. It **MUST** be compatible with the algorithm as defined by Table 6-4.

The parameter `maxKeySize` **MUST** be a valid value as defined in Table 5-9 for the algorithm.

The operation is placed in **initial** state.

**Table 6-4: TEE\_AllocateOperation Allowed Modes**

Algorithm	Possible Modes
TEE_ALG_AES_ECB_NOPAD	TEE_MODE_ENCRYPT TEE_MODE_DECRYPT
TEE_ALG_AES_CBC_NOPAD	
TEE_ALG_AES_CTR	
TEE_ALG_AES_CTS	
TEE_ALG_AES_XTS	
TEE_ALG_AES_CCM	
TEE_ALG_AES_GCM	
TEE_ALG_DES_ECB_NOPAD	
TEE_ALG_DES_CBC_NOPAD	
TEE_ALG_DES3_ECB_NOPAD	
TEE_ALG_DES3_CBC_NOPAD	
TEE_ALG_DES_CBC_MAC_NOPAD	TEE_MODE_MAC
TEE_ALG_AES_CBC_MAC_NOPAD	
TEE_ALG_AES_CBC_MAC_PKCS5	
TEE_ALG_AES_CMAC	
TEE_ALG_DES_CBC_MAC_PKCS5	
TEE_ALG_DES3_CBC_MAC_NOPAD	
TEE_ALG_DES3_CBC_MAC_PKCS5	
TEE_ALG_RSASSA_PKCS1_V1_5_MD5	TEE_MODE_SIGN TEE_MODE_VERIFY
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 (if supported)	

Algorithm	Possible Modes
TEE_ALG_RSAES_PKCS1_V1_5	TEE_MODE_ENCRYPT TEE_MODE_DECRYPT
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_DH_DERIVE_SHARED_SECRET	TEE_MODE_DERIVE
TEE_ALG_ECDH_DERIVE_SHARED_SECRET (if supported)	
TEE_ALG_MD5	TEE_MODE_DIGEST
TEE_ALG_SHA1	
TEE_ALG_SHA224	
TEE_ALG_SHA256	
TEE_ALG_SHA384	
TEE_ALG_SHA512	
TEE_ALG_HMAC_MD5	TEE_MODE_MAC
TEE_ALG_HMAC_SHA1	
TEE_ALG_HMAC_SHA224	
TEE_ALG_HMAC_SHA256	
TEE_ALG_HMAC_SHA384	
TEE_ALG_HMAC_SHA512	

Note that all algorithms listed in Table 6-4 MUST 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 enumerated 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 which is not explicitly associated with a defined return code for this function.

## 6.2.2 TEE\_FreeOperation

```
void TEE_FreeOperation( TEE_OperationHandle operation );
```

### Description

The TEE\_Free Operation 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.

### Parameters

- operation: Reference to operation handle

**Specification Number:** 10    **Function Number:** 0xC03

### Panic Reasons

- operation is not a valid handle on an operation.
- If the Implementation detects any other error.



### 6.2.3 TEE\_GetOperationInfo

```
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 6.1.2):

- `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_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`: 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.

#### Parameters

- `operation`: Handle on the operation
- `operationInfo`: Pointer to a structure filled with the operation information

**Specification Number:** 10    **Function Number:** 0xC04

#### Panic Reasons

- `operation` is not a valid opened operation handle.
- If the Implementation detects any other error.

## 6.2.4 TEE\_GetOperationInfoMultiple

```
TEE_Result TEE_GetOperationInfoMultiple(
    TEE_OperationHandle operation,
    [outbuf] TEE_OperationInfoMultiple* operationInfoMultiple, uint32_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_AllocateOperation`
- `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:

$(*operationSize - \text{sizeof}(\text{TEE\_OperationInfoMultiple})) / \text{sizeof}(\text{TEE\_OperationInfoKey}) + 1$

**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

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

## 6.2.5 TEE\_ResetOperation

```
void TEE_ResetOperation( TEE_OperationHandle operation );
```

### Description

For a multi-stage operation, the `TEE_ResetOperation` function resets the operation to **initial** state before initialization, but after the key has been set.

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

- `operation` is not a valid opened operation handle.
- The key has not been set yet.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

## 6.2.6 TEE\_SetOperationKey

```
TEE_Result TEE_SetOperationKey(
    TEE_OperationHandle operation,
    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 **MUST** be in **initial** state before the operation and remains in **initial** state afterwards.

The key object type and size **MUST** be compatible with the type and size of the operation. The operation mode **MUST** be compatible with key usage:

- In general, the operation mode **MUST** 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 **MUST** contain both the `TEE_USAGE_ENCRYPT` and `TEE_USAGE_VERIFY` flags.
  - For `TEE_MODE_DECRYPT`, the object usage **MUST** 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 6-5: Public Key Allowed Modes**

Key Type	Allowed Operation Modes
<code>TEE_TYPE_RSA_PUBLIC_KEY</code>	<code>TEE_MODE_VERIFY</code> or <code>TEE_MODE_ENCRYPT</code>
<code>TEE_TYPE_DSA_PUBLIC_KEY</code>	<code>TEE_MODE_VERIFY</code>
<code>TEE_TYPE_ECDSA_PUBLIC_KEY</code> (optional)	<code>TEE_MODE_VERIFY</code>
<code>TEE_TYPE_ECDH_PUBLIC_KEY</code> (optional)	<code>TEE_MODE_DERIVE</code>

- 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 6-6: Key-Pair Parts for Operation Modes**

Operation Mode	Key Parts Used
TEE_MODE_VERIFY	Public
TEE_MODE_SIGN	Private
TEE_MODE_ENCRYPT	Public
TEE_MODE_DECRYPT	Private
TEE_MODE_DERIVE	Public and Private

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

- `operation` is not a valid opened operation handle.
- `key` is not `TEE_HANDLE_NULL` and is not a valid handle on a key object.
- `key` is not initialized.
- The operation expects no key (digest mode) or two keys (AES-XTS algorithm) .
- The type, size, or usage of `key` is not compatible with the algorithm, mode, or size of the operation.
- `operation` is not in **initial** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.2.7 TEE\_SetOperationKey2

```
TEE_Result TEE_SetOperationKey2(  
    TEE_OperationHandle operation,  
    TEE_ObjectHandle    key1,  
    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`.

This function works like `TEE_SetOperationKey` except that two keys are set instead of a single key.

`key1` and `key2` MUST both be non-NULL or both NULL.

### 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.

### Panic Reasons

- `operation` is not a valid opened operation handle.
- `key1` and `key2` are not both `TEE_HANDLE_NULL` and `key1` or `key2` or both are not valid handles on a key object.
- `key1` and/or `key2` are not initialized.
- The operation expects no key (digest mode) or a single key (all but AES-XTS algorithm) .
- The type, size, or usage of `key1` or `key2` is not compatible with the algorithm, mode, or size of the operation.
- `operation` is not in **initial** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.2.8 TEE\_CopyOperation

```
void TEE_CopyOperation(  
    TEE_OperationHandle dstOperation,  
    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` **MUST** 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` **MUST** 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

- `dstOperation` or `srcOperation` is not a valid opened operation handle.
- The algorithm or mode differ in `dstOperation` and `srcOperation`.
- `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.



## 6.3 Message Digest Functions

### 6.3.1 TEE\_DigestUpdate

```
void TEE_DigestUpdate(  
    TEE_OperationHandle operation,  
    [inbuf] void* chunk, uint32_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

- `operation` is not a valid operation handle of class `TEE_OPERATION_DIGEST`.
- Input data exceeds maximum length for algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

### 6.3.2 TEE\_DigestDoFinal

```
TEE_Result TEE_DigestDoFinal(  
    TEE_OperationHandle operation,  
    [inbuf] void* chunk, uint32_t chunkLen,  
    [outbuf] void* hash, uint32_t *hashLen );
```

#### 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_DIGEST`.
- Input data exceeds maximum length for algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 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

```
void TEE_CipherInit(  
    TEE_OperationHandle operation,  
    [inbuf] void* IV, uint32_t IVLen );
```

#### Description

The `TEE_CipherInit` function starts the symmetric cipher operation.

The operation **MUST** 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 or the initial counter value as appropriate

**Specification Number:** 10    **Function Number:** 0xE02

#### Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- No key is programmed in the `operation`.
- The Initialization Vector does not have the length required by the algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

## 6.4.2 TEE\_CipherUpdate

```
TEE_Result TEE_CipherUpdate(  
    TEE_OperationHandle operation,  
    [inbuf] void*          srcData, uint32_t srcLen,  
    [outbuf] void*          destData, uint32_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 operation **MUST** 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- The operation has not been started yet with `TEE_CipherInit` or has already been finalized.
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

### 6.4.3 TEE\_CipherDoFinal

```
TEE_Result TEE_CipherDoFinal(  
    TEE_OperationHandle operation,  
    [inbuf] void* srcData, uint32_t srcLen,  
    [outbufopt] void* destData, uint32_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 operation **MUST** 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_CIPHER`.
- The operation has not been started yet with `TEE_CipherInit` or has already been finalized.
- 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.
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 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 **MUST** use the functions defined in section 6.6.

### 6.5.1 TEE\_MACInit

```
void TEE_MACInit(  
    TEE_OperationHandle operation,  
    [inbuf] void* IV, uint32_t IVLen );
```

#### Description

The `TEE_MACInit` function initializes a MAC operation.

The operation **MUST** 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- No key is programmed in the `operation`.
- The Initialization Vector does not have the length required by the algorithm.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

## 6.5.2 TEE\_MACUpdate

```
void TEE_MACUpdate(  
    TEE_OperationHandle operation,  
    [inbuf] void* chunk, uint32_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 **MUST** 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.
- Input data exceeds maximum length for algorithm.
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

### 6.5.3 TEE\_MACComputeFinal

```
TEE_Result TEE_MACComputeFinal(  
    TEE_OperationHandle operation,  
    [inbuf] void* message, uint32_t messageLen,  
    [outbuf] void* mac, uint32_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 MUST 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_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to contain the computed MAC

#### Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.
- Input data exceeds maximum length for algorithm.
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.



## 6.5.4 TEE\_MACCompareFinal

```
TEE_Result TEE_MACCompareFinal(  
    TEE_OperationHandle operation,  
    [inbuf] void* message, uint32_t messageLen,  
    [inbuf] void* mac, uint32_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 **MUST** 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_MAC`.
- The operation has not been started yet with `TEE_MACInit` or has already been finalized.
- Input data exceeds maximum length for algorithm.
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 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

```
TEE_Result TEE_AEInit(  
    TEE_OperationHandle operation,  
    [inbuf] void*         nonce, uint32_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

- operation is not a valid operation handle of class TEE\_OPERATION\_AE.
- No key is programmed in the operation.
- The nonce length is not compatible with the length required by the algorithm.
- operation is not in **initial** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.6.2 TEE\_AEUpdateAAD

```
void TEE_AEUpdateAAD(  
    TEE_OperationHandle operation,  
    [inbuf] void* AADdata, uint32_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 operation may be in either **initial** or **active** state and enters **active** 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

- operation is not a valid operation handle of class TEE\_OPERATION\_AE.
- The operation has not started yet.
- The AAD length has already been reached (AES-CCM only).
- operation is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

### 6.6.3 TEE\_AEUpdate

```
TEE_Result TEE_AEUpdate(  
    TEE_OperationHandle operation,  
    [inbuf] void*          srcData, uint32_t srcLen,  
    [outbuf] void*          destData, uint32_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_AEFinal` routine.

The operation may be in either **initial** or **active** state and enters **active** state afterwards.

#### 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- The operation has not started yet.
- The required AAD length has not been provided yet (AES-CCM only).
- The payload length has already been reached (AES-CCM only).
- `operation` is not in **active** state.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.6.4 TEE\_AEEncryptFinal

```

TEE_Result TEE_AEEncryptFinal(
    TEE_OperationHandle operation,
    [inbuf] void* srcData, uint32_t srcLen,
    [outbuf] void* destData, uint32_t* destLen,
    [outbuf] void* tag, uint32_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 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_AE`.
- The operation has not started yet.
- The required AAD and payload have not been provided.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.6.5 TEE\_AEDecryptFinal

```

TEE_Result TEE_AEDecryptFinal(
    TEE_OperationHandle operation,
    [inbuf] void* srcData, uint32_t srcLen,
    [outbuf] void* destData, uint32_t *destLen,
    [in] void* tag, uint32_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 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

- operation is not a valid operation handle of class TEE\_OPERATION\_AE.
- The operation has not started yet.
- The required AAD and payload have not been provided.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 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

```
TEE_Result TEE_AsymmetricEncrypt(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params,      uint32_t paramCount
    [inbuf] void* srcData,          uint32_t srcLen,
    [outbuf] void* destData,        uint32_t *destLen );

TEE_Result TEE_AsymmetricDecrypt(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params,      uint32_t paramCount
    [inbuf] void* srcData,          uint32_t srcLen,
    [outbuf] void* destData,        uint32_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

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-7.

**Table 6-7: Asymmetric Encrypt/Decrypt Operation Parameters**

Algorithm	Possible Operation Parameters
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_XXX	TEE_ATTR_RSA_OAEP_LABEL: This parameter is optional. If not present, an empty label is assumed.

## Parameters

- operation: Handle on the operation, which MUST 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.

## Panic Reasons

- operation is not a valid operation handle of class TEE\_OPERATION\_ASYMMETRIC\_CIPHER.
- No key is programmed in the operation.
- The mode is not compatible with the function.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.



## 6.7.2 TEE\_AsymmetricSignDigest

```
TEE_Result TEE_AsymmetricSignDigest(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params,      uint32_t paramCount,
    [inbuf] void* digest,          uint32_t digestLen,
    [outbuf] void* signature,      uint32_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.

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_P192` (if supported)
- `TEE_ALG_ECDSA_P224` (if supported)
- `TEE_ALG_ECDSA_P256` (if supported)
- `TEE_ALG_ECDSA_P384` (if supported)
- `TEE_ALG_ECDSA_P521` (if supported)

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-8.

**Table 6-8: Asymmetric Sign Operation Parameters**

Algorithm	Possible Operation Parameters
<code>TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX</code>	<code>TEE_ATTR_RSA_PSS_SALT_LENGTH</code> : Number of bytes in the salt. This parameter is optional. If not present, the salt length is equal to the hash length.

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 MUST 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_ASYMMETRIC_SIGNATURE`.
- No key is programmed in the operation.
- The operation mode is not `TEE_MODE_SIGN`.
- `digestLen` is not equal to the hash size of the algorithm
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

### 6.7.3 TEE\_AsymmetricVerifyDigest

```
TEE_Result TEE_AsymmetricVerifyDigest(
    TEE_OperationHandle operation,
    [in] TEE_Attribute* params,      uint32_t paramCount,
    [inbuf] void* digest,          uint32_t digestLen,
    [inbuf] void* signature, uint32_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\_P192 (if supported)
- TEE\_ALG\_ECDSA\_P224 (if supported)
- TEE\_ALG\_ECDSA\_P256 (if supported)
- TEE\_ALG\_ECDSA\_P384 (if supported)
- TEE\_ALG\_ECDSA\_P521 (if supported)

The parameters `params`, `paramCount` contain the operation parameters listed in Table 6-9.

**Table 6-9: Asymmetric Verify Operation Parameters**

Algorithm	Possible Operation Parameters
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_XXX	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.

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 MUST 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

- `operation` is not a valid operation handle of class `TEE_OPERATION_ASYMMETRIC_SIGNATURE`.
- No key is programmed in the operation.
- The operation mode is not `TEE_MODE_VERIFY`.
- `digestLen` is not equal to the hash size of the algorithm
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error which is not explicitly associated with a defined return code for this function.

## 6.8 Key Derivation Functions

### 6.8.1 TEE\_DeriveKey

```
void TEE_DeriveKey(
    TEE_OperationHandle operation,
    [in] TEE_Attribute*    params, uint32_t paramCount
    TEE_ObjectHandle     derivedKey );
```

#### Description

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**

Algorithm	Possible Operation Parameters
TEE_ALG_DH_DERIVE_SHARED_SECRET	TEE_ATTR_DH_PUBLIC_VALUE: Public key part of the other party. This parameter is mandatory.
TEE_ALG_ECDH_NIST_P192_DERIVE_SHARED_SECRET TEE_ALG_ECDH_NIST_P224_DERIVE_SHARED_SECRET TEE_ALG_ECDH_NIST_P256_DERIVE_SHARED_SECRET TEE_ALG_ECDH_NIST_P384_DERIVE_SHARED_SECRET TEE_ALG_ECDH_NIST_P521_DERIVE_SHARED_SECRET (if supported)	TEE_ATTR_ECC_PUBLIC_VALUE_X, TEE_ATTR_ECC_PUBLIC_VALUE_Y: Public key part of the other party. This parameter is mandatory.

The `derivedKey` handle MUST refer to an object with type `TEE_TYPE_GENERIC_SECRET`.

On completion the derived key is placed into the `TEE_ATTR_SECRET_VALUE` attribute of the `derivedKey` handle.

#### Parameters

- `operation`: Handle on the operation, which MUST have been suitably set up with an operation key
- `params`, `paramCount`: Operation parameters
- `derivedKey`: Handle on an uninitialized transient object filled with the derived key

**Specification Number:** 10    **Function Number:** 0x1201

#### Panic Reasons

- `operation` is not a valid operation handle of class `TEE_OPERATION_KEY_DERIVATION`.
- No key is programmed in the operation.
- A mandatory parameter is missing.
- The operation mode is not `TEE_MODE_DERIVE`.
- Hardware or cryptographic algorithm failure
- If the Implementation detects any other error.

## 6.9 Random Data Generation Function

### 6.9.1 TEE\_GenerateRandom

```
void TEE_GenerateRandom(  
    [out] void*      randomBuffer,  
    uint32_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.

## 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. Note that the algorithm identifiers have the structure defined in Table 6-12. (Normative references for the algorithms may be found in Annex C.)

Implementations MAY define their own algorithms. Such algorithms MUST have implementation-defined algorithm identifiers, and those identifiers MUST use the structure defined in Table 6-12.

**Table 6-11: List of Algorithm Identifiers**

Name	Identifier	Comments
TEE_ALG_AES_ECB_NOPAD	0x10000010	
TEE_ALG_AES_CBC_NOPAD	0x100000110	
TEE_ALG_AES_CTR	0x100000210	The counter MUST be encoded as a 16-byte buffer in big-endian form. Between two consecutive blocks, the counter MUST be incremented by 1. If it reaches the limit of all 128 bits set to 1, it MUST wrap around to 0.
TEE_ALG_AES_CTS	0x100000310	
TEE_ALG_AES_XTS	0x100000410	
TEE_ALG_AES_CBC_MAC_NOPAD	0x300000110	
TEE_ALG_AES_CBC_MAC_PKCS5	0x300000510	
TEE_ALG_AES_CMAC	0x300000610	
TEE_ALG_AES_CCM	0x400000710	
TEE_ALG_AES_GCM	0x400000810	
TEE_ALG_DES_ECB_NOPAD	0x100000011	
TEE_ALG_DES_CBC_NOPAD	0x100000111	
TEE_ALG_DES_CBC_MAC_NOPAD	0x300000111	
TEE_ALG_DES_CBC_MAC_PKCS5	0x300000511	
TEE_ALG_DES3_ECB_NOPAD	0x100000013	Triple DES MUST be understood as Encrypt-Decrypt-Encrypt mode with two or three keys.
TEE_ALG_DES3_CBC_NOPAD	0x100000113	
TEE_ALG_DES3_CBC_MAC_NOPAD	0x300000113	
TEE_ALG_DES3_CBC_MAC_PKCS5	0x300000513	

Name	Identifier	Comments
TEE_ALG_RSASSA_PKCS1_V1_5_MD5	0x70001830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA1	0x70002830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA224	0x70003830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA256	0x70004830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA384	0x70005830	
TEE_ALG_RSASSA_PKCS1_V1_5_SHA512	0x70006830	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1	0x70212930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224	0x70313930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256	0x70414930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384	0x70515930	
TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512	0x70616930	
TEE_ALG_RSAES_PKCS1_V1_5	0x60000130	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1	0x60210230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224	0x60310230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256	0x60410230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384	0x60510230	
TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512	0x60610230	
TEE_ALG_RSA_NOPAD	0x60000030	
TEE_ALG_DSA_SHA1	0x70002131	
TEE_ALG_DSA_SHA224	0x70003131	
TEE_ALG_DSA_SHA256	0x70004131	
TEE_ALG_DH_DERIVE_SHARED_SECRET	0x80000032	
TEE_ALG_MD5	0x50000001	
TEE_ALG_SHA1	0x50000002	
TEE_ALG_SHA224	0x50000003	
TEE_ALG_SHA256	0x50000004	
TEE_ALG_SHA384	0x50000005	
TEE_ALG_SHA512	0x50000006	
TEE_ALG_HMAC_MD5	0x30000001	
TEE_ALG_HMAC_SHA1	0x30000002	
TEE_ALG_HMAC_SHA224	0x30000003	
TEE_ALG_HMAC_SHA256	0x30000004	
TEE_ALG_HMAC_SHA384	0x30000005	
TEE_ALG_HMAC_SHA512	0x30000006	



Name	Identifier	Comments
TEE_ALG_ECDSA_P192	0x70001042	If supported
TEE_ALG_ECDSA_P224	0x70002042	If supported
TEE_ALG_ECDSA_P256	0x70003042	If supported
TEE_ALG_ECDSA_P384	0x70004042	If supported
TEE_ALG_ECDSA_P521	0x70005042	If supported
TEE_ALG_ECDH_P192	0x80001042	If supported
TEE_ALG_ECDH_P224	0x80002042	If supported
TEE_ALG_ECDH_P256	0x80003042	If supported
TEE_ALG_ECDH_P384	0x80004042	If supported
TEE_ALG_ECDH_P521	0x80005042	If supported

**Table 6-12: Structure of Algorithm Identifier**

Bits	Function	Values
Bits [31:28]	Specify the algorithm class and determine which function can be called	0x1: Block cipher 0x3: MAC 0x4: Authenticated Encryption cipher 0x5: Digest 0x6: Asymmetric cipher 0x7: Asymmetric signature 0x8: Key derivation 0xA: Object handle
Bits [7:0]	Identify the underlying main algorithm itself	0x00: Generic Secret Key object 0x01: MD5 0x02: SHA-1 0x03: SHA-224 0x04: SHA-256 0x05: SHA-384 0x06: SHA-512 0x10: AES 0x11: DES 0x12: DES2 (only for key generation) 0x13: DES3 0x30: RSA 0x31: DSA 0x32: DH 0x41: ECDSA (if supported) 0x42: ECDH (if supported) 0xBE: Object is invalid 0xBF: No appropriate algorithm Algorithm numbers above 0xC0 are reserved for implementation-specific algorithms.
Bits [11:8]	Define the chaining mode or padding	
Bits [15:12]	Define the message digest for asymmetric signature algorithms	
Bits [19:16]	Define the MGF for RSA PSS and RSA OAEP algorithms or Define the elliptic curve used in ECC methods if supported	

Bits	Function	Values
Bits [23:20]	Define the internal hash used by the MGF for RSA OAEP (for signature algorithms, equal to the message digest)	
Bits [27:24]	Not used	

## 6.10.2 Object Types

Object handles are a special class of algorithm handle and follow the rules in Table 6-12 but only use the object type and algorithm fields.

**Table 6-13: List of Object Types**

Name	Identifier
TEE_TYPE_AES	0xA0000010
TEE_TYPE_DES	0xA0000011
TEE_TYPE_DES3	0xA0000013
TEE_TYPE_HMAC_MD5	0xA0000001
TEE_TYPE_HMAC_SHA1	0xA0000002
TEE_TYPE_HMAC_SHA224	0xA0000003
TEE_TYPE_HMAC_SHA256	0xA0000004
TEE_TYPE_HMAC_SHA384	0xA0000005
TEE_TYPE_HMAC_SHA512	0xA0000006
TEE_TYPE_RSA_PUBLIC_KEY	0xA0000030
TEE_TYPE_RSA_KEYPAIR	0xA1000030
TEE_TYPE_DSA_PUBLIC_KEY	0xA0000031
TEE_TYPE_DSA_KEYPAIR	0xA1000031
TEE_TYPE_DH_KEYPAIR	0xA1000032
TEE_TYPE_ECDSA_PUBLIC_KEY	0xA0000041
TEE_TYPE_ECDSA_KEYPAIR	0xA1000041
TEE_TYPE_ECDH_PUBLIC_KEY	0xA0000042
TEE_TYPE_ECDH_KEYPAIR	0xA1000042
TEE_TYPE_GENERIC_SECRET	0xA0000000
TEE_TYPE_CORRUPTED_OBJECT	0xA00000BE
TEE_TYPE_DATA	0xA00000BF

Object types using implementation-specific algorithms are defined by the implementation.

The 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.

The TEE\_TYPE\_DATA is used to represent objects which have no cryptographic attributes, just a data stream.

### 6.10.3 Elliptic Curve Types

If elliptic curve cryptography (ECC) is supported then the curve to be used is defined as follows. All curve definitions come from <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf> [NIST Re Cur].

**Table 6-14: List of Supported ECC Curves**

Name	Identifier
TEE_ECC_CURVE_NIST_P192	0x00000001
TEE_ECC_CURVE_NIST_P224	0x00000002
TEE_ECC_CURVE_NIST_P256	0x00000003
TEE_ECC_CURVE_NIST_P384	0x00000004
TEE_ECC_CURVE_NIST_P521	0x00000005
Reserved for future use	0x00000006-0x7FFFFFFF
Implementation defined	0x80000000-0xFFFFFFFF

## 6.11 Object or Operation Attributes

Table 6-15: Object or Operation Attributes

Name	Value	Protection	Type	Format (Table 6-16)	Comment
TEE_ATTR_SECRET_VALUE	0xC0000000	Protected	Ref	binary	Used for all secret keys for symmetric ciphers, MACs, and HMACs
TEE_ATTR_RSA_MODULUS	0xD0000130	Public	Ref	bignum	
TEE_ATTR_RSA_PUBLIC_EXPONENT	0xD0000230	Public	Ref	bignum	
TEE_ATTR_RSA_PRIVATE_EXPONENT	0xC0000330	Protected	Ref	bignum	
TEE_ATTR_RSA_PRIME1	0xC0000430	Protected	Ref	bignum	Usually referred to as $p$ .
TEE_ATTR_RSA_PRIME2	0xC0000530	Protected	Ref	bignum	$q$
TEE_ATTR_RSA_EXPONENT1	0xC0000630	Protected	Ref	bignum	$dp$
TEE_ATTR_RSA_EXPONENT2	0xC0000730	Protected	Ref	bignum	$dq$
TEE_ATTR_RSA_COEFFICIENT	0xC0000830	Protected	Ref	bignum	$iq$
TEE_ATTR_DSA_PRIME	0xD0001031	Public	Ref	bignum	$p$
TEE_ATTR_DSA_SUBPRIME	0xD0001131	Public	Ref	bignum	$q$
TEE_ATTR_DSA_BASE	0xD0001231	Public	Ref	bignum	$g$
TEE_ATTR_DSA_PUBLIC_VALUE	0xD0000131	Public	Ref	bignum	$y$
TEE_ATTR_DSA_PRIVATE_VALUE	0xC0000231	Protected	Ref	bignum	$x$
TEE_ATTR_DH_PRIME	0xD0001032	Public	Ref	bignum	$p$
TEE_ATTR_DH_SUBPRIME	0xD0001132	Public	Ref	bignum	$q$
TEE_ATTR_DH_BASE	0xD0001232	Public	Ref	bignum	$g$
TEE_ATTR_DH_X_BITS	0xF0001332	Public	Value	int	$\ell$
TEE_ATTR_DH_PUBLIC_VALUE	0xD0000132	Public	Ref	bignum	$y$
TEE_ATTR_DH_PRIVATE_VALUE	0xC0000232	Protected	Ref	bignum	$x$
TEE_ATTR_RSA_OAEP_LABEL	0xD0000930	Public	Ref	binary	
TEE_ATTR_RSA_PSS_SALT_LENGTH	0xF0000A30	Public	Value	int	
TEE_ATTR_ECC_PUBLIC_VALUE_X	0xD0000141	Public	Ref	bignum	
TEE_ATTR_ECC_PUBLIC_VALUE_Y	0xD0000241	Public	Ref	bignum	
TEE_ATTR_ECC_PRIVATE_VALUE	0xC0000341	Protected	Ref	bignum	$d$
TEE_ATTR_ECC_CURVE	0xF0000441	Public	Value	int	Value from Table 6-14

**Table 6-16: Attribute Format Definitions**

Format	Description
binary	An array of unsigned octets
bignum	An unsigned bignum in big-endian binary format. Leading zero bytes are allowed.
int	Values attributes represented in a single integer returned/read from argument a.

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**

Bit	Function	Values
Bit [29]	Defines whether the attribute is a buffer or value attribute	0: buffer attribute 1: value attribute
Bit [28]	Defines whether the attribute is protected or public	0: protected attribute 1: public attribute

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**

Name	Value
<code>TEE_ATTR_FLAG_VALUE</code>	<code>0x20000000</code>
<code>TEE_ATTR_FLAG_PUBLIC</code>	<code>0x10000000</code>

## 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 **MUST 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 **MUST** 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 **MUST** 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`: one field for the seconds and one field for the milliseconds within this second.

### 7.1 Data Types

#### 7.1.1 TEE\_Time

```
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

```
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 **MUST** 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 7-1: Values of the `gpd.tee.systemTime.protectionLevel` Property**

Value	Meaning
100	System time based on REE-controlled timers. Can be tampered by the REE. The implementation <b>MUST</b> still guarantee that the system time is monotonic, i.e. successive calls to <code>TEE_GetSystemTime</code> <b>MUST</b> return increasing values of the system time.
1000	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> .

#### 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

```
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 **MUST** 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 which is not explicitly associated with a defined return code for this function.

### 7.2.3 TEE\_GetTAPersistentTime

```
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.

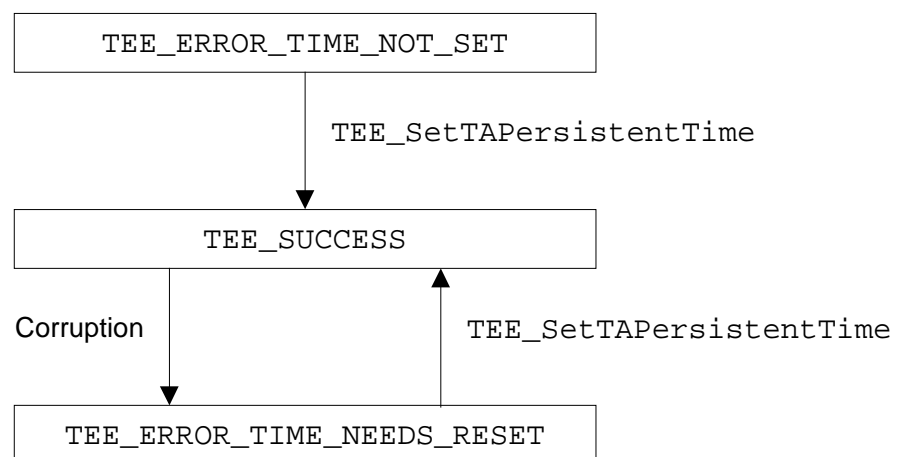
This function can return the following statuses (as well as other status values discussed in “Return Code”):

- 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 MUST 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 MUST 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. Until the persistent time has been reset, the status TEE\_ERROR\_TIME\_NEEDS\_RESET will always be returned.

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**



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.

**Table 7-2: Values of the `gpd.tee.TAPersistentTime.protectionLevel` Property**

Value	Meaning
100	Persistent time based on an REE-controlled real-time clock and on the TEE Trusted Storage for the storage of origins. The implementation <b>MUST</b> guarantee that rollback of persistent time is detected to the fullest extent allowed by the Trusted Storage.
1000	Persistent time based on a TEE-controlled real-time clock and the TEE Trusted Storage. The real-time clock <b>MUST</b> 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 <b>MUST</b> be detected by the Implementation.

The number of seconds in the TA Persistent Time may exceed the range of the `uint32_t` integer type. In this case, the function **MUST** 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` **MUST** 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 which is not explicitly associated with a defined return code for this function.

## 7.2.4 TEE\_SetTAPersistentTime

```
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 which is not explicitly associated with a defined return code for this function.

## 7.2.5 TEE\_GetREETime

```
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. Allowing the possibility of expanding the Cryptographic Operations API means that some of its functions can be left as optional to implement.

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 **MUST** 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 **MUST** ensure that  $M$  is at least 2048. The exact value of  $M$  can be retrieved as the implementation property `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 **MUST** 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 **MUST** 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.

```
typedef uint32_t TEE_BigInt;
```

The following constraints are put on the internal representation of the `TEE_BigInt`:

- 1) The size of the representation **MUST** be a multiple of 4 bytes.
- 2) The extra memory within the representation to store metadata **MUST NOT** exceed 8 bytes.
- 3) The representation **MUST** be stored 32-bit aligned in memory.

Exactly how a multi-precision integer is represented internally is implementation-specific but it **MUST** 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:

```
uint32_t      twoints[2 * TEE_BigIntSizeInU32(1024)];
TEE_BigInt*   first  = twoints;
TEE_BigInt*   second = twoints + TEE_BigIntSizeInU32(1024);

/* Or if we do it dynamically */
TEE_BigInt* op1;
op1 = TEE_Malloc(TEE_BigIntSizeInU32(1024) * sizeof(TEE_BigInt), 0);
/* 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.

### 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 **MUST** 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.

```
typedef uint32_t TEE_BigIntFMMContext;
```

The following constraints are put on the internal representation of the `TEE_BigIntFMMContext`:

- 1) The size of the representation **MUST** be a multiple of 4 bytes.
- 2) The representation **MUST** be stored 32-bit aligned in memory.

Exactly how this context is represented internally is implementation-specific but it **MUST** 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.

```
typedef uint32_t TEE_BigIntFMM;
```

The following constraints are put on the internal representation of the `TEE_BigIntFMM`:

- 1) The size of the representation **MUST** be a multiple of 4 bytes.
- 2) The representation **MUST** be stored 32-bit aligned in memory.

Exactly how this transformed representation is stored internally is implementation-specific but it **MUST** 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 MUST 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 MUST 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

```
#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

```
uint32_t TEE_BigIntFMMContextSizeInU32( uint32_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 MUST 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.

### 8.4.3 TEE\_BigIntFMMSizeInU32

```
uint32_t TEE_BigIntFMMSizeInU32( uint32_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 MUST 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.

## 8.5 Initialization Functions

These functions initialize the arithmetical objects after the TA has allocated the memory to store them. The Trusted Application MUST call the corresponding initialization function after it has allocated the memory for the arithmetical object.

### 8.5.1 TEE\_BigIntInit

```
void TEE_BigIntInit(  
    [out] TEE_BigInt *bigInt,  
    uint32_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:

```
TEE_BigInt *a;  
uint32_t len;  
len = TEE_BigIntSizeInU32(bitSize);  
a =(TEE_BigInt *)TEE_Malloc(len * sizeof(TEE_BigInt), 0);  
ret = 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.

## 8.5.2 TEE\_BigIntInitFMMContext

```
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:

```
TEE_BigIntFMMContext* ctx;  
uint32_t len = TEE_BigIntFMMContextSizeInU32(bitsize);  
ctx=(TEE_BigIntFMMContext *)TEE_Malloc(len * sizeof(TEE_BigIntFMMContext), 0);  
/*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 **MUST** 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.

### 8.5.3 TEE\_BigIntInitFMM

```
void TEE_BigIntInitFMM(  
    [in] TEE_BigIntFMM *bigIntFMM,  
    uint32_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:

```
TEE_BigIntFMM *a;  
uint32_t len;  
len = TEE_BigIntFMMSizeInU32(modulusSizeinBits);  
a = (TEE_BigIntFMM *)TEE_Malloc(len * sizeof(TEE_BigIntFMM), 0);  
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.

## 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

```
TEE_Result TEE_BigIntConvertFromOctetString(  
    [out]    TEE_BigInt *dest,  
    [inbuf]  uint8_t    *buffer, uint32_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 which is not explicitly associated with a defined return code for this function.

## 8.6.2 TEE\_BigIntConvertToOctetString

```
TEE_Result TEE_BigIntConvertToOctetString(  
    [outbuf] void*      buffer, uint32_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 which is not explicitly associated with a defined return code for this function.

### 8.6.3 TEE\_BigIntConvertFromS32

```
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 a `TEE_BigInt` to store the result
- `shortVal`: Input value

**Specification Number:** 10    **Function Number:** 0x1702

#### Result Size

The result MUST have memory allocation for holding a 32-bit signed value.

#### Panic Reasons

- If the Implementation detects any error.

### 8.6.4 TEE\_BigIntConvertToS32

```
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 which is not explicitly associated with a defined return code for this function.



## 8.7 Logical Operations

### 8.7.1 TEE\_BigIntCmp

```
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

```
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

```
void TEE_BigIntShiftRight(
    [out] TEE_BigInt *dest,
    [in]  TEE_BigInt *op
        uint32_t      bits );
```

#### Description

The `TEE_BigIntShiftRight` function computes  $|dest| = |op| \gg bits$  and `dest` will have the same sign as `op`.<sup>4</sup> 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.

#### 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.

### 8.7.4 TEE\_BigIntGetBit

```
bool TEE_BigIntGetBit(
    [in] TEE_BigInt *src,
        uint32_t     bitIndex );
```

#### Description

The `TEE_BigIntGetBit` function returns the `bitIndex`th 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 `bitIndex`th 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 `bitIndex`th bit in `|src|`. True represents a “1” and false represents a “0”.

#### Panic Reasons

- If the Implementation detects any error.

<sup>4</sup> The notation  $|x|$  means the absolute value of  $x$ .

### 8.7.5 TEE\_BigIntGetBitCount

```
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.

## 8.8 Basic Arithmetic Operations

This section describes basic arithmetical operations addition, subtraction, negation, multiplication, squaring, and division.

### 8.8.1 TEE\_BigIntAdd

```
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.

#### 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* MUST have memory allocation for holding  $\max(\text{magnitude}(op1), \text{magnitude}(op2)) + 1$  bits.<sup>5</sup>

#### Panic Reasons

- If the Implementation detects any error.

---

<sup>5</sup> The magnitude function is defined in section 8.7.5.

## 8.8.2 TEE\_BigIntSub

```
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.

### 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 MUST have memory allocation for holding  $\max(\text{magnitude}(op1), \text{magnitude}(op2)) + 1$  bits.

### Panic Reasons

- If the Implementation detects any error.

### 8.8.3 TEE\_BigIntNeg

```
void TEE_BigIntNeg(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op );
```

#### Description

The TEE\_BigIntNeg function negates an operand:  $dest = -op$ . `dest` and `op` MAY point to the same memory region.

#### 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 MUST have memory allocation for `magnitude(op)` bits.

#### Panic Reasons

- If the Implementation detects any error.

## 8.8.4 TEE\_BigIntMul

```
void TEE_BigIntMul(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op1,  
    [in]  TEE_BigInt *op2 );
```

### Description

The TEE\_BigIntMul function computes  $dest = op1 * op2$ . All or some of *dest*, *op1*, and *op2* MAY point to the same memory region.

### 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:** 0x1903

### Result Size

The result MUST have memory allocation for  $(\text{magnitude}(op1) + \text{magnitude}(op2))$  bits.

### Panic Reasons

- If the Implementation detects any error.

### 8.8.5 TEE\_BigIntSquare

```
void TEE_BigIntSquare(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op );
```

#### Description

The TEE\_BigIntSquare function computes  $dest = op * op$ . `dest` and `op` MAY point to the same memory region.

#### Parameters

- `dest`: Pointer to TEE\_BigInt to store the result  $op * op$
- `op`: Pointer to the operand to be squared

**Specification Number:** 10    **Function Number:** 0x1905

#### Result Size

The result MUST have memory allocation for  $2 * \text{magnitude}(op)$  bits.

#### Panic Reasons

- If the Implementation detects any error.



### 8.8.6 TEE\_BigIntDiv

```
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:

op1	op2	dest_q	dest_r	Expression
53	7	7	4	$53 = 7*7 + 4$
-53	7	-7	-4	$-53 = (-7)*7 + (-4)$
53	-7	-7	+4	$53 = (-7)*(-7) + 4$
-53	-7	7	-4	$-53 = 7*(-7) + (-4)$

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` MUST 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`, MUST have memory allocation for 0 bytes if  $|op1| \leq |op2|$  and  $magnitude(op1) - magnitude(op2)$  if  $|op1| > |op2|$ .

The remainder `dest_r` MUST have memory allocation to hold  $magnitude(op2)$  bits.

#### Panic Reasons

- `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

```
void TEE_BigIntMod(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op,  
    [in]  TEE_BigInt *n );
```

#### Description

The `TEE_BigIntMod` function computes  $dest = op \pmod n$  such that  $0 \leq dest < n$ . `dest` and `op` MAY point to the same memory region but `n` MUST point to a unique memory region. For negative `op` the function follows the normal convention that  $-1 = (n-1) \pmod n$ .

#### Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result  $op \pmod n$ . 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 MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A03

#### Result Size

The result `dest` MUST have memory allocation for `magnitude(n)` bits.<sup>6</sup>

#### Panic Reasons

- `n < 2`
- If the Implementation detects any other error.

---

<sup>6</sup> The magnitude function is defined in section 8.7.5.

## 8.9.2 TEE\_BigIntAddMod

```
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  $dest = (op1 + op2) \pmod n$ . All or some of `dest`, `op1`, and `op2` MAY point to the same memory region but `n` MUST point to a unique memory region.

### Parameters

- `dest`: Pointer to `TEE_BigInt` to hold the result  $(op1 + op2) \pmod n$
- `op1`: Pointer to the first operand. Operand MUST be in the interval  $[0, n-1]$ .
- `op2`: Pointer to the second operand. Operand MUST be in the interval  $[0, n-1]$ .
- `n`: Pointer to the modulus. Modulus MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A01

### Result Size

The result `dest` MUST have memory allocation for `magnitude(n)` bits.

### Panic Reasons

- $n < 2$
- If the Implementation detects any other error.

### 8.9.3 TEE\_BigIntSubMod

```
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  $dest = (op1 - op2) \pmod n$ . All or some of dest, op1, and op2 MAY point to the same memory region but n MUST point to a unique memory region.

#### Parameters

- dest: Pointer to TEE\_BigInt to hold the result  $(op1 - op2) \pmod n$
- op1: Pointer to the first operand. Operand MUST be in the interval  $[0, n-1]$ .
- op2: Pointer to the second operand. Operand MUST be in the interval  $[0, n-1]$ .
- n: Pointer to the modulus. Modulus MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A06

#### Result Size

The result dest MUST have memory allocation for  $\text{magnitude}(n)$  bits.

#### Panic Reasons

- $n < 2$
- If the Implementation detects any other error.

### 8.9.4 TEE\_BigIntMulMod

```
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  $dest = (op1 * op2) \pmod n$ . All or some of dest, op1, and op2 MAY point to the same memory region but n MUST point to a unique memory region.

#### Parameters

- dest: Pointer to TEE\_BigInt to hold the result  $(op1 * op2) \pmod n$
- op1: Pointer to the first operand. Operand MUST be in the interval  $[0, n-1]$ .
- op2: Pointer to the second operand. Operand MUST be in the interval  $[0, n-1]$ .
- n: Pointer to the modulus. Modulus MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A04

#### Result Size

The result dest MUST have memory allocation for  $\text{magnitude}(n)$  bits.

#### Panic Reasons

- $n < 2$
- If the Implementation detects any other error.

### 8.9.5 TEE\_BigIntSquareMod

```
void TEE_BigIntSquareMod(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op,  
    [in]  TEE_BigInt *n );
```

#### Description

The TEE\_BigIntSquareMod function computes  $dest = (op * op) \pmod n$ . dest and op1 MAY point to the same memory region but n MUST point to a unique memory region.

#### Parameters

- dest: Pointer to TEE\_BigInt to hold the result  $(op * op) \pmod n$
- op: Pointer to the operand. Operand MUST be in the interval  $[0, n-1]$ .
- n: Pointer to the modulus. Modulus MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A05

#### Result Size

The result dest MUST have memory allocation for  $\text{magnitude}(n)$  bits.

#### Panic Reasons

- $n < 2$
- If the Implementation detects any other error.

### 8.9.6 TEE\_BigIntInvMod

```
void TEE_BigIntInvMod(  
    [out] TEE_BigInt *dest,  
    [in]  TEE_BigInt *op,  
    [in]  TEE_BigInt *n );
```

#### Description

The `TEE_BigIntInvMod` function computes `dest` such that  $dest * op = 1 \pmod n$ . `dest` and `op` MAY point to the same memory region. 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  $(op^{-1}) \pmod n$
- `op`: Pointer to the operand. Operand MUST be in the interval  $[1, n-1]$ .
- `n`: Pointer to the modulus. Modulus MUST be larger than 1.

**Specification Number:** 10    **Function Number:** 0x1A02

#### Result Size

The result `dest` MUST have memory allocation for `magnitude(n)` bits.

#### Panic Reasons

- $n < 2$
- $op = 0$
- If the Implementation detects any other error.

## 8.10 Other Arithmetic Operations

### 8.10.1 TEE\_BigIntRelativePrime

```
bool TEE_BigIntRelativePrime(  
    [in]    TEE_BigInt *op1,  
    [in]    TEE_BigInt *op2 );
```

#### Description

The TEE\_BigIntRelativePrime function determines whether  $\gcd(\text{op1}, \text{op2}) == 1$ . op1 and op2 MAY point to the same memory region.

#### Parameters

- op1: Pointer to the first operand
- op2: Pointer to the second operand

**Specification Number:** 10    **Function Number:** 0x1B03

#### Return Value

- true if  $\gcd(\text{op1}, \text{op2}) == 1$
- false otherwise
- if any error is detected by the Implementation.



### 8.10.2 TEE\_BigIntComputeExtendedGcd

```
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. Furthermore it computes the coefficients u and v such that  $u*op1+v*op2==gcd$ . op1 and op2 MAY point to the same memory region. 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 MUST be able to hold  $\max(\text{magnitude}(\text{op1}), \text{magnitude}(\text{op2}))$  bits.<sup>7</sup>
- The absolute value of u is in the range  $[0, |\text{op2}/\text{gcd}|-1]$ .<sup>8</sup>
- The absolute value of v is in the range  $[0, |\text{op1}/\text{gcd}|-1]$ .

#### Panic Reasons

- If the Implementation detects any error.

<sup>7</sup> The magnitude function is defined in section 8.7.5.

<sup>8</sup> The notation  $|x|$  means the absolute value of x.

### 8.10.3 TEE\_BigIntIsProbablePrime

```
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 MUST 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 MUST be at least 80.

The algorithm for performing the primality test is implementation-specific, but its correctness and efficiency MUST 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

```
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 MUST 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

```
void TEE_BigIntConvertFromFMM(  
    [out] TEE_BigInt      *dest,  
    [in]  TEE_BigIntFMM   *src,  
    [in]  TEE_BigInt      *n,  
    [in]  TEE_BigIntFMMContext *context );
```

#### Description

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

```
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` MUST 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.

#### 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.

## Annex A Panicked Function Identification

If this specification is used in conjunction with the TEE TA Debug Specification ([Debug]), then the specification number is 10 and the values listed in Table A-1 MUST be associated with the function declared.

**Table A-1: Function Identification Values**

Category	Function	Function Number in hexadecimal	Function Number in decimal
TA Interface	TA_CloseSessionEntryPoint	0x101	257
	TA_CreateEntryPoint	0x102	258
	TA_DestroyEntryPoint	0x103	259
	TA_InvokeCommandEntryPoint	0x104	260
	TA_OpenSessionEntryPoint	0x105	261
Property Access	TEE_AllocatePropertyEnumerator	0x201	513
	TEE_FreePropertyEnumerator	0x202	514
	TEE_GetNextProperty	0x203	515
	TEE_GetPropertyAsBinaryBlock	0x204	516
	TEE_GetPropertyAsBool	0x205	517
	TEE_GetPropertyAsIdentity	0x206	518
	TEE_GetPropertyAsString	0x207	519
	TEE_GetPropertyAsU32	0x208	520
	TEE_GetPropertyAsUUID	0x209	521
	TEE_GetPropertyName	0x20A	522
	TEE_ResetPropertyEnumerator	0x20B	523
	TEE_StartPropertyEnumerator	0x20C	524
Panic Function	TEE_Panic	0x301	769
Internal Client API	TEE_CloseTASession	0x401	1025
	TEE_InvokeTACommand	0x402	1026
	TEE_OpenTASession	0x403	1027
Cancellation	TEE_GetCancellationFlag	0x501	1281
	TEE_MaskCancellation	0x502	1282
	TEE_UnmaskCancellation	0x503	1283

Category	Function	Function Number in hexadecimal	Function Number in decimal
Memory Management	TEE_CheckMemoryAccessRights	0x601	1537
	TEE_Free	0x602	1538
	TEE_GetInstanceData	0x603	1539
	TEE_Malloc	0x604	1540
	TEE_MemCompare	0x605	1541
	TEE_MemFill	0x606	1542
	TEE_MemMove	0x607	1543
	TEE_Realloc	0x608	1544
	TEE_SetInstanceData	0x609	1545
Generic Object	TEE_CloseObject	0x701	1793
	TEE_GetObjectBufferAttribute	0x702	1794
	TEE_GetObjectInfo (deprecated)	0x703	1795
	TEE_GetObjectValueAttribute	0x704	1796
	TEE_RestrictObjectUsage (deprecated)	0x705	1797
	TEE_GetObjectInfo1	0x706	1798
	TEE_RestrictObjectUsage1	0x707	1799
Transient Object	TEE_AllocateTransientObject	0x801	2049
	TEE_CopyObjectAttributes (deprecated)	0x802	2050
	TEE_FreeTransientObject	0x803	2051
	TEE_GenerateKey	0x804	2052
	TEE_InitRefAttribute	0x805	2053
	TEE_InitValueAttribute	0x806	2054
	TEE_PopulateTransientObject	0x807	2055
	TEE_ResetTransientObject	0x808	2056
	TEE_CopyObjectAttributes1	0x809	2057
Persistent Object	TEE_CloseAndDeletePersistentObject (deprecated)	0x901	2305
	TEE_CreatePersistentObject	0x902	2306
	TEE_OpenPersistentObject	0x903	2307
	TEE_RenamePersistentObject	0x904	2308
	TEE_CloseAndDeletePersistentObject1	0x905	2309

Category	Function	Function Number in hexadecimal	Function Number in decimal
Persistent Object Enumeration	TEE_AllocatePersistentObjectEnumerator	0xA01	2561
	TEE_FreePersistentObjectEnumerator	0xA02	2562
	TEE_GetNextPersistentObject	0xA03	2563
	TEE_ResetPersistentObjectEnumerator	0xA04	2564
	TEE_StartPersistentObjectEnumerator	0xA05	2565
Data Stream Access	TEE_ReadObjectData	0xB01	2817
	TEE_SeekObjectData	0xB02	2818
	TEE_TruncateObjectData	0xB03	2819
	TEE_WriteObjectData	0xB04	2820
Generic Operation	TEE_AllocateOperation	0xC01	3073
	TEE_CopyOperation	0xC02	3074
	TEE_FreeOperation	0xC03	3075
	TEE_GetOperationInfo	0xC04	3076
	TEE_ResetOperation	0xC05	3077
	TEE_SetOperationKey	0xC06	3078
	TEE_SetOperationKey2	0xC07	3079
	TEE_GetOperationInfoMultiple	0xC08	3080
Message Digest	TEE_DigestDoFinal	0xD01	3329
	TEE_DigestUpdate	0xD02	3330
Symmetric Cipher	TEE_CipherDoFinal	0xE01	3585
	TEE_CipherInit	0xE02	3586
	TEE_CipherUpdate	0xE03	3587
MAC	TEE_MACCompareFinal	0xF01	3841
	TEE_MACComputeFinal	0xF02	3842
	TEE_MACInit	0xF03	3843
	TEE_MACUpdate	0xF04	3844
Authenticated Encryption	TEE_AEDecryptFinal	0x1001	4097
	TEE_AEEncryptFinal	0x1002	4098
	TEE_AEInit	0x1003	4099
	TEE_AEUpdate	0x1004	4100
	TEE_AEUpdateAAD	0x1005	4101



Category	Function	Function Number in hexadecimal	Function Number in decimal
Asymmetric	TEE_AsymmetricDecrypt	0x1101	4353
	TEE_AsymmetricEncrypt	0x1102	4354
	TEE_AsymmetricSignDigest	0x1103	4355
	TEE_AsymmetricVerifyDigest	0x1104	4356
Key Derivation	TEE_DeriveKey	0x1201	4609
Random Data Generation	TEE_GenerateRandom	0x1301	4865
Time	TEE_GetREETime	0x1401	5121
	TEE_GetSystemTime	0x1402	5122
	TEE_GetTAPersistentTime	0x1403	5123
	TEE_SetTAPersistentTime	0x1404	5124
	TEE_Wait	0x1405	5125
Memory Allocation and Size of Objects	TEE_BigIntFMMContextSizeInU32	0x1501	5377
	TEE_BigIntFMMSizeInU32	0x1502	5378
Initialization	TEE_BigIntInit	0x1601	5633
	TEE_BigIntInitFMM	0x1602	5634
	TEE_BigIntInitFMMContext	0x1603	5635
Converter	TEE_BigIntConvertFromOctetString	0x1701	5889
	TEE_BigIntConvertFromS32	0x1702	5890
	TEE_BigIntConvertToOctetString	0x1703	5891
	TEE_BigIntConvertToS32	0x1704	5892
Logical Operation	TEE_BigIntCmp	0x1801	6145
	TEE_BigIntCmpS32	0x1802	6146
	TEE_BigIntGetBit	0x1803	6147
	TEE_BigIntGetBitCount	0x1804	6148
	TEE_BigIntShiftRight	0x1805	6149
Basic Arithmetic	TEE_BigIntAdd	0x1901	6401
	TEE_BigIntDiv	0x1902	6402
	TEE_BigIntMul	0x1903	6403
	TEE_BigIntNeg	0x1904	6404
	TEE_BigIntSquare	0x1905	6405
	TEE_BigIntSub	0x1906	6406

Category	Function	Function Number in hexadecimal	Function Number in decimal
Modular Arithmetic	TEE_BigIntAddMod	0x1A01	6657
	TEE_BigIntInvMod	0x1A02	6658
	TEE_BigIntMod	0x1A03	6659
	TEE_BigIntMulMod	0x1A04	6660
	TEE_BigIntSquareMod	0x1A05	6661
	TEE_BigIntSubMod	0x1A06	6662
Other Arithmetic	TEE_BigIntComputeExtendedGcd	0x1B01	6913
	TEE_BigIntIsProbablePrime	0x1B02	6914
	TEE_BigIntRelativePrime	0x1B03	6915
Fast Modular Multiplication	TEE_BigIntComputeFMM	0x1C01	7169
	TEE_BigIntConvertFromFMM	0x1C02	7170
	TEE_BigIntConvertToFMM	0x1C03	7171

## Annex B      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.

### B.1.1      TEE\_GetObjectInfo – Deprecated

```
void TEE_GetObjectInfo(
    TEE_ObjectHandle    object,
    [out] TEE_ObjectInfo* objectInfo );
```

#### Description

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` and the rest of the fields are set to 0.
- `keySize`: Set to 0 for an uninitialized object
- `maxKeySize`
  - 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**

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

## B.1.2 TEE\_RestrictObjectUsage – Deprecated

```
void TEE_RestrictObjectUsage(  
    TEE_ObjectHandle object,  
    uint32_t          objectUsage );
```

### Description

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 MUST 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

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

### B.1.3 TEE\_CopyObjectAttributes – Deprecated

```
void TEE_CopyObjectAttributes(  
    TEE_ObjectHandle destObject,  
    TEE_ObjectHandle srcObject );
```

#### Description

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` MUST refer to an uninitialized object handle and MUST therefore be a transient object.

The source and destination objects MUST have compatible types and sizes in the following sense:

- The type of `destObject` MUST be a subtype of `srcObject`, i.e. one of the conditions listed in Table 5-11 MUST be true.
- The size of `srcObject` MUST 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

- `srcObject` is not initialized.
- `destObject` is initialized.
- The type and size of `srcObject` and `destObject` are not compatible.
- If the Implementation detects any other error.

## B.1.4 TEE\_CloseAndDeletePersistentObject - Deprecated

```
void TEE_CloseAndDeletePersistentObject( TEE_ObjectHandle object );
```

### Description

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 MUST 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

- 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.

## **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**

Name	References	URL
TEE_ALG_AES_ECB_NOPAD TEE_ALG_AES_CBC_NOPAD TEE_ALG_AES_CTR	FIPS 197 (AES) NIST SP800-38A (ECB, CBC, CTR)	<a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf">http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf</a>
TEE_ALG_AES_CTS	FIPS 197 (AES) NIST SP800-38A Addendum (CTS = CBC-CS3)	<a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-8A.pdf">http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-8A.pdf</a>
TEE_ALG_AES_XTS	IEEE Std 1619-2007	<a href="http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4493431">http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4493431</a>
TEE_ALG_AES_CCM	FIPS 197 (AES) RFC 3610 (CCM)	<a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://tools.ietf.org/html/rfc3610">http://tools.ietf.org/html/rfc3610</a>
TEE_ALG_AES_GCM	FIPS 197 (AES) NIST 800-38D (GCM)	<a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf">http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf</a>
TEE_ALG_DES_ECB_NOPAD TEE_ALG_DES_CBC_NOPAD TEE_ALG_DES3_ECB_NOPAD TEE_ALG_DES3_CBC_NOPAD	FIPS 46 (DES, 3DES) FIPS 81 (ECB, CBC)	<a href="http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf">http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf</a> <a href="http://www.itl.nist.gov/fipspubs/fip81.htm">http://www.itl.nist.gov/fipspubs/fip81.htm</a>
TEE_ALG_AES_CBC_MAC_NOPAD TEE_ALG_AES_CBC_MAC_PKCS5 TEE_ALG_DES_CBC_MAC_NOPAD TEE_ALG_DES_CBC_MAC_PKCS5 TEE_ALG_DES3_CBC_MAC_NOPAD TEE_ALG_DES3_CBC_MAC_PKCS5	FIPS 46 (DES, 3DES) FIPS 197 (AES) RFC 1423 (PKCS5 Pad)	<a href="http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf">http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf</a> <a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://tools.ietf.org/html/rfc1423">http://tools.ietf.org/html/rfc1423</a>
TEE_ALG_AES_CMAC	FIPS 197 (AES) NIST SP800-38B (CMAC)	<a href="http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf">http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf</a> <a href="http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf">http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf</a>

Name	References	URL
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	PKCS #1 (RSA, PKCS1 v1.5, PSS) RFC 1321 (MD5) FIPS 180-4 (SHA-1, SHA-2)	<a href="ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf">ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf</a> <a href="http://tools.ietf.org/html/rfc1321">http://tools.ietf.org/html/rfc1321</a> <a href="http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf">http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf</a>
TEE_ALG_DSA_SHA1 TEE_ALG_DSA_2048_SHA224 TEE_ALG_DSA_2048_SHA256 TEE_ALG_DSA_3072_SHA256	FIPS 180-4 (SHA-1) FIPS 186-2 (DSA)	<a href="http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf">http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf</a> <a href="http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf">http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf</a>
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	PKCS #1 (RSA, PKCS1 v1.5, OAEP) FIPS 180-4 (SHA-1, SHA-2)	<a href="ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf">ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf</a> <a href="http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf">http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf</a>
TEE_ALG_RSA_NOPAD	PKCS #1 (RSA primitive)	<a href="ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf">ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf</a>
TEE_ALG_DH_DERIVE_SHARED_SECRET	PKCS #3	<a href="ftp://ftp.rsasecurity.com/pub/pkcs/ps/pkcs-3.ps">ftp://ftp.rsasecurity.com/pub/pkcs/ps/pkcs-3.ps</a>
TEE_ALG_MD5	RFC 1321	<a href="http://tools.ietf.org/html/rfc1321">http://tools.ietf.org/html/rfc1321</a>

Name	References	URL
TEE_ALG_SHA1 TEE_ALG_SHA224 TEE_ALG_SHA256 TEE_ALG_SHA384 TEE_ALG_SHA512	FIPS 180-4	<a href="http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf">http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf</a>
TEE_ALG_HMAC_MD5 TEE_ALG_HMAC_SHA1	RFC 2202	<a href="http://tools.ietf.org/html/rfc2202">http://tools.ietf.org/html/rfc2202</a>
TEE_ALG_HMAC_SHA224 TEE_ALG_HMAC_SHA256 TEE_ALG_HMAC_SHA384 TEE_ALG_HMAC_SHA512	RFC 4231	<a href="http://tools.ietf.org/html/rfc4231">http://tools.ietf.org/html/rfc4231</a>
TEE_ALG_ECDSA_P192 TEE_ALG_ECDSA_P224 TEE_ALG_ECDSA_P256 TEE_ALG_ECDSA_P384 TEE_ALG_ECDSA_P521	FIPS 186-4 ANSI X9.62	<a href="http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf">http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf</a> <a href="http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005">http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005</a>
TEE_ALG_ECDH_P192 TEE_ALG_ECDH_P224 TEE_ALG_ECDH_P256 TEE_ALG_ECDH_P384 TEE_ALG_ECDH_P521	NIST SP800-56A Cofactor Static Unified Model FIPS 186-4 (curve definitions)	<a href="http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf">http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf</a> <a href="http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf">http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf</a>

# Functions

TA\_CloseSessionEntryPoint, 46  
TA\_CreateEntryPoint, 43  
TA\_DestroyEntryPoint, 43  
TA\_InvokeCommandEntryPoint, 47  
TA\_OpenSessionEntryPoint, 44  
TEE\_AEDecryptFinal, 166  
TEE\_AEEncryptFinal, 165  
TEE\_AEInit, 162  
TEE\_AEUpdate, 164  
TEE\_AEUpdateAAD, 163  
TEE\_AllocateOperation, 140  
TEE\_AllocatePersistentObjectEnumerator, 126  
TEE\_AllocatePropertyEnumerator, 60  
TEE\_AllocateTransientObject, 102  
TEE\_AsymmetricDecrypt, 167  
TEE\_AsymmetricEncrypt, 167  
TEE\_AsymmetricSignDigest, 169  
TEE\_AsymmetricVerifyDigest, 171  
TEE\_BigIntAdd, 204  
TEE\_BigIntAddMod, 211  
TEE\_BigIntCmp, 201  
TEE\_BigIntCmpS32, 201  
TEE\_BigIntComputeExtendedGcd, 217  
TEE\_BigIntComputeFMM, 221  
TEE\_BigIntConvertFromFMM, 220  
TEE\_BigIntConvertFromOctetString, 198  
TEE\_BigIntConvertFromS32, 200  
TEE\_BigIntConvertToFMM, 219  
TEE\_BigIntConvertToOctetString, 199  
TEE\_BigIntConvertToS32, 200  
TEE\_BigIntDiv, 209  
TEE\_BigIntFMMContextSizeInU32, 194  
TEE\_BigIntFMMSizeInU32, 194  
TEE\_BigIntGetBit, 202  
TEE\_BigIntGetBitCount, 203  
TEE\_BigIntInit, 195  
TEE\_BigIntInitFMM, 197  
TEE\_BigIntInitFMMContext, 196  
TEE\_BigIntInvMod, 215  
TEE\_BigIntIsProbablePrime, 218  
TEE\_BigIntMod, 210  
TEE\_BigIntMul, 207  
TEE\_BigIntMulMod, 213  
TEE\_BigIntNeg, 206  
TEE\_BigIntRelativePrime, 216  
TEE\_BigIntShiftRight, 202  
TEE\_BigIntSizeInU32 (macro), 193  
TEE\_BigIntSquare, 208  
TEE\_BigIntSquareMod, 214  
TEE\_BigIntSub, 205  
TEE\_BigIntSubMod, 212  
TEE\_CheckMemoryAccessRights, 79  
TEE\_CipherDoFinal, 157  
TEE\_CipherInit, 155  
TEE\_CipherUpdate, 156  
TEE\_CloseAndDeletePersistentObject (deprecated), 231  
TEE\_CloseAndDeletePersistentObject1, 124  
TEE\_CloseObject, 101  
TEE\_CloseTASession, 73  
TEE\_CopyObjectAttributes (deprecated), 230  
TEE\_CopyObjectAttributes1, 112  
TEE\_CopyOperation, 152  
TEE\_CreatePersistentObject, 119  
TEE\_DeriveKey, 173  
TEE\_DigestDoFinal, 154  
TEE\_DigestUpdate, 153  
TEE\_Free, 86  
TEE\_FreeOperation, 144  
TEE\_FreePersistentObjectEnumerator, 127  
TEE\_FreePropertyEnumerator, 60  
TEE\_FreeTransientObject, 105  
TEE\_GenerateKey, 114  
TEE\_GenerateRandom, 174  
TEE\_GetCancellationFlag, 77  
TEE\_GetInstanceData, 82  
TEE\_GetNextPersistentObject, 130  
TEE\_GetNextProperty, 63  
TEE\_GetObjectBufferAttribute, 99  
TEE\_GetObjectInfo (deprecated), 227  
TEE\_GetObjectInfo1, 96  
TEE\_GetObjectValueAttribute, 100  
TEE\_GetOperationInfo, 145  
TEE\_GetOperationInfoMultiple, 146  
TEE\_GetPropertyAsBinaryBlock, 57  
TEE\_GetPropertyAsBool, 55  
TEE\_GetPropertyAsIdentity, 59  
TEE\_GetPropertyAsString, 54  
TEE\_GetPropertyAsU32, 56  
TEE\_GetPropertyAsUUID, 58  
TEE\_GetPropertyName, 62  
TEE\_GetREETime, 189  
TEE\_GetSystemTime, 184  
TEE\_GetTAPersistentTime, 186

TEE_InitRefAttribute, 111	TEE_RenamePersistentObject, 125
TEE_InitValueAttribute, 111	TEE_ResetOperation, 148
TEE_InvokeTACommand, 74	TEE_ResetPersistentObjectEnumerator, 128
TEE_MACCompareFinal, 161	TEE_ResetPropertyEnumerator, 61
TEE_MACComputeFinal, 160	TEE_ResetTransientObject, 106
TEE_MACInit, 158	TEE_RestrictObjectUsage (deprecated), 229
TEE_MACUpdate, 159	TEE_RestrictObjectUsage1, 98
TEE_Malloc, 83	TEE_SeekObjectData, 135
TEE_MaskCancellation, 78	TEE_SetInstanceData, 82
TEE_MemCompare, 87	TEE_SetOperationKey, 149
TEE_MemFill, 88	TEE_SetOperationKey2, 151
TEE_MemMove, 86	TEE_SetTAPersistentTime, 188
TEE_OpenPersistentObject, 117	TEE_StartPersistentObjectEnumerator, 129
TEE_OpenTASession, 72	TEE_StartPropertyEnumerator, 61
TEE_Panic, 71	TEE_TruncateObjectData, 134
TEE_PopulateTransientObject, 107	TEE_UnmaskCancellation, 78
TEE_ReadObjectData, 131	TEE_Wait, 185
TEE_Realloc, 85	TEE_WriteObjectData, 133

# Functions by Category

## Asymmetric

TEE\_AsymmetricDecrypt, 167  
 TEE\_AsymmetricEncrypt, 167  
 TEE\_AsymmetricSignDigest, 169  
 TEE\_AsymmetricVerifyDigest, 171

## Authenticated Encryption

TEE\_AEDecryptFinal, 166  
 TEE\_AEEncryptFinal, 165  
 TEE\_AEInit, 162  
 TEE\_AEUpdate, 164  
 TEE\_AEUpdateAAD, 163

## Basic Arithmetic

TEE\_BigIntAdd, 204  
 TEE\_BigIntDiv, 209  
 TEE\_BigIntMul, 207  
 TEE\_BigIntNeg, 206  
 TEE\_BigIntSquare, 208  
 TEE\_BigIntSub, 205

## Cancellation

TEE\_GetCancellationFlag, 77  
 TEE\_MaskCancellation, 78  
 TEE\_UnmaskCancellation, 78

## Converter

TEE\_BigIntConvertFromOctetString, 198  
 TEE\_BigIntConvertFromS32, 200  
 TEE\_BigIntConvertToOctetString, 199  
 TEE\_BigIntConvertToS32, 200

## Data Stream Access

TEE\_ReadObjectData, 131  
 TEE\_SeekObjectData, 135  
 TEE\_TruncateObjectData, 134  
 TEE\_WriteObjectData, 133

## Deprecated

TEE\_CloseAndDeletePersistentObject, 231  
 TEE\_CopyObjectAttributes, 230  
 TEE\_GetObjectInfo, 227  
 TEE\_RestrictObjectUsage, 229

## Fast Modular Multiplication

TEE\_BigIntComputeFMM, 221  
 TEE\_BigIntConvertFromFMM, 220  
 TEE\_BigIntConvertToFMM, 219

## Generic Object

TEE\_CloseObject, 101  
 TEE\_GetObjectBufferAttribute, 99  
 TEE\_GetObjectInfo (deprecated), 227  
 TEE\_GetObjectInfo1, 96  
 TEE\_GetObjectValueAttribute, 100  
 TEE\_RestrictObjectUsage (deprecated), 229  
 TEE\_RestrictObjectUsage1, 98

## Generic Operation

TEE\_AllocateOperation, 140  
 TEE\_CopyOperation, 152

TEE\_FreeOperation, 144

TEE\_GetOperationInfo, 145  
 TEE\_GetOperationInfoMultiple, 146  
 TEE\_ResetOperation, 148  
 TEE\_SetOperationKey, 149  
 TEE\_SetOperationKey2, 151

## Initialization

TEE\_BigIntInit, 195  
 TEE\_BigIntInitFMM, 197  
 TEE\_BigIntInitFMMContext, 196

## Internal Client API

TEE\_CloseTASession, 73  
 TEE\_InvokeTACommand, 74  
 TEE\_OpenTASession, 72

## Key Derivation

TEE\_DeriveKey, 173

## Logical Operation

TEE\_BigIntCmp, 201  
 TEE\_BigIntCmpS32, 201  
 TEE\_BigIntGetBit, 202  
 TEE\_BigIntGetBitCount, 203  
 TEE\_BigIntShiftRight, 202

## MAC

TEE\_MACCompareFinal, 161  
 TEE\_MACComputeFinal, 160  
 TEE\_MACInit, 158  
 TEE\_MACUpdate, 159

## Memory Allocation and Size of Objects

TEE\_BigIntFMMContextSizeInU32, 194  
 TEE\_BigIntFMMSizeInU32, 194  
 TEE\_BigIntSizeInU32 (macro), 193

## Memory Management

TEE\_CheckMemoryAccessRights, 79  
 TEE\_Free, 86  
 TEE\_GetInstanceData, 82  
 TEE\_Malloc, 83  
 TEE\_MemCompare, 87  
 TEE\_MemFill, 88  
 TEE\_MemMove, 86  
 TEE\_Realloc, 85  
 TEE\_SetInstanceData, 82

## Message Digest

TEE\_DigestDoFinal, 154  
 TEE\_DigestUpdate, 153

## Modular Arithmetic

TEE\_BigIntAddMod, 211  
 TEE\_BigIntInvMod, 215  
 TEE\_BigIntMod, 210  
 TEE\_BigIntMulMod, 213  
 TEE\_BigIntSquareMod, 214  
 TEE\_BigIntSubMod, 212

## Other Arithmetic

- TEE\_BigIntComputeExtendedGcd, 217
- TEE\_BigIntIsProbablePrime, 218
- TEE\_BigIntRelativePrime, 216
- Panic Function
  - TEE\_Panic, 71
- Persistent Object
  - TEE\_CloseAndDeletePersistentObject (deprecated), 231
  - TEE\_CloseAndDeletePersistentObject1, 124
  - TEE\_CreatePersistentObject, 119
  - TEE\_OpenPersistentObject, 117
  - TEE\_RenamePersistentObject, 125
- Persistent Object Enumeration
  - TEE\_AllocatePersistentObjectEnumerator, 126
  - TEE\_FreePersistentObjectEnumerator, 127
  - TEE\_GetNextPersistentObject, 130
  - TEE\_ResetPersistentObjectEnumerator, 128
  - TEE\_StartPersistentObjectEnumerator, 129
- Property Access
  - TEE\_AllocatePropertyEnumerator, 60
  - TEE\_FreePropertyEnumerator, 60
  - TEE\_GetNextProperty, 63
  - TEE\_GetPropertyAsBinaryBlock, 57
  - TEE\_GetPropertyAsBool, 55
  - TEE\_GetPropertyAsIdentity, 59
  - TEE\_GetPropertyAsString, 54
  - TEE\_GetPropertyAsU32, 56
  - TEE\_GetPropertyAsUUID, 58
  - TEE\_GetPropertyName, 62
  - TEE\_ResetPropertyEnumerator, 61
  - TEE\_StartPropertyEnumerator, 61
- Random Data Generation
  - TEE\_GenerateRandom, 174
- Symmetric Cipher
  - TEE\_CipherDoFinal, 157
  - TEE\_CipherInit, 155
  - TEE\_CipherUpdate, 156
- TA Interface
  - TA\_CloseSessionEntryPoint, 46
  - TA\_CreateEntryPoint, 43
  - TA\_DestroyEntryPoint, 43
  - TA\_InvokeCommandEntryPoint, 47
  - TA\_OpenSessionEntryPoint, 44
- Time
  - TEE\_GetREETime, 189
  - TEE\_GetSystemTime, 184
  - TEE\_GetTAPersistentTime, 186
  - TEE\_SetTAPersistentTime, 188
  - TEE\_Wait, 185
- Transient Object
  - TEE\_AllocateTransientObject, 102
  - TEE\_CopyObjectAttributes (deprecated), 230
  - TEE\_CopyObjectAttributes1, 112
  - TEE\_FreeTransientObject, 105
  - TEE\_GenerateKey, 114
  - TEE\_InitRefAttribute, 111
  - TEE\_InitValueAttribute, 111
  - TEE\_PopulateTransientObject, 107
  - TEE\_ResetTransientObject, 106