GLOBALPLATFORM®
THE STANDARD FOR SECURE DIGITAL SERVICES AND DEVICES

# GlobalPlatform Technology
# TEE Trusted User Interface Low-level API
# Version 1.0

**Public Release**

**March 2018**

**Document Reference: GPD_SPE_055**

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

# Contents

**1 Introduction ..... 8**
1.1 Audience ..... 8
1.2 IPR Disclaimer ..... 8
1.3 References ..... 9
1.4 Terminology and Definitions ..... 10
1.5 Abbreviations and Notations ..... 12
1.6 Revision History ..... 13
**2 Trusted User Interface Objectives ..... 14**
2.1 Target ..... 14
2.2 Purpose ..... 14
2.3 Functionalities and Use Cases ..... 15
**3 Trusted User Interface Principles ..... 16**
3.1 Overall Architecture ..... 16
3.2 Event Loop and Peripherals ..... 18
3.2.1 Peripherals ..... 18
3.2.1.1 Access to Peripherals from a TA ..... 18
3.2.1.2 Multiple Access to Peripherals (Informative) ..... 19
3.2.2 Event Loop ..... 20
3.2.3 Peripheral State ..... 20
3.2.4 Handles ..... 20
3.3 Peripheral State Table ..... 21
3.3.1 Peripheral Name ..... 21
3.3.2 Firmware Information ..... 21
3.3.3 Manufacturer ..... 22
3.3.4 Flags ..... 22
3.3.5 Exclusive Access ..... 23
3.4 Operating System Pseudo-peripheral ..... 24
3.4.1 State Table ..... 24
3.4.2 Events ..... 24
3.5 Security Indicator ..... 25
3.6 TUI Session ..... 25
3.7 Image Formats ..... 26
3.8 Power and OS Events Management ..... 26
3.9 Display Orientation ..... 27
3.10 Specification Version Number Property ..... 27
3.11 Structure Versions ..... 28
**4 Trusted User Interface Low-level API ..... 29**
4.1 Implementation Properties ..... 29
4.2 Header File ..... 30
4.2.1 API Version ..... 30
4.2.2 Version Compatibility Definitions ..... 31
4.3 Data Constants ..... 33
4.3.1 Return Codes ..... 33
4.3.2 Maximum Sizes ..... 33
4.3.3 TEE_EVENT_TYPE ..... 34
4.3.4 TEE_EVENT_TUI_BUTTON_ACTION ..... 36
4.3.5 TEE_EVENT_TUI_BUTTON_TYPE ..... 37
4.3.6 TEE_EVENT_TUI_KEY_ACTION ..... 38

4.3.7 TEE_EVENT_TUI_REE_TYPE ..... 39
4.3.8 TEE_EVENT_TUI_TEE_TYPE ..... 40
4.3.9 TEE_EVENT_TUI_TOUCH_ACTION ..... 41
4.3.10 TEE_PERIPHERAL_TYPE ..... 42
4.3.11 TEE_PERIPHERAL_FLAGS ..... 43
4.3.12 TEE_PeripheralStateId Values ..... 44
4.3.12.1 TEE_PERIPHERAL_TUI_BUTTON State Table ..... 44
4.3.12.2 TEE_PERIPHERAL_TUI_KEYBOARD State Table ..... 45
4.3.12.3 TEE_PERIPHERAL_TUI_TOUCH State Table ..... 45
4.3.13 TEE_TUI_DISPLAY_INFO_FLAGS ..... 46
4.3.14 TEE_TUI_INIT_SESSION_LOW_FLAGS ..... 46
4.3.15 TEE_TUI_OPERATION ..... 47
4.3.16 TEE_TUI_SURFACE_OPACITY ..... 48
4.4 Data Types ..... 49
4.5 Data Structures ..... 49
4.5.1 TEE_Peripheral ..... 49
4.5.2 TEE_PeripheralDescriptor ..... 50
4.5.3 TEE_PeripheralHandle ..... 51
4.5.4 TEE_PeripheralId ..... 51
4.5.5 TEE_PeripheralState ..... 52
4.5.6 TEE_PeripheralStateId ..... 53
4.5.7 TEE_PeripheralValueType ..... 53
4.5.8 TEE_Event ..... 54
4.5.9 Generic Payloads ..... 55
4.5.9.1 TEE_Event_AccessChange ..... 55
4.5.10 TEE_Event TUI Payloads ..... 56
4.5.10.1 TEE_Event_TUI_Button ..... 56
4.5.10.2 TEE_Event_TUI_Keyboard ..... 57
4.5.10.3 TEE_Event_TUI_REE ..... 58
4.5.10.4 TEE_Event_TUI_TEE ..... 59
4.5.10.5 TEE_Event_TUI_Touch ..... 59
4.5.11 TEE_EventQueueHandle ..... 60
4.5.12 TEE_EventSourceHandle ..... 61
4.5.13 TEE_TUIDisplay ..... 61
4.5.14 TEE_TUIDisplayInfo ..... 62
4.5.15 TEE_TUIImage ..... 63
4.5.16 TEE_TUIImageSource ..... 64
4.5.17 TEE_TUISurface ..... 64
4.5.18 TEE_TUISurfaceBuffer ..... 65
4.5.19 TEE_TUISurfaceInfo ..... 66
4.6 Functions ..... 67
4.6.1 TEE_Peripheral_Close ..... 67
4.6.2 TEE_Peripheral_CloseMultiple ..... 68
4.6.3 TEE_Peripheral_GetPeripherals ..... 69
4.6.4 TEE_Peripheral_GetState ..... 70
4.6.5 TEE_Peripheral_GetStateTable ..... 71
4.6.6 TEE_Peripheral_Open ..... 72
4.6.7 TEE_Peripheral_OpenMultiple ..... 74
4.6.8 TEE_Peripheral_Read ..... 76
4.6.9 TEE_Peripheral_SetState ..... 78
4.6.10 TEE_Peripheral_Write ..... 79
4.6.11 TEE_Event_AddSources ..... 80

4.6.12 TEE_Event_CancelSources ..... 81
4.6.13 TEE_Event_CloseQueue ..... 82
4.6.14 TEE_Event_DropSources ..... 83
4.6.15 TEE_Event_ListSources ..... 84
4.6.16 TEE_Event_OpenQueue ..... 85
4.6.17 TEE_Event_Wait ..... 87
4.6.18 TEE_TUI_BlitDisplaySurface ..... 89
4.6.19 TEE_TUI_CloseSession ..... 90
4.6.20 TEE_TUI_DrawRectangle ..... 91
4.6.21 TEE_TUI_GetDisplayInformation ..... 92
4.6.22 TEE_TUI_GetDisplaySurface ..... 94
4.6.23 TEE_TUI_GetPNGInformation ..... 95
4.6.24 TEE_TUI_GetSurface ..... 96
4.6.25 TEE_TUI_GetSurfaceBuffer ..... 97
4.6.26 TEE_TUI_GetSurfaceInformation ..... 98
4.6.27 TEE_TUI_InitSessionLow ..... 99
4.6.28 TEE_TUI_ReleaseSurface ..... 101
4.6.29 TEE_TUI_SetPNG ..... 102
4.6.30 TEE_TUI_TransformSurface ..... 103

**Annex A Panicked Function Identification ..... 105**

**Annex B TEE TUI Low-level API Usage ..... 107**

# Figures

Figure 3-1: TEE with TUI Architecture ..... 16
Figure 3-2: Example of Multiple Access to Bus-oriented Peripheral (Informative) ..... 19

# Tables

Table 1-1: Normative References ..... 9
Table 1-2: Informative References ..... 9
Table 1-3: Terminology and Definitions ..... 10
Table 1-4: Abbreviations and Notations ..... 12
Table 1-5: Revision History ..... 13
Table 2-1: Comparison of TUI High-level and Low-level API Capabilities ..... 14
Table 3-1: TEE_PERIPHERAL_STATE_NAME Values ..... 21
Table 3-2: TEE_PERIPHERAL_STATE_FW_INFO Values ..... 21
Table 3-3: TEE_PERIPHERAL_STATE_MANUFACTURER Values ..... 22
Table 3-4: TEE_PERIPHERAL_STATE_FLAGS Values ..... 22
Table 3-5: TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS Values ..... 23
Table 3-6: TEE_PERIPHERAL_OS State Table Values ..... 24
Table 3-7: Specification Version Number Property – 32-bit Integer Structure ..... 27
Table 3-8: Structure Versions ..... 28
Table 4-1: Implementation Properties ..... 29
Table 4-2: Return Codes ..... 33
Table 4-3: Maximum Sizes of Structure Payloads ..... 33
Table 4-4: TEE_EVENT_TYPE Values ..... 35
Table 4-5: TEE_EVENT_TUI_BUTTON_ACTION Values ..... 36
Table 4-6: TEE_EVENT_TUI_BUTTON_TYPE Values ..... 37
Table 4-7: TEE_EVENT_TUI_KEY_ACTION Values ..... 38
Table 4-8: TEE_EVENT_TUI_REE_TYPE Values ..... 39
Table 4-9: TEE_EVENT_TUI_TEE_TYPE Values ..... 40
Table 4-10: TEE_EVENT_TUI_TOUCH_ACTION Values ..... 41
Table 4-11: TEE_PERIPHERAL_TYPE Values ..... 42
Table 4-12: TEE_PERIPHERAL_FLAGS Values ..... 43

Table 4-13: TEE_PeripheralStateId Values ..... 44
Table 4-14: TEE_PERIPHERAL_TUI_BUTTON State Table ..... 44
Table 4-15: TEE_PERIPHERAL_TUI_KEYBOARD State Table ..... 45
Table 4-16: TEE_PERIPHERAL_TUI_TOUCH State Table ..... 45
Table 4-17: TEE_TUI_DISPLAY_INFO_FLAGS Values ..... 46
Table 4-18: TEE_TUI_INIT_SESSION_LOW_FLAGS Values ..... 46
Table 4-19: TEE_TUI_OPERATION Values ..... 47
Table 4-20: TEE_TUI_SURFACE_OPACITY Values ..... 48
Table 4-21: TEE_PeripheralValueType Values ..... 53
Table 4-22: TEE_TUIImageSource Values ..... 64
Table A-1: Function Identification Values ..... 105

# 1 Introduction

Many sensitive use cases lead to an interaction with the user. These are mainly, but not exclusively, related to financial services and corporate usages: bill payment, money transfer, document signature validation, privacy, etc.

While the TEE Internal Core API ([TEE Core API]) offers the possibility to execute all sensitive operations within a Trusted Application (TA) running in the Trusted Execution Environment (TEE), certain applications need to expose sensitive information to the user for validation or need to get sensitive information from the user. Entering a PIN and signing a document are examples of operations that need to be handled inside the TEE for the Trusted Application and not to rely on facilities in the Rich Execution Environment (REE).

This specification defines a low-level interface to a frame buffer and input events, which the Trusted Application can use to create custom user interfaces. The event loop can also be used to retrieve events from other peripherals not related to TUI. Because of this, the event loop will be moved to [TEE Core API] in v1.2.

The TEE Trusted User Interface API ([TEE TUI]) provides only a high-level interface that enables a Trusted Application to display a limited number of screen types, including a password entry screen. However, it is not possible to define every possible user interaction using those screens.

A Trusted Application can choose to open either a high-level or a low-level session. A single Trusted Application may do both, at different times. However, a session – whether high- or low-level – claims an exclusive lock on a display, so it is not possible for a TA to open two sessions at the same time. (Multiple TAs may each open a low-level session, as long as they don't access any of the same resources.)

It is possible to create all the high-level screens using the low-level API. It is permissible for a Trusted OS to use a linked library to provide those high-level functions in a Trusted Application, that library making use of the low-level API.

This document is an addendum to [TEE Core API].

## 1.1 Audience

This document is intended to support software developers implementing Trusted Applications running inside the TEE that need to display sensitive information to the user or retrieve sensitive data from the user.

This document is also intended for implementers of the Trusted User Interface in the TEE itself.

## 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 References

**Table 1-1: Normative References**

| Standard / Specification | Description | Ref |
|---|---|---|
| GPD_SPE_010 | GlobalPlatform Technology<br>TEE Internal Core API Specification | [TEE Core API] |
| GPD_SPE_042 | GlobalPlatform Technology<br>TEE TUI Extension: Biometrics API | [TEE TUI Bio] |
| GPD_SPE_120 | GlobalPlatform Technology<br>TEE Management Framework | [TEE Mgmt Fmwk] |
| ISO/IEC 9899:1999 | Programming languages – C | [C99] |
| PNG | ISO/IEC 15948:2004 – Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification | [ISO 15948] |
| RFC 2119 | Key words for use in RFCs to Indicate Requirement Levels | [RFC 2119] |
| USB HID Usage Tables | Table 12, page 53, Version 1.12 of 'Universal Serial Bus HID Usage Tables' published 28/10/2004 by the USB Implementers' Forum | [USB Key] |
| USB HID Country Codes | Section 6.2.1, page 23, Version 1.11 of 'Device Class Definition for Human Interface Devices (HID)' published 27/06/2001 by the USB Implementers' Forum | [USB Country] |
| USB Human Interface Devices | Section 8.3, page 56, Version 1.11 of 'Device Class Definition for Human Interface Devices (HID)', published 27/06/2001 by the USB Implementers' Forum | [USB HID] |

**Table 1-2: Informative References**

| Standard / Specification | Description | Ref |
|---|---|---|
| GPD_SPE_007 | GlobalPlatform Technology<br>TEE Client API Specification | [TEE Client API] |
| GPD_SPE_009 | GlobalPlatform Technology<br>TEE System Architecture | [TEE Sys Arch] |
| GPD_SPE_020 | GlobalPlatform Technology<br>TEE Trusted User Interface API | [TEE TUI] |
| GPD_SPE_021 | GlobalPlatform Technology<br>TEE Protection Profile | [TEE PP] |
| GPD_SPE_025 | GlobalPlatform Technology<br>TEE TA Debug Specification | [TEE TA Debug] |
| GP_GUI_001 | GlobalPlatform Document Management Guide | [Doc Mgmt] |

| Standard / Specification | Description | Ref |
|---|---|---|
| Porter/Duff Compositing | T. Porter and T. Duff Compositing Digital Images<br>Computer Graphics Volume 18, Number 3 July 1984 pp 253-259 available<br>http://keithp.com/~keithp/porterduff/p253-porter.pdf | [Porter Duff] |

## 1.4 Terminology and Definitions

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

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.

**Table 1-3: Terminology and Definitions**

| Term | Definition |
|---|---|
| Client Application (CA) | An application running outside of the Trusted Execution Environment (TEE) making use of the TEE Client API ([TEE Client API]) to access facilities provided by Trusted Applications inside the TEE.<br>Contrast *Trusted Application (TA)*. |
| Data Object | An object containing a data stream but no key material. |
| Display | A physical integral rendering component or a remote rendering component connected via a video port providing one or more protected data paths and equivalent security to an integral component. |
| Event API | An API that supports the event loop. Includes the following functions, among others:<br>`TEE_Event_AddSources`<br>`TEE_Event_OpenQueue`<br>`TEE_Event_Wait`<br>The Event API is currently defined in this specification, but will be defined in [TEE Core API] beginning with v1.2. |
| Event loop | A mechanism by which a TA can enquire for and then process messages from types of peripherals including pseudo-peripherals. |
| Object Identifier | A variable length binary buffer identifying a persistent object. |
| Panic | An exception that kills a whole TA instance. See [TEE Core API] section 2.3.3 for full definition. |

| Term | Definition |
|---|---|
| Peripheral API | A low-level API that enables a Trusted Application to interact with peripherals via the Trusted OS. Includes the following functions, among others:<br>`TEE_Peripheral_GetPeripherals`<br>`TEE_Peripheral_GetStateTable`<br>`TEE_Peripheral_Open`<br>The Peripheral API is currently defined in this specification, but will be defined in [TEE Core API] beginning with v1.2. |
| Property | An immutable value identified by a name. |
| 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 untrusted.<br>Contrast *Trusted Execution Environment (TEE)*. |
| Rich OS | Typically, an OS providing a much wider variety of features than are provided by the OS running inside the TEE. It is very open in its ability to accept applications. It will have been developed with functionality and performance as key goals, rather than security. Due to its size and needs, the Rich OS will run in an execution environment outside of the TEE hardware (often called an REE – Rich Execution Environment) with much lower physical security boundaries. From the TEE viewpoint, everything in the REE is considered untrusted, though from the Rich OS point of view there may be internal trust structures.<br>Contrast *Trusted OS*. |
| Screen | A defined combination of displayed graphical components along with a set of defined reactions to user interaction with those components through an input device such as keyboard or touchscreen. This terminology is used extensively in [TEE TUI]. |
| Session | Logically connects multiple functions invoked on a Trusted Application. |
| Trusted Application (TA) | An application running inside the Trusted Execution Environment (TEE) that provides security-related functionality to Client Applications outside of the TEE or to other Trusted Applications inside the TEE.<br>Contrast *Client Application (CA)*. |
| Trusted Execution Environment (TEE) | An execution environment that runs alongside but isolated from an REE.<br>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.<br>Contrast *Rich Execution Environment (REE)*. |
| Trusted OS | An operating system running in the TEE providing the TEE Internal Core API to Trusted Applications.<br>Contrast *Rich OS*. |

| Term | Definition |
|---|---|
| Trusted Storage | Storage that is protected either by the hardware of the TEE or cryptographically by keys held in the TEE, and that contains data that is accessible only to the Trusted Application that created the data. |
| Trusted User Interface (Trusted UI or TUI) | A user interface that ensures that the displays and input components are controlled by the TEE and isolated from the REE and even the TAs. |

## 1.5 Abbreviations and Notations

**Table 1-4: Abbreviations and Notations**

| Abbreviation / Notation | Meaning |
|---|---|
| API | Application Programming Interface |
| GPU | Graphics Processing Unit |
| ID | Identifier |
| IEC | International Electrotechnical Commission |
| IETF | Internet Engineering Task Force |
| ISO | International Organization for Standardization |
| LED | Light Emitting Diode |
| LS | Liaison Statement |
| NFC | Near Field Communication |
| PIN | Personal Identification Number |
| PNG | Portable Network Graphics |
| REE | Rich Execution Environment |
| RFC | Request For Comments; may denote a memorandum published by the IETF |
| RFU | Reserved for future use |
| RGB | Red Green Blue; an additive color model |
| SDO | Standards Defining Organization |
| TA | Trusted Application |
| TEE | Trusted Execution Environment |
| TUI | Trusted User Interface |
| UI | User Interface |
| VNC | Virtual Network Computing |

## 1.6 Revision History

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

**Table 1-5: Revision History**

| Date | Version | Description |
|---|---|---|
| March 2018 | 1.0 | Public Release |

# 2 Trusted User Interface Objectives

## 2.1 Target

The low-level API defined in this specification is targeted at a TEE running within a device equipped with at least one display and optionally a keypad (e.g. smartphone, tablet, set-top box). If no display has touch input capable of being controlled by the TEE, the device SHALL have a wired keyboard or a biometric device under TEE control.

This specification assumes that the TUI components are wired and integral to the device. Remote peripherals are not explicitly described but SHALL provide one or more protected data paths and equivalent security to an integral component.

## 2.2 Purpose

Table 2-1 compares the capabilities of the high-level Trusted User Interface API described in [TEE TUI] and the low-level TUI API described in this specification.

**Table 2-1: Comparison of TUI High-level and Low-level API Capabilities**

| Capability | TEE Trusted User Interface API ([TEE TUI]) | TEE TUI Low-level API (this specification) |
|---|---|---|
| Graphic Display | Limited graphic capability. | Flexible graphic capability. |
| Secure Display | Information displayed to the user cannot be accessed, modified, or obscured by any software within the REE or by an unauthorized application in the TEE. | Same |
| Secure Input | Information entered by the user cannot be derived or modified by any software within the REE or by an unauthorized application in the TEE. | Same |
| Screen Types | Limited number of screen types, including a password entry screen. | Using a frame buffer and input events, the Trusted Application can create custom user interfaces. |
| Session Concurrence | While a Trusted Application has a high-level session open, it cannot open another session, either high- or low-level, nor can any other TA. | While a Trusted Application has a low-level session open, it cannot open another session, either high- or low-level.<br>Different TAs may open concurrent sessions provided they use separate displays. |
| Security Indicator | The security indicator may be a physical indicator, or it may be an image known to the user and displayed on screen by the TEE. | The security indicator must be one of the following:<br>• A separate physical indicator under the control of the TEE and never accessible to the REE<br>• An area of the screen permanently under the control of the TEE and never under the control of the REE or a TA |

## 2.3 Functionalities and Use Cases

[TEE TUI] presents a high-level interface that allows a Trusted Application to display a few basic screens, such as PIN entry; however, this is not suitable for all situations.

This specification presents a low-level interface which offers a secure frame buffer to which the Trusted Application can write an image.

This image may be created by any of the following:

- The Trusted Application itself
- A remote server, with the TA only rendering the image into memory in a manner analogous to a VNC server
- A combination of the above

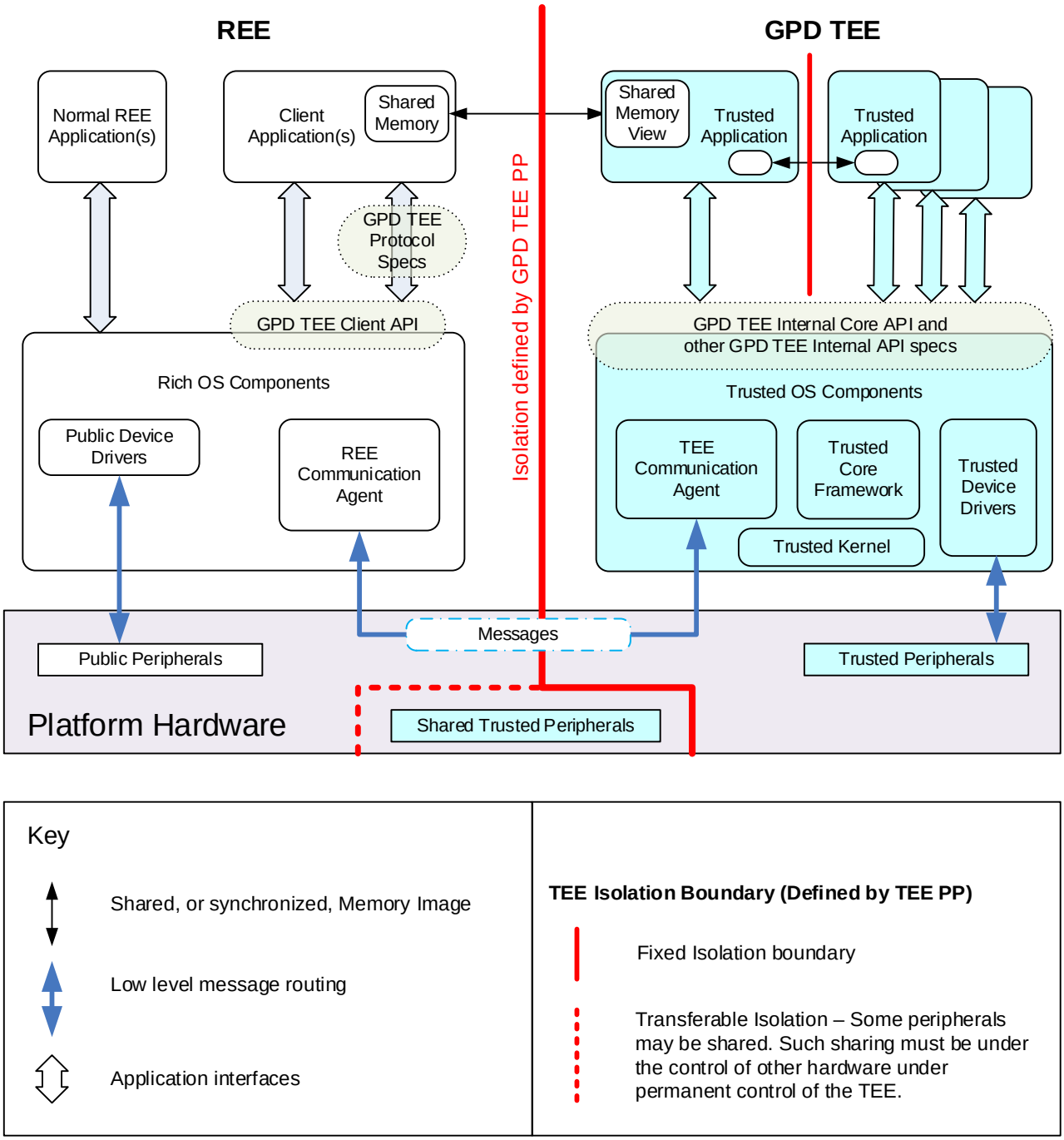
To provide feedback, the TEE offers an event queue that captures touch events, keyboard events, events created from physical buttons, and events from the REE or other sources.

An implementation may also lock event streams from other peripherals, even if the TEE does not have a driver that can interpret information from those peripherals, preventing information from the selected peripherals from being sent to an REE application. This helps eliminate side channels that could otherwise leak information about the user interface. Such peripherals may include, but are not limited to, cameras, accelerometers, gyroscopes, microphones, proximity sensors, or magnetometers.

# 3 Trusted User Interface Principles

## 3.1 Overall Architecture

**Figure 3-1: TEE with TUI Architecture**




A typical architecture implementing the Trusted User Interface consists of either a touchscreen or keyboard peripheral and a display controller peripheral. When a Trusted User Interface is displayed, those peripherals SHALL NOT be accessible for reading or writing by the REE and indication of associated events SHALL NOT be received by the REE. At other times, it is up to a specific platform or a specific Trusted OS either to give back control of those peripherals to the REE or to provide some other method to allow the REE access to the data and events from those peripherals.

This specification permits an implementation with multiple displays. The TEE can choose whether to allow different Trusted Applications to run sessions on the different displays. This will depend on the exact configuration. In a single user device, such as a mobile phone, the TEE will probably run only a single TUI session, but in an infotainment system in a car or plane, it is likely that it will need to support separate simultaneous sessions.

In most cases, remote displays connected via USB or HDMI are under the control of the REE. Such displays SHALL NOT be used for TUI sessions. However, a TEE that can establish a sufficiently secure path to the display MAY be used.

## 3.2 Event Loop and Peripherals

### 3.2.1 Peripherals

A peripheral is an ancillary component used to interact with a system, with the software interface between peripheral and system being provided by a device driver. On a typical device that includes a TEE, there may be many peripherals. The TEE is not expected to have software drivers for interacting with every peripheral attached to the device.

There are several classes of peripheral:

- Peripherals that are temporarily or permanently isolated from non-TEE entities, managed by the TEE, and fully usable by a TA through the APIs the TEE offers. These devices are described as TEE ownable.
- Peripherals that are under the total control of the REE or other entity outside the TEE, and are not usable by the TEE.
- Peripherals where the TEE cannot interpret events – because it does not have the required driver – but where the TEE can control the flow of events, for example by routing flow through the TEE or by controlling the clock on a bus. These devices are described as TEE controllable.
- Peripherals for which a TEE can parse and forward events, even though the TEE does not fully control that source; e.g. a sockets interface to the REE. As the interface is hosted by the REE, it is REE controlled, but TEE parseable.

TA and TEE implementers should be aware of potential side channel attacks and provide and/or control appropriate interfaces to restrict those attacks. For example, a TEE could be configured to stop access by entities outside the TEE to specific peripherals such as accelerometers to prevent indirect interpretation of touch screen use during a TUI session.

The `TEE_Peripheral_GetPeripherals` function enables the TA to discover which peripherals the TEE knows about, and their characteristics, while other functions support low-level interaction with peripherals.

When a data source (or sink) is handed back to the REE, or transferred between TA instances, any state specific to previous TA activity or TA/user interaction SHALL be removed to prevent information leakage.

#### 3.2.1.1 Access to Peripherals from a TA

Peripherals which are under the full or partial control of the TEE (i.e. peripherals which are TEE ownable, TEE parseable, or TEE controllable) MAY support exclusive access by no more than one TA at any one time.

A Trusted OS MAY provide additional access control mechanisms which are out of scope of this specification, either because they are described in separate GlobalPlatform specifications or because they are implementation-specific. An (informative) example is a Trusted OS that limits access to a peripheral to those TAs that reside in specific security domains.

The Trusted OS SHALL recover ownership of all peripherals with open handles from a TA in the following scenarios:

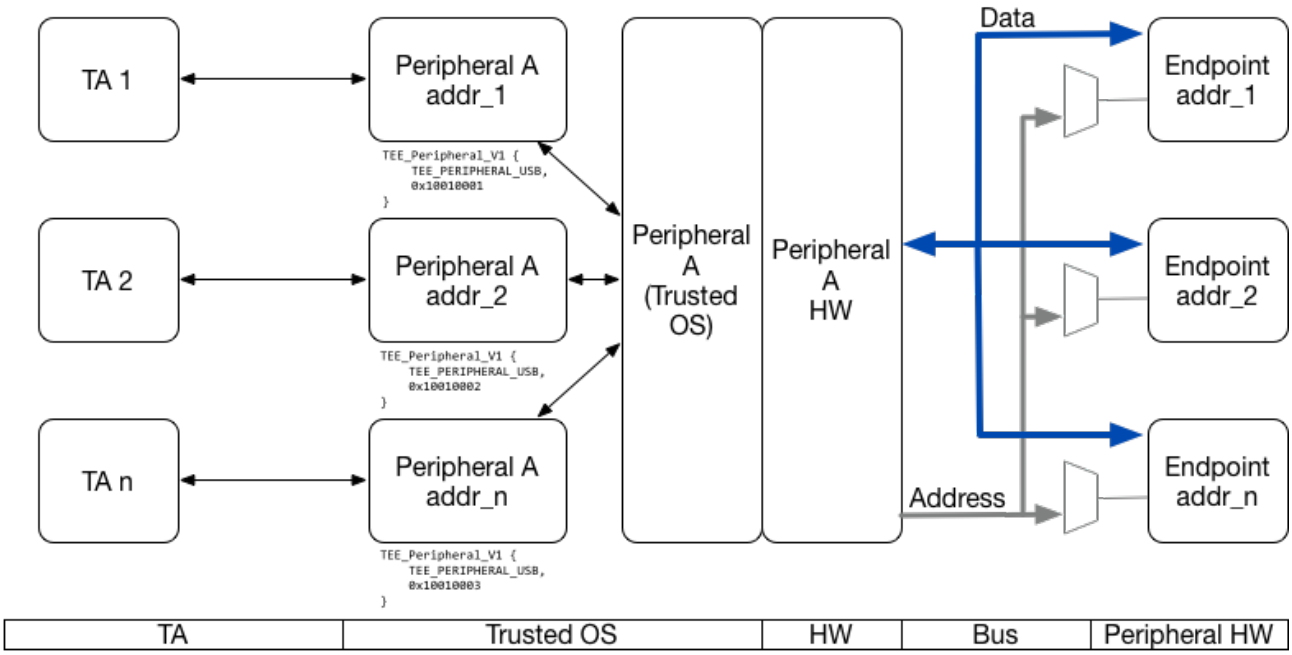
- The TA Panics.
- `TA_DestroyEntryPoint` is called for the TA owning the peripheral.

#### 3.2.1.2 Multiple Access to Peripherals (Informative)

Some peripherals offer multiple channels, addressing capability, or other mechanisms which have the potential to allow access to multiple endpoints. It may be convenient in some scenarios to assign different logical endpoints to different TAs, while supporting a model of exclusive access to the peripheral per TA.

One approach, shown in Figure 3-2, is to implement a separate driver interface for each of the multiple endpoints. For example, a driver for an I²C interface may support separate endpoints for each I²C address, while itself being the exclusive owner of the I²C peripheral. Such drivers SHOULD ensure that information leakage between clients of the different endpoints is prevented.

**Figure 3-2: Example of Multiple Access to Bus-oriented Peripheral (Informative)**



### 3.2.2 Event Loop

The event loop is a mechanism by which a TA can enquire for and then process messages from types of peripherals including pseudo-peripherals. The event loop can simplify TA programming in scenarios where peripheral interaction occurs asynchronously with respect to TEE operation.

Events are polymorphic, with the ability to transport device-specific payloads.

The underlying implementation of the event loop is implementation-dependent; however, the Trusted OS SHALL ensure that:

- A TA can only successfully obtain an event source for a peripheral for which it already has an open handle. This ensures that if a peripheral supports exclusive access by a single TA, sensitive information coming from a peripheral can be consumed by only that TA, preventing opportunities for information leakage.
- Events submitted to the event queue for a given peripheral are submitted in the order in which they occur. No guarantee is made of the ordering of events from different peripherals.
- An error scenario in the event subsystem which results in a Panic SHALL NOT cause a Panic in TAs which are blocked waiting on synchronous operations. It will either be attributed to a TEE level problem (e.g. a corrupt library) or will occur in the `TEE_Event_Wait` function.

### 3.2.3 Peripheral State

The Peripheral API provides an abstracted interface to some of the hardware features of the underlying device. It can be desirable to enable a TA to read and/or configure the hardware in a specific way, for example it may be necessary to set data transmission rates on a serial peripheral, or to discover the manufacturer of a biometric sensor

The Peripheral API provides a mechanism by which TAs can discover information about the peripherals they use, and by which modifiable parameters can be identified and updated. It is intended to ensure that peripherals for which GlobalPlatform specifies interfaces can be used in a portable manner by TAs.

It is expected that other GlobalPlatform specifications may define state items for peripherals.

### 3.2.4 Handles

***Note:*** This handle will be defined in [TEE Core API] v1.2 and later.

The value `TEE_INVALID_HANDLE` is used by the peripheral subsystem to denote an invalid handle.

```
#define TEE_INVALID_HANDLE   ((TEE_EventQueueHandle) (0))
```

## 3.3 Peripheral State Table

Every peripheral instance has a table of associated state information. A TA can obtain this table by calling `TEE_Peripheral_GetStateTable`. Each item in the state table is of `TEE_PeripheralState` type.

The peripheral state table can be used to retrieve standardized, and peripheral specific, information about the peripheral. It also contains identifiers that can then be used for direct get/put control specific aspects of the peripheral.

For example, a serial interface peripheral may expose interfaces to its control registers to provide direct access to readable parity error counters and writable baud rate settings.

The state table returned by `TEE_Peripheral_GetStateTable` is a read-only snapshot of peripheral state at function call time. Some of the values in the table may support modification by the caller using the `TEE_Peripheral_SetState` function – this is indicated by the value of the `ro` field.

The following sections define the state table items which could be present in the peripheral state table. Other specifications may define additional items.

### 3.3.1 Peripheral Name

***Note:*** This state table entry will be defined in [TEE Core API] v1.2 and later.

Peripherals SHALL provide a state table entry that defines a printable name for the peripheral.

**Table 3-1: `TEE_PERIPHERAL_STATE_NAME` Values**

| `TEE_PeripheralValueType` Field | Value |
|---|---|
| `tag` | `TEE_PERIPHERAL_VALUE_STRING` |
| `id` | `TEE_PERIPHERAL_STATE_NAME` |
| `ro` | `true` |
| `u.stringVal` | Pointer to a NULL-terminated printable string which contains a printable peripheral name; SHOULD be unique among the peripherals that are presented to a given TA |

### 3.3.2 Firmware Information

***Note:*** This state table entry will be defined in [TEE Core API] v1.2 and later.

Peripherals MAY provide a state table entry that identifies the firmware version executing on the peripheral. This entry is only relevant to peripherals which contain a processor.

**Table 3-2: `TEE_PERIPHERAL_STATE_FW_INFO` Values**

| `TEE_PeripheralValueType` Field | Value |
|---|---|
| `tag` | `TEE_PERIPHERAL_VALUE_STRING` |
| `id` | `TEE_PERIPHERAL_STATE_FW_INFO` |
| `ro` | `true` |
| `u.stringVal` | Pointer to a NULL-terminated printable string which contains information about the firmware running in the peripheral |

### 3.3.3 Manufacturer

***Note:*** This state table entry will be defined in [TEE Core API] v1.2 and later.

Peripherals MAY provide a state table entry that identifies the manufacturer of the peripheral.

**Table 3-3: `TEE_PERIPHERAL_STATE_MANUFACTURER` Values**

| `TEE_PeripheralValueType` Field | Value |
|---|---|
| `tag` | `TEE_PERIPHERAL_VALUE_STRING` |
| `id` | `TEE_PERIPHERAL_STATE_MANUFACTURER` |
| `ro` | `true` |
| `u.stringVal` | Pointer to a NULL-terminated printable string which contains information about the manufacturer of the peripheral |

### 3.3.4 Flags

***Note:*** This state table entry will be defined in [TEE Core API] v1.2 and later.

Peripherals SHALL provide a state table entry that provides information about the way in which the Trusted OS can manage the input and output from this peripheral from the calling TA using one or more of the values defined for `TEE_PERIPHERAL_FLAGS` – these may be combined in a bitwise manner.

**Table 3-4: `TEE_PERIPHERAL_STATE_FLAGS` Values**

| `TEE_PeripheralValueType` Field | Value |
|---|---|
| `tag` | `TEE_PERIPHERAL_VALUE_UINT32` |
| `id` | `TEE_PERIPHERAL_STATE_FLAGS` |
| `ro` | `true` |
| `u.uint32Val` | A combination of zero or more of the `TEE_PERIPHERAL_FLAGS` values defined in section 4.3.11 |

### 3.3.5 Exclusive Access

***Note:*** This state table entry will be defined in [TEE Core API] v1.2 and later.

Peripherals SHALL provide a state table entry that identifies whether the peripheral supports exclusive access.

**Table 3-5: `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` Values**

| `TEE_PeripheralValueType` Field | Value |
|---|---|
| `tag` | `TEE_PERIPHERAL_VALUE_BOOL` |
| `id` | `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` |
| `ro` | `true` |
| `u.boolVal` | Set to `true` if this peripheral can be opened for exclusive access. |

**Note:** The value of the `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` field SHALL be set to the same value on all TAs running on a given TEE which have access to that peripheral.

## 3.4 Operating System Pseudo-peripheral

Any Trusted OS that supports the Peripheral API SHALL provide a single instance of a TEE pseudo-peripheral. This SHALL be exposed by a TEE pseudo-peripheral with a `TEE_PERIPHERAL_TYPE` of `TEE_PERIPHERAL_OS`. This is used to pass events originating in the Trusted and Rich Operating Systems. The type of the event indicates the original source of the event.

The TEE pseudo-peripheral SHALL NOT expose a `TEE_PeripheralHandle`, as it supports neither the polled Peripheral API nor writeable state. It SHALL expose a `TEE_EventSourceHandle`.

The TEE pseudo-peripheral SHALL NOT be lockable for exclusive access, and SHALL be exposed to all TA instances. Any TA in the Trusted OS can subscribe to its event queue if it wishes to do so.

### 3.4.1 State Table

***Note:*** This state table will be defined in [TEE Core API] v1.2 and later.

The peripheral state table for the TEE pseudo-peripheral SHALL contain the values listed in Table 3-6.

**Table 3-6: `TEE_PERIPHERAL_OS` State Table Values**

| TEE_PeripheralValueType.id | TEE_PeripheralValueType.u |
|---|---|
| `TEE_PERIPHERAL_STATE_NAME` | `"TEE"` |
| `TEE_PERIPHERAL_STATE_FLAGS` | `TEE_PERIPHERAL_FLAG_EVENT_SOURCE` |
| `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` | `false` |

### 3.4.2 Events

The TEE pseudo-peripheral, when opened, SHALL return a `TEE_PeripheralDescriptor` which SHALL contain a valid `TEE_EventSourceHandle` and an invalid `TEE_PeripheralHandle` because it acts only as an event source.

The TEE pseudo-peripheral can act as a source for the event types listed in section 4.5.9.

## 3.5 Security Indicator

The security indicator is a specific indication that associated display and input peripherals can be considered trusted by the user, i.e. they are controlled by the TEE and isolated from the REE and even the TAs. It can be either a hardware controlled security indication such as an LED state or the use of an area of display under permanent TEE control.

It is highly recommended that the security indicator be managed directly by the TEE itself. The TEE SHOULD by default provide a security indicator when a TUI is displayed. Otherwise, the property value `gpd.tee.tui.low.securityIndicator` is set to `false` and the functionality of the security indicator SHALL be provided by the TA.

If the TEE provides the security indicator and the TA set the `TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR` flag when opening the TUI session, the TEE SHALL turn the indicator on when the TA first updates the display using the `TEE_TUI_BlitDisplaySurface` function and turn it off when the session is terminated, whether due to a TA action, a fatal event, or a timeout.

It is the duty of the TA to determine that the TEE controls sufficient peripherals to deliver the security levels it requires and only request the security indicator in this case.

If a TEE supports multiple displays – that is, if `gpd.tee.tui.low.simultaneousSecureDisplays` is set to `true` – then the TEE SHALL provide a separate indicator for each display.

If a single TEE sourced indicator is available and requested by the TA:

- The indicator SHALL only be on whenever a single session is active on all available displays.

If separate TEE sourced indicators are available and requested by the TA:

- The indicator for a given display SHALL be on only if a session is active on that display.

***Note:*** [TEE TUI] permits the use of an image known only to the TEE and displayed as part of the TUI as a security indicator. When using the low-level API defined in this specification, the entire display is under the control of the TA and therefore this mechanism is not considered. The value of the `gpd.tee.tui.low.securityIndicator` parameter may therefore differ from the `gpd.tee.tui.securityIndicator` parameter.

## 3.6 TUI Session

When a TA requests an associated display and input peripherals to be used by the TUI, that TA's access to those UI resources SHALL be exclusive; this means that it is not possible to display several TUIs simultaneously on the same display. A session mechanism permits a TA to reserve exclusive access to TUI resources. This ownership is limited by overriding events, as described in section 3.8.

The TA SHOULD take control of the input and output of the UI only for the minimum time required to carry out the interaction that needs to be secure.

## 3.7 Image Formats

This specification supports images in raw RGB and in the Portable Network Graphics (PNG) format [ISO 15948].

While an implementation might support the full feature set of the PNG standard, in this specification the mandatory support is reduced. At least the following SHALL be supported:

- Two color types:

  Grayscale (0) up to 8 bits depth

  Truecolor (2) up to 24 bits

- Interlacing method 0 (no interlacing)
- Ancillary chunks are ignored

## 3.8 Power and OS Events Management

TUI displays often display critical and sensitive data and operations that need to be immediate and exclusive. Trusted Applications must provide an experience that makes the user confident of the data displayed and entered. On the other hand, some events that occur on the platform in the REE, such as those related to power management or incoming calls, are also critical. This section clarifies the expected behavior of the TUI session in such a situation.

When the following power management events occur during a TUI session, the implementation SHALL cause the TUI session to terminate:

- Device Reset event
- Device Turn Off event
- Sleep mode Turn On event

If the TEE is processing a TUI Low-level API function call, the call will fail and return the error TEE_ERROR_EXTERNAL_CANCEL. The TEE SHOULD also place a TEE_EVENT_TUI_REE_CANCEL event onto any open event queue. Because the session is now terminated, any subsequent TUI Low-level API call requiring a TUI session will fail with the error TEE_ERROR_BAD_STATE until a new TUI session is opened.

When an OS specific event occurs during a TUI session, the TUI session MAY terminate. Typical OS specific events are incoming calls, calendar events, email notifications, etc. The choice to terminate a TUI session on a given OS specific event is implementation specific. If the TUI session terminates, the TEE SHOULD also place a TEE_EVENT_TUI_REE_DISPLAYSTOPPED event onto any open event queue.

When a TUI session is terminated, the TUI SHALL disappear from the display to make sure that the user is not confused and the control of the display is given back to the REE. It is acceptable for the TEE TUI to display a warning that the trusted display mode is being left before handing control of the display back to the REE.

## 3.9 Display Orientation

By default, this specification allows display of a screen in a fixed manner, either vertically or horizontally. An implementation SHALL support one of the two orientations and MAY support both.

Knowledge of the current orientation is not critical and can be considered as informative. It is not supplied by this API and MAY be obtained by the Client Application of the TA that wishes to display a TUI screen or may be supplied by the TEE.

Changes of orientation may be reported as events generated by the REE or TEE. The ability to generate these events relies on access to the accelerometer, which may not be available.

## 3.10 Specification Version Number Property

This specification defines a TEE property containing the version number of the specification that the implementation conforms to. The property can be retrieved using the normal Property Access Functions defined in [TEE Core API]. The property SHALL be named "`gpd.tee.tui.low.version`" and SHALL be of integer type with the interpretation given below.

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

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

**Table 3-7: Specification Version Number Property – 32-bit Integer Structure**

| Bits [24-31] (MSB) | Bits [16-23] | Bits [8-15] | Bits [0-7] (LSB) |
|---|---|---|---|
| Major version number of the specification | Minor version number of the specification | Maintenance version number of the specification | Reserved for future use. Currently SHALL be zero. |

So, for example:

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

This places the following requirement on the version numbering:

- No specification can have a Major or Minor or Maintenance version number greater than 255.

## 3.11 Structure Versions

It is expected that TAs will be loaded onto devices over the air using the TEE Management Framework ([TEE Mgmt Fmwk]), so it will be useful to be able to write a TA to work with different versions of the API. Therefore, each structure defined in this specification that the implementation can return has a version field and a union of the different versions. As this is the first release, all structures currently only have one member in this union. Each function that returns a structure enables the TA to specify the version of the structure it wants.

An implementation of this API should be able to issue the current version of a structure and such older versions as are defined by that version of the specification and any configuration documents.

A TA should check the version of the API at start up.

If the TEE implements an earlier version, the TA should determine whether it can work with the versions of structures defined in that version of the specification. If it cannot, it should exit gracefully.

If the TA requests an old version, the API should return this version if it can or else return an error if the version is no longer supported.

If the TA requests a version that is higher than that supported by the API, the API should return the most recent version it supports.

The TA should always check the version of the structure returned before attempting to use it.

**Table 3-8: Structure Versions**

| Structure | Version in API 1.0 |
|---|---|
| `TEE_Event` | 1 |
| `TEE_Event_TUI_Button` | 1 |
| `TEE_Event_TUI_Keyboard` | 1 |
| `TEE_Event_TUI_REE` | 1 |
| `TEE_Event_TUI_TEE` | 1 |
| `TEE_Event_TUI_Touch` | 1 |
| `TEE_Peripheral` | 1 |
| `TEE_PeripheralDescriptor` | 1 |
| `TEE_TUIDisplayInfo` | 1 |
| `TEE_TUISurfaceBuffer` | 1 |
| `TEE_TUISurfaceInfo` | 1 |

# 4 Trusted User Interface Low-level API

## 4.1 Implementation Properties

The following table defines the implementation properties regarding the TEE TUI Low-level API.

**Table 4-1: Implementation Properties**

| Property Name | Type | Meaning |
|---|---|---|
| `gpd.tee.event.maxSources` | Integer | The maximum number of secure event sources the implementation can support. |
| `gpd.tee.tui.low.securityIndicator` | Boolean | If `true`, the device has a separate security indicator managed by the TEE.<br>If `false`, a security indicator SHALL be managed by the TA itself. |
| `gpd.tee.tui.low.session.timeout` | Integer | Duration of the timeout associated with a TUI session in milliseconds. Typical value is around 10 seconds.<br>This is separate from the timeout used in the event loop (e.g. in `TEE_Event_OpenQueue`, described in section 4.6.16).<br>The TUI session SHOULD close if there is no request to update the display within this period, or if the TA, having received the event `TEE_EVENT_TUI_REE_CANCEL`, has not closed the session within this period. |
| `gpd.tee.tui.low.numDisplays` | Integer | The maximum number of secure displays available to this TEE. |
| `gpd.tee.tui.low.version` | Integer | The version of the TEE TUI Low-level API specification supported by this implementation; see section 3.10. |
| `gpd.tee.tui.low.`<br>`simultaneousSecureDisplays` | Boolean | If `true`, the TEE permits simultaneous sessions to open secure output on different displays.<br>If `false`, the Trusted OS permits only a single secure display at a time. |

## 4.2 Header File

The header file for the TEE TUI Low-level API SHALL have the name "`tee_tui_low_api.h`".

```
#include "tee_tui_low_api.h"
```

### 4.2.1 API Version

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

```
#define TEE_TUI_LOW_MAJOR_VERSION ([Major version number])
#define TEE_TUI_LOW_MINOR_VERSION ([Minor version number])
#define TEE_TUI_LOW_MAINTENANCE_VERSION ([Maintenance version number])
#define TEE_TUI_LOW_VERSION (TEE_TUI_LOW_MAJOR_VERSION << 24) +
                            (TEE_TUI_LOW_MINOR_VERSION << 16) +
                            (TEE_TUI_LOW_MAINTENANCE_VERSION << 8)
```

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

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

`TEE_TUI_LOW_MAJOR_VERSION` indicates the major version number of the TEE TUI Low-level API. It SHALL be set to the major version number of this specification.

`TEE_TUI_LOW_MINOR_VERSION` indicates the minor version number of the TEE TUI Low-level API. It SHALL be set to the minor version number of this specification. If the minor version is zero, then one zero shall be present.

`TEE_TUI_LOW_MAINTENANCE_VERSION` indicates the maintenance version number of the TEE TUI Low-level API. It SHALL be set to the maintenance version number of this specification. If the maintenance version is zero, then one zero shall be present.

The definitions of "Major Version", "Minor Version", and "Maintenance Version" in the version number of this specification are determined as defined in the GlobalPlatform Document Management Guide ([Doc Mgmt]). In particular, the value of `TEE_TUI_LOW_MAINTENANCE_VERSION` SHALL be zero if it is not already defined as part of the version number of this document. The "Draft Revision" number SHALL NOT be provided as an API version indication.

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

```
#define TEE_TUI_LOW_[Major version number]_[Minor version number]
```

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

```
#define TEE_TUI_LOW_[Major version number]_[Minor version number]_[Maintenance
version number]
```

Some examples of version definitions:

For GlobalPlatform TEE TUI Low-level API Specification v1.3, these would be:

```
#define TEE_TUI_LOW_MAJOR_VERSION        (1)
#define TEE_TUI_LOW_MINOR_VERSION        (3)
#define TEE_TUI_LOW_MAINTENANCE_VERSION  (0)
#define TEE_TUI_LOW_1_3
```

And the value of `TEE_TUI_LOW_VERSION` would be `0x01030000`.

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

```
#define TEE_TUI_LOW_MAJOR_VERSION        (2)
#define TEE_TUI_LOW_MINOR_VERSION        (14)
#define TEE_TUI_LOW_MAINTENANCE_VERSION  (7)
#define TEE_TUI_LOW_2_14_7
```

And the value of `TEE_TUI_LOW_VERSION` would be `0x020E0700`.

### 4.2.2 Version Compatibility Definitions

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

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

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

```
#define TEE_TUI_LOW_REQUIRED_MAJOR_VERSION         (major)
#define TEE_TUI_LOW_REQUIRED_MINOR_VERSION         (minor)
#define TEE_TUI_LOW_REQUIRED_MAINTENANCE_VERSION   (maintenance)
```

The following rules govern the use of `TEE_TUI_LOW_REQUIRED_MAJOR_VERSION`, `TEE_TUI_LOW_REQUIRED_MINOR_VERSION`, and `TEE_TUI_LOW_REQUIRED_MAINTENANCE_VERSION` by TA implementers:

- If `TEE_TUI_LOW_REQUIRED_MAINTENANCE_VERSION` is defined by a TA, then `TEE_TUI_LOW_REQUIRED_MAJOR_VERSION` and `TEE_TUI_LOW_REQUIRED_MINOR_VERSION` SHALL also be defined by the TA.
- If `TEE_TUI_LOW_REQUIRED_MINOR_VERSION` is defined by a TA, then `TEE_TUI_LOW_REQUIRED_MAJOR_VERSION` SHALL also be defined by the TA.

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

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

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

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

If **none** of the definitions above is set, a Trusted OS SHALL select the most recent version of this specification that it supports, as defined in section 4.2.1.

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

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

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

As an example, consider a TA which requires strict compatibility with TEE Trusted User Interface Low-level API v1.0:

```
#define TEE_TUI_LOW_REQUIRED_MAJOR_VERSION          (1)
#define TEE_TUI_LOW_REQUIRED_MINOR_VERSION          (0)
#define TEE_TUI_LOW_REQUIRED_MAINTENANCE_VERSION    (0)
```

Due to the semantics of the C preprocessor, the above definitions SHALL be defined before the main body of definitions in "tee_tui_low_api.h" is processed. The mechanism by which this occurs is out of scope of this document.

## 4.3 Data Constants

### 4.3.1 Return Codes

In addition to the return codes specified in [TEE Core API], new ones are used in this specification.

**Table 4-2: Return Codes**

| Constant Name | Value |
|---|---|
| TEE_ERROR_TIMEOUT | 0xffff3001 |
| TEE_ERROR_OLD_VERSION | 0xf0100005 |

***Note:*** The value of TEE_ERROR_OLD_VERSION is taken from the TEE Internal Core API numbering range, as it will be defined in [TEE Core API] v1.2 and later.

***Note:*** TEE_ERROR_TIMEOUT was defined in [TEE Core API] v1.0, deleted in v1.1, and will be restored in v1.2.

### 4.3.2 Maximum Sizes

***Note:*** These maximum payload sizes will be defined in [TEE Core API] v1.2 and later.

Table 4-3 defines the maximum size of structure payloads.

If a specification supported by a given Trusted OS requires a larger maximum size for any of the values in Table 4-3, the implementation SHOULD use the larger size.

**Table 4-3: Maximum Sizes of Structure Payloads**

| Constant Name | Value |
|---|---|
| TEE_MAX_EVENT_PAYLOAD_SIZE | 32 bytes |

### 4.3.3 TEE_EVENT_TYPE

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

`TEE_EVENT_TYPE` is a value indicating the source of an event.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_EVENT_TYPE;
#endif
```

To distinguish the event types defined in various specifications:

- GlobalPlatform event types SHALL have nibble 8 (the high nibble) = `0`, and SHALL include the specification number as a 3-digit BCD (Binary Coded Decimal) value in nibbles 7 through 5.

  For example, GPD_SPE_**123** may define specification unique event type codes `0x01230000` to `0x0123ffff`.

  All event types defined in this specification have the high word set to `0x0055`.

  Events from sources defined in TEE TUI Extension: Biometrics API ([TEE TUI Bio]) have the event type `0x00420000`.

- Event types created by external bodies SHALL have nibble 8 = 1.
- Implementation defined event types SHALL have nibble 8 = 2.

Table 4-4 lists event types defined to date.

Implementations may not support all event types; however, it is recommended that TA developers define event handlers for all of the events defined on the peripherals they support. To determine which event types are supported by a particular peripheral, the developer can consult the documentation for that peripheral.

**Table 4-4: TEE_EVENT_TYPE Values**

| Constant Name | Value |
|---|---|
| Reserved for future use | 0x00000000 – 0x0000ffff |
| Reserved for GlobalPlatform TEE specifications numbered 001 - 009 | 0x00010000 – 0x0009ffff |
| TEE_EVENT_TYPE_ALL | 0x00100000 |
| Reserved for [TEE Core API] | 0x00100001 – 0x0010fffe |
| TEE_EVENT_TYPE_ILLEGAL_VALUE | 0x0010ffff |
| Reserved for GlobalPlatform TEE specifications numbered 011 - 041 | 0x00110000 – 0x0041ffff |
| TEE_EVENT_TYPE_BIO<br>Defined in [TEE TUI Bio]; if the Biometrics API is not implemented, reserved. | 0x00420000 |
| Reserved for [TEE TUI Bio] | 0x00420001 – 0x0042ffff |
| Reserved for GlobalPlatform TEE specifications numbered 043 – 054 | 0x00430000 – 0x0054ffff |
| TEE_EVENT_TYPE_TUI_ALL | 0x00550000 |
| TEE_EVENT_TYPE_TUI_BUTTON | 0x00550001 |
| TEE_EVENT_TYPE_TUI_KEYBOARD | 0x00550002 |
| TEE_EVENT_TYPE_TUI_REE | 0x00550003 |
| TEE_EVENT_TYPE_TUI_TOUCH | 0x00550004 |
| Reserved for future versions of this specification | 0x00550005 – 0x0055ffff |
| Reserved for GlobalPlatform TEE specifications numbered 056 – 999 | 0x00560000 – 0x0999ffff |
| Reserved for external bodies; number space managed by GlobalPlatform | 0x10000000 – 0x1fffffff |
| Implementation Defined | 0x20000000 – 0x2fffffff |
| Reserved for future use | 0x30000000 – 0xffffffff |

TEE_EVENT_TYPE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.4 TEE_EVENT_TUI_BUTTON_ACTION

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_BUTTON_ACTION is a value indicating the action of a button.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_EVENT_TUI_BUTTON_ACTION;
#endif
```

**Table 4-5: TEE_EVENT_TUI_BUTTON_ACTION Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_BUTTON_UP | 0x00000000 |
| TEE_EVENT_TUI_BUTTON_DOWN | 0x00000001 |
| Reserved | 0x00000002 – 0x7ffffffe |
| TEE_EVENT_TUI_BUTTON_ACTION_ILLEGAL_VALUE | 0x7fffffff |
| Implementation Defined | 0x80000000 – 0xffffffff |

TEE_EVENT_TUI_BUTTON_ACTION_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.5 TEE_EVENT_TUI_BUTTON_TYPE

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_BUTTON_TYPE is a value indicating the functionality attributed to a button.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_EVENT_TUI_BUTTON_TYPE;
#endif
```

**Table 4-6: TEE_EVENT_TUI_BUTTON_TYPE Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_BUTTON_BACK | 0x00000000 |
| TEE_EVENT_TUI_BUTTON_HOME | 0x00000001 |
| TEE_EVENT_TUI_BUTTON_MENU | 0x00000002 |
| TEE_EVENT_TUI_BUTTON_POWER | 0x00000003 |
| TEE_EVENT_TUI_BUTTON_VOLUMEDOWN | 0x00000004 |
| TEE_EVENT_TUI_BUTTON_VOLUMEUP | 0x00000005 |
| Reserved | 0x00000006 – 0x7ffffffe |
| TEE_EVENT_TUI_BUTTON_TYPE_ILLEGAL_VALUE | 0x7fffffff |
| Implementation Defined | 0x80000000 – 0xffffffff |

TEE_EVENT_TUI_BUTTON_TYPE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.6 TEE_EVENT_TUI_KEY_ACTION

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_KEY_ACTION is a value defining the action of a key.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint8_t TEE_EVENT_TUI_KEY_ACTION;
#endif
```

**Table 4-7: TEE_EVENT_TUI_KEY_ACTION Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_KEY_ACTION_UP | 0x00 |
| TEE_EVENT_TUI_KEY_ACTION_DOWN | 0x01 |
| Reserved | 0x02 – 0x7e |
| TEE_EVENT_TUI_KEY_ACTION_ILLEGAL_VALUE | 0x7f |
| Implementation Defined | 0x80 – 0xff |

TEE_EVENT_TUI_KEY_ACTION_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.7 TEE_EVENT_TUI_REE_TYPE

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_REE_TYPE is a value indicating the event being passed from the REE. The meaning of these constants is described in section 4.5.10.3.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint16_t TEE_EVENT_TUI_REE_TYPE;
#endif
```

**Table 4-8: TEE_EVENT_TUI_REE_TYPE Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_REE_DISPLAYSTOPPED | 0x0000 |
| TEE_EVENT_TUI_REE_CANCEL | 0x0001 |
| TEE_EVENT_TUI_REE_ROT90 | 0x0002 |
| TEE_EVENT_TUI_REE_ROT180 | 0x0003 |
| TEE_EVENT_TUI_REE_ROT270 | 0x0004 |
| Reserved | 0x0005 – 0x7ffe |
| TEE_EVENT_TUI_REE_ILLEGAL_VALUE | 0x7fff |
| Implementation Defined | 0x8000 – 0xffff |

Used for events that originate in the REE and are passed by the TEE to the Trusted Application via the event queue of the TEE_PERIPHERAL_OS peripheral.

TEE_EVENT_TUI_REE_DISPLAYSTOPPED indicates that the device has taken back control of the display; for instance, because a call is being received.

TEE_EVENT_TUI_REE_CANCEL indicates that some process has sent a cancellation; for instance, because the battery is low.

All rotations are clockwise and relative to the last known orientation.

A TEE_EVENT_TUI_REE_ROT*xxx* event should be used when the source of the event comes from an untrusted peripheral; that is, one under the control of the REE.

A TEE_EVENT_TUI_TEE_ROT*xxx* event (see TEE_EVENT_TUI_TEE_TYPE in section 4.3.8) should be used when the source of the event comes from a trusted peripheral; that is, one under direct control of the TEE.

These events are used to inform the TA, which must then update the image based on the event.

TEE_EVENT_TUI_REE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.8 TEE_EVENT_TUI_TEE_TYPE

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_TEE_TYPE is a value indicating the event being passed from the TEE.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint16_t TEE_EVENT_TUI_TEE_TYPE;
#endif
```

**Table 4-9: TEE_EVENT_TUI_TEE_TYPE Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_TEE_DISPLAYSTOPPED | 0x0000 |
| TEE_EVENT_TUI_TEE_CANCEL | 0x0001 |
| TEE_EVENT_TUI_TEE_ROT90 | 0x0002 |
| TEE_EVENT_TUI_TEE_ROT180 | 0x0003 |
| TEE_EVENT_TUI_TEE_ROT270 | 0x0004 |
| Reserved | 0x0005 – 0x7ffe |
| TEE_EVENT_TUI_TEE_ILLEGAL_VALUE | 0x7fff |
| Implementation Defined | 0x8000 – 0xffff |

Used for events that originate in the TEE and are passed by the TEE to the Trusted Application via the event queue of the TEE_PERIPHERAL_OS peripheral.

TEE_EVENT_TUI_TEE_DISPLAYSTOPPED indicates that the device has taken back control of the display; for instance, because a call is being received.

TEE_EVENT_TUI_TEE_CANCEL indicates that some process has sent a cancellation; for instance, because the battery is low.

All rotations are clockwise and relative to the last known orientation.

A TEE_EVENT_TUI_TEE_ROT*xxx* event should be used when the source of the event comes from a trusted peripheral; that is, one under direct control of the TEE.

A TEE_EVENT_TUI_REE_ROT*xxx* event (see TEE_EVENT_TUI_REE_TYPE in section 4.3.7) should be used when the source of the event comes from an untrusted peripheral; that is, one under the control of the REE.

These events are used to inform the TA, which must then update the image based on the event.

TEE_EVENT_TUI_TEE_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.9 TEE_EVENT_TUI_TOUCH_ACTION

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_EVENT_TUI_TOUCH_ACTION is a value indicating the event being passed from the TEE.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint16_t TEE_EVENT_TUI_TOUCH_ACTION;
#endif
```

**Table 4-10: TEE_EVENT_TUI_TOUCH_ACTION Values**

| Constant Name | Value |
|---|---|
| TEE_EVENT_TUI_TOUCH_ACTION_DOWN | 0x0000 |
| TEE_EVENT_TUI_TOUCH_ACTION_MOVE | 0x0001 |
| TEE_EVENT_TUI_TOUCH_ACTION_UP | 0x0002 |
| Reserved | 0x0003 – 0x7ffe |
| TEE_EVENT_TUI_TOUCH_ACTION_ILLEGAL_VALUE | 0x7fff |
| Implementation Defined | 0x8000 – 0xffff |

TEE_EVENT_TUI_TOUCH_ACTION_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.10 TEE_PERIPHERAL_TYPE

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

TEE_PERIPHERAL_TYPE is a value used to identify a peripheral attached to the device.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_PERIPHERAL_TYPE;
#endif
```

The TEE_Peripheral_GetPeripherals function lists all the peripherals known to the TEE.

**Table 4-11: TEE_PERIPHERAL_TYPE Values**

| Constant Name | Value |
|---|---|
| Reserved | 0x00000000 |
| TEE_PERIPHERAL_OS | 0x00000001 |
| TEE_PERIPHERAL_CAMERA | 0x00000002 |
| TEE_PERIPHERAL_MICROPHONE | 0x00000003 |
| TEE_PERIPHERAL_ACCELEROMETER | 0x00000004 |
| TEE_PERIPHERAL_NFC | 0x00000005 |
| TEE_PERIPHERAL_BLUETOOTH | 0x00000006 |
| TEE_PERIPHERAL_USB | 0x00000007 |
| TEE_PERIPHERAL_FINGERPRINT | 0x00000008 |
| TEE_PERIPHERAL_KEYBOARD | 0x00000009 |
| TEE_PERIPHERAL_TOUCH | 0x0000000A |
| TEE_PERIPHERAL_BIO | 0x0000000B |
| Reserved for GlobalPlatform specifications | 0x0000000C – 0x3fffffff |
| Reserved for other Specification Development Organizations (SDOs) under Liaison Statement (LS) | 0x40000000 – 0x7ffffffe |
| TEE_PERIPHERAL_ILLEGAL_VALUE | 0x7fffffff |
| Implementation Defined | 0x80000000 – 0xffffffff |

TEE_PERIPHERAL_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.11 TEE_PERIPHERAL_FLAGS

***Note:*** These values will be defined in [TEE Core API] v1.2 and later.

**Table 4-12: `TEE_PERIPHERAL_FLAGS` Values**

| Constant Name | Value | Meaning |
|---|---|---|
| `TEE_PERIPHERAL_FLAG_REE_CONTROLLED` | `0x00000000` | The Trusted OS cannot control this peripheral.<br>All events are passed directly to the REE even during TUI sessions. |
| `TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE` | `0x00000001` | The Trusted OS can control this peripheral.<br>Events SHALL NOT be passed to the REE during TUI sessions. |
| `TEE_PERIPHERAL_FLAG_EVENT_SOURCE` | `0x00000002` | The TEE can parse the events generated by this peripheral.<br>The peripheral can be attached to an event queue. |
| `TEE_PERIPHERAL_FLAG_LOCKED` | `0x00000004` | This peripheral has been locked for access by **a** TA or the REE. |
| `TEE_PERIPHERAL_FLAG_OWNED` | `0x00000008` | This peripheral has been locked for access by **this** TA instance. |

The flags `TEE_PERIPHERAL_FLAG_REE_CONTROLLED` and `TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE` are mutually exclusive.

If an event source has the `TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE` flag but not the `TEE_PERIPHERAL_FLAG_EVENT_SOURCE` flag, the TEE can control the source, but not understand it. Any events generated while the TEE has control of the source SHALL be dropped.

### 4.3.12 TEE_PeripheralStateId Values

***Note:*** These values will be defined in [TEE Core API] v1.2 and later.

`TEE_PeripheralState` instances are used to provide information about peripherals to a TA. The following field values, which represent legal values of type `TEE_PeripheralStateId` which can be used to identify specific peripheral state items, are defined in this specification. Other specifications may define additional values for `TEE_PeripheralStateId`.

**Table 4-13: `TEE_PeripheralStateId` Values**

| Constant Name | Value |
|---|---|
| Reserved | `0x00000000` |
| `TEE_PERIPHERAL_STATE_NAME` | `0x00000001` |
| `TEE_PERIPHERAL_STATE_FW_INFO` | `0x00000002` |
| `TEE_PERIPHERAL_STATE_MANUFACTURER` | `0x00000003` |
| `TEE_PERIPHERAL_STATE_FLAGS` | `0x00000004` |
| Reserved for GlobalPlatform specifications | `0x00000005 – 0x3fffffff` |
| Reserved for other SDOs under LS | `0x40000000 – 0x7ffffffe` |
| `TEE_PERIPHERAL_STATE_ILLEGAL_VALUE` | `0x7fffffff` |
| `Implementation Defined` | `0x80000000 – 0xffffffff` |

`TEE_PERIPHERAL_STATE_ILLEGAL_VALUE` is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

#### 4.3.12.1 TEE_PERIPHERAL_TUI_BUTTON State Table

The peripheral state table for a Button peripheral SHALL contain the following values:

**Table 4-14: `TEE_PERIPHERAL_TUI_BUTTON` State Table**

| TEE_PeripheralValueType.id | TEE_PeripheralValueType.u |
|---|---|
| `TEE_PERIPHERAL_STATE_NAME` | “Button” |
| `TEE_PERIPHERAL_STATE_TUI_BUTTON` | A `TEE_EVENT_TUI_BUTTON_TYPE` |

The `TEE_PERIPHERAL_TUI_BUTTON` structure defines the hardware buttons that can be reported by the TEE. One structure is returned for each button under the control of the TEE.

During a TUI session, if the user presses this button, no information is returned to the REE.

There may be buttons that are not under the control of the TEE. If the user presses these buttons, it is reported to the REE. This does not generate a TUI event; however, the REE may generate a `TEE_Event_TUI_REE`.

#### 4.3.12.2 TEE_PERIPHERAL_TUI_KEYBOARD State Table

The peripheral state table for a Keyboard peripheral SHALL contain the following values:

**Table 4-15: TEE_PERIPHERAL_TUI_KEYBOARD State Table**

| TEE_PeripheralValueType.id | TEE_PeripheralValueType.u |
|---|---|
| TEE_PERIPHERAL_STATE_NAME | "Keyboard" |
| TEE_PERIPHERAL_STATE_TUI_COUNTRY | countryCode |

- countryCode: The language the keyboard is localized for is described using the numbering defined in the USB HID keyboard country code enumeration [USB Country]. A TA could use this to determine the language the keyboard is localized for. If the keyboard is not localized, it returns a value of 0 (Not Supported).

#### 4.3.12.3 TEE_PERIPHERAL_TUI_TOUCH State Table

The peripheral state table for a Touch sensor peripheral SHALL contain the following values:

**Table 4-16: TEE_PERIPHERAL_TUI_TOUCH State Table**

| TEE_PeripheralValueType.id | TEE_PeripheralValueType.u |
|---|---|
| TEE_PERIPHERAL_STATE_NAME | "Touch" |
| TEE_PERIPHERAL_STATE_TUI_FINGERS | uint32_t |
| TEE_PERIPHERAL_STATE_TUI_CAN_DEBOUNCE | Boolean |
| TEE_PERIPHERAL_STATE_TUI_DEBOUNCE_STATE | Boolean |

- TEE_PERIPHERAL_STATE_TUI_FINGERS: The number of simultaneous fingers that this display supports. This SHALL be 1 unless TEE_TUI_DISPLAY_INFO_FLAG_MULTITOUCH is set in the TEE_TUIDisplayInfo structure for this display.
- TEE_PERIPHERAL_STATE_TUI_CAN_DEBOUNCE: This is set to true if the implementation can debounce inputs.
- TEE_PERIPHERAL_STATE_TUI_DEBOUNCE_STATE: This is a settable property. It is usually set to the value of TEE_PERIPHERAL_STATE_TUI_CAN_DEBOUNCE. The TA can set this to false to receive raw input from the touch source.

### 4.3.13 TEE_TUI_DISPLAY_INFO_FLAGS

**Table 4-17: TEE_TUI_DISPLAY_INFO_FLAGS Values**

| Constant Name | Value | Meaning |
|---|---|---|
| TEE_TUI_DISPLAY_INFO_FLAG_NONESET | 0x00000000 | No flags set. |
| TEE_TUI_DISPLAY_INFO_FLAG_HASTOUCH | 0x00000001 | The TEE can return secure touch events generated by this display.<br>Therefore, there will be an event source associated with this display. |
| TEE_TUI_DISPLAY_INFO_FLAG_HASPRESSURE | 0x00000002 | The display returns pressure information about touches.<br>If the display CANNOT return pressure information, then pressure values SHALL be set to 127 for DOWN and MOVE events and 0 for UP events. |
| TEE_TUI_DISPLAY_INFO_FLAG_CANDEBOUNCE | 0x00000004 | The TEE can attempt to debounce touch events from this display. |
| TEE_TUI_DISPLAY_INFO_FLAG_MULTITOUCH | 0x00000008 | The display can generate touch events for multiple fingers at the same time. The number of fingers supported is reported in the peripheral state table. |

### 4.3.14 TEE_TUI_INIT_SESSION_LOW_FLAGS

**Table 4-18: TEE_TUI_INIT_SESSION_LOW_FLAGS Values**

| Constant Name | Value | Meaning |
|---|---|---|
| TEE_TUI_INIT_SESSION_LOW_FLAG_NONESET | 0x00000000 | No flags set. |
| TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR | 0x00000001 | The TA requires the TEE to set the security indicator. |

This flag enables the TA to specify whether it wants the TEE to set the security indicator.

The TA should only request the security indicator if the TEE can control all the peripherals that the TA believe might offer a side channel and the TA has been able to open all these peripherals.

The TA SHALL NOT set the TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR flag if the TEE does not offer a security indicator; that is, if gpd.tee.tui.low.securityIndicator is false. If a TA does this, the TEE will panic.

### 4.3.15 TEE_TUI_OPERATION

**Table 4-19: `TEE_TUI_OPERATION` Values**

| Constant Name | Value | Operation |
|---|---|---|
| `TEE_TUI_OPERATION_NULL` | 0x00000000 | No transformation |
| `TEE_TUI_OPERATION_ROTATE90` | 0x00000001 | Rotate the image 90 degrees clockwise |
| `TEE_TUI_OPERATION_ROTATE180` | 0x00000002 | Rotate the image 180 degrees |
| `TEE_TUI_OPERATION_ROTATE270` | 0x00000003 | Rotate the image 270 degrees clockwise; that is, 90 degrees counterclockwise |
| `TEE_TUI_OPERATION_FLIPH` | 0x00000004 | Flip the image about its horizontal axis |
| `TEE_TUI_OPERATION_FLIPV` | 0x00000008 | Flip the image about its vertical axis |
| `TEE_TUI_OPERATION_INVERTRED` | 0x00000010 | Invert the value of the Red color values |
| `TEE_TUI_OPERATION_INVERTGREEN` | 0x00000020 | Invert the value of the Green color values |
| `TEE_TUI_OPERATION_INVERTBLUE` | 0x00000040 | Invert the value of the Blue color values |
| `TEE_TUI_OPERATION_INVERTALPHA` | 0x00000080 | Invert the value of the Alpha channel |
| `TEE_TUI_OPERATION_ILLEGAL_VALUE` | 0x80000000 | See below |
| Reserved | | The reserved bits are defined by the mask `0x8000ffff`. |
| Implementation Defined | | The implementation defined bits are defined by the mask `0x7fff0000`. |

`TEE_TUI_OPERATION_ILLEGAL_VALUE` is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.3.16 TEE_TUI_SURFACE_OPACITY

**Table 4-20: TEE_TUI_SURFACE_OPACITY Values**

| Constant Name | Operation (see also notes after table) | Value |
|---|---|---|
| TEE_TUI_SURFACE_OPACITY_OPAQUE | target.RGBA = source.RGBA | 0x00000001 |
| TEE_TUI_SURFACE_OPACITY_SOURCEALPHA | target.RGB = source.RGB * source.Alpha | 0x00000002 |
| TEE_TUI_SURFACE_OPACITY_TARGETALPHA | target.RGB = source.RGB * target.Alpha | 0x00000003 |
| TEE_TUI_SURFACE_OPACITY_TRANSPARENT | target.RGBA = target.RGBA + source.RGBA | 0x00000004 |
| TEE_TUI_SURFACE_OPACITY_BLENDSOURCE | target.RGBA = target.RGBA * source.Alpha + source.RGBA | 0x00000005 |
| TEE_TUI_SURFACE_OPACITY_BLENDTARGET | target.RGBA = target.RGBA + source.RGBA * target.Alpha | 0x00000006 |
| TEE_TUI_SURFACE_OPACITY_BLEND | target.RGBA = target.RGBA * source.Alpha + source.RGBA * target.Alpha | 0x00000007 |
| Reserved | | 0x00000008 – 0x7ffffffe |
| TEE_TUI_SURFACE_OPACITY_ILLEGAL_VALUE | See below. | 0x7fffffff |
| Implementation Defined | | 0x80000000 – 0xffffffff |

#### Notes

- Addition (+) is a blending operation.

  For each color component, the output component is (a+b)/2 using Integer Division; therefore, adding two identical colors results in an unchanged color.

  Adding black to a color makes it darker.

  Adding white to a color makes it lighter, but will not result in white.

- Multiplication (*) is a scaling operation.

  For each color component, the output is (a * b)/255 using Integer Division.

  Multiplying by zero gives zero.

  Multiplying by 255 leaves the value unchanged.

- The alpha blending operation is fixed as source over destination, as defined in Porter/Duff Compositing [Porter Duff].

TEE_TUI_SURFACE_OPACITY_ILLEGAL_VALUE is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

## 4.4 Data Types

This specification uses the data types defined in [TEE Core API] section 3.2, plus those listed below.

***Note:*** These additional data types will be defined in [TEE Core API] v1.2 and later.

- `size_t`: The unsigned integer type of the result of the sizeof operator.
- `uint64_t`: Unsigned 64-bit integer

In the event of any difference between the definitions in this specification and those in the C99 standard (ISO/IEC 9899:1999 – [C99]), C99 shall prevail.

## 4.5 Data Structures

Implementations of [TEE Core API] that define the data structures required for processing events and peripherals will define the sentinel `TEE_CORE_API_EVENT`; therefore, implementations of this API SHALL check for this value before defining the structure.

Several data structures defined in this specification are versioned. This allows a TA written against an earlier version of this API than that implemented by a TEE to request the version of the structure it understands.

### 4.5.1 TEE_Peripheral

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

`TEE_Peripheral` is a structure used to provide information about a single peripheral to a TA.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
   typedef struct
   {
       uint32_t               version;
       union  {
              TEE_Peripheral_V1 v1;
              } u;

   } TEE_Peripheral;

   typedef struct
   {
       TEE_PERIPHERAL_TYPE    periphType;
       TEE_PeripheralId       id;
   } TEE_Peripheral_V1;
#endif
```

- `version`: The version of the structure – currently always 1.
- `periphType`: The type of the peripheral.
- `id`: A unique identifier for a given peripheral on a TEE.

A TEE may have more than one peripheral of the same TEE_PERIPHERAL_TYPE. The id parameter provides a TEE-unique identifier for a specific peripheral, and the implementation SHOULD provide further information about the specific peripheral instance in the TEE_PERIPHERAL_STATE_NAME field described in section 3.3.1.

The id parameter for a given peripheral SHOULD NOT change between Trusted OS version updates on a device.

### 4.5.2 TEE_PeripheralDescriptor

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

TEE_PeripheralDescriptor is a structure collecting the information required to access a peripheral.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
    typedef struct
    {
        uint32_t             version;
        union  {
                TEE_PeripheralDescriptor_V1 v1;
                } u;
    } TEE_PeripheralDescriptor

  typedef struct
  {
    TEE_PeripheralId      id;
    TEE_PeripheralHandle  peripheralHandle;
    TEE_EventSourceHandle eventSourceHandle;
  } TEE_PeripheralDescriptor_V1;
#endif
```

The structure fields have the following meanings:

- The version field identifies the version of the TEE_PeripheralDescriptor structure. In this version of the specification it SHALL be set to 1.
- The id field contains a unique identifier for the peripheral with which this TEE_PeripheralDescriptor instance is associated.
- The peripheralHandle field contains a TEE_PeripheralHandle which, if valid, enables an owning TA to perform API calls which might alter peripheral state.
- The eventSourceHandle field contains a TEE_EventSourceHandle which can be used to attach events generated by the peripheral to an event queue.

### 4.5.3 TEE_PeripheralHandle

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

A `TEE_PeripheralHandle` is an opaque handle used to manage direct access to a peripheral.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef struct __TEE_PeripheralHandle* TEE_PeripheralHandle;
#endif
```

TA implementations SHOULD NOT assume that the same `TEE_PeripheralHandle` will be returned for different sessions.

The value `TEE_INVALID_HANDLE` is used to indicate an invalid `TEE_PeripheralHandle`. All other values denote a valid `TEE_PeripheralHandle`.

### 4.5.4 TEE_PeripheralId

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

A `TEE_PeripheralId` is a `uint32_t`, used as a unique identifier for a peripheral on a given TEE.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_PeripheralId;
#endif
```

`TEE_PeripheralId` SHALL be unique on a given TEE, and SHALL be constant for a given peripheral between TEE reboots. If a peripheral is removed and reinserted, the same value of `TEE_PeripheralId` SHALL be associated with it.

### 4.5.5 TEE_PeripheralState

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

`TEE_PeripheralState` is a structure containing the current value of an individual peripheral state value on a given TEE.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef struct
  {
    uint32_t                version;
   TEE_PeripheralValueType    tag;
   TEE_PeripheralStateID      id;
   bool                     ro;
   union {
       uint64_t              uint64Val;
       uint32_t              uint32Val;
       uint16_t              uint16Val;
       uint8_t               uint8Val;
       bool                  boolVal;
       const char*           stringVal;
       } u;
  } TEE_PeripheralState;
#endif
```

The structure fields have the following meanings:

- The `version` field identifies the version of the `TEE_PeripheralState` structure. In this version of the specification it SHALL be set to 1.
- The `tag` field is a `TEE_PeripheralStateValueType` instance indicating which field in the union, `u`, should be accessed to obtain the correct configuration value.
- The `id` field is a unique identifier for this node in the peripheral configuration tree. It can be used in the set/get API calls to select a peripheral configuration value directly.
- The `ro` field is `true` if this configuration value cannot be updated by the calling TA. A TA SHOULD NOT call `TEE_PeripheralSetState` with a given `TEE_PeripheralStateId` if the `ro` field of the corresponding `TEE_PeripheralState` is `true`. An implementation MAY generate an error if this is not respected.
- The union field, `u`, contains fields representing the different data types which can be used to store peripheral configuration information.

A Trusted OS MAY indicate different `TEE_PeripheralState` information to different TAs on the system. Therefore a TA SHOULD NOT pass `TEE_PeripheralState` to another TA as the information it contains may not be valid for the other TA.

### 4.5.6 TEE_PeripheralStateId

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

A `TEE_PeripheralStateId` is an identifier for a peripheral state entry on a given TEE.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_PeripheralStateId;
#endif
```

Legal values in this specification for `TEE_PeripheralStateId` are listed in section 4.3.12. Further values may be defined in other specifications.

### 4.5.7 TEE_PeripheralValueType

***Note:*** This type will be defined in [TEE Core API] v1.2 and later.

`TEE_PeripheralValueType` indicates which of several types has been used to store the configuration information in a `TEE_PeripheralState.tag` field.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef uint32_t TEE_PeripheralValueType;
#endif
```

**Table 4-21: `TEE_PeripheralValueType` Values**

| Constant Name | Value |
|---|---|
| TEE_PERIPHERAL_VALUE_UINT64 | 0x00000000 |
| TEE_PERIPHERAL_VALUE_UINT32 | 0x00000001 |
| TEE_PERIPHERAL_VALUE_UINT16 | 0x00000002 |
| TEE_PERIPHERAL_VALUE_UINT8 | 0x00000003 |
| TEE_PERIPHERAL_VALUE_BOOL | 0x00000004 |
| TEE_PERIPHERAL_VALUE_STRING | 0x00000005 |
| Reserved | 0x00000006 – 0x7ffffffe |
| TEE_PERIPHERAL_VALUE_ILLEGAL_VALUE | 0x7fffffff |
| Implementation Defined | 0x80000000 – 0xffffffff |

`TEE_PERIPHERAL_VALUE_ILLEGAL_VALUE` is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.5.8 TEE_Event

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

`TEE_Event` is a container for events in the event loop.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
    typedef struct {
      uint32_t                  version;
      union {
             TEE_Event_V1 v1;
            } u;
   } TEE_Event;

   typedef struct {
      TEE_EVENT_TYPE            eventType;
      uint64_t                  timestamp;
      TEE_EventSourceHandle     eventSourceHandle;
      uint8_t                   payload[TEE_MAX_EVENT_PAYLOAD_SIZE];
    } TEE_Event_V1;
#endif
```

The `TEE_Event` structure holds an individual event; the payload holds the byteblock that depends on the type of the event:

- `version`: The version of the structure – currently always 1.
- `eventType`: A value identifying the type of event.
- `timestamp`: The time the event occurred given as milliseconds since the TEE was started. The value of timestamp is guaranteed to increase monotonically so that the ordering of events in time is guaranteed. A Trusted OS SHOULD use the same underlying source of time information as used for `TEE_GetSystemTime`, described in [TEE Core API].
- `eventSourceHandle`: The handle of the specific event source that created this event.
- `payload`: A block of `TEE_MAX_EVENT_PAYLOAD_SIZE` bytes containing a `TEE_Event_`*xxx_xxxx* structure of the appropriate type. If an event needs to refer to data larger than the `TEE_MAX_EVENT_PAYLOAD_SIZE`, it should use a pointer to an external byteblock. Trusted User Interface types, `TEE_Event_TUI_`*xxxx*, are defined in section 4.5.10, TEE_Event TUI Payloads. Other specifications will define further types. Any trailing bytes SHALL be zero.

In general, if an event cannot be sufficiently described within the constraints of the `payload` field of `TEE_MAX_EVENT_PAYLOAD_SIZE`, the contents of the field may be data structure containing handles or pointers to further structures that together fully describe the event.

### 4.5.9 Generic Payloads

This section describes a generic `payload` field of the `TEE_Event` structure.

#### 4.5.9.1 TEE_Event_AccessChange

***Note:*** This event will be defined in [TEE Core API] v1.2 and later.

This event is generated if the accessibility of a peripheral to this TA changes.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef struct {
    uint32_t         version;
    TEE_PeripheralId id;
    uint32_t         flags;
  } TEE_Event_AccessChange;
#endif
```

- `version`: The version of the structure – currently always 1.
- `id`: The `TEE_PeripheralId` for the peripheral for which the access change event was generated. This uniquely identifies the peripheral for which the access status has changed.
- `flags`: The new state of `TEE_PERIPHERAL_STATE_FLAGS`. For details of the legal values for this field, see the description of the `u.uint32Val` field in section 3.3.4.

This event SHALL be sent to all TAs which have registered to the `TEE_PERIPHERAL_OS` event queue when an access permission change occurs – including the TA which initiated the change.

A consequence of `TEE_Event_AccessChange` is that some of the peripheral state table information may change. As such, each TA instance SHOULD call `TEE_Peripheral_GetStateTable` to obtain fresh information when it receives this event.

### 4.5.10 TEE_Event TUI Payloads

This section describes the `payload` field of the `TEE_Event` structure for TUI specific events.

#### 4.5.10.1 TEE_Event_TUI_Button

```
typedef struct
{
    uint32_t                        version;
    union   {
            TEE_Event_TUI_Button_V1 v1;
            } u;
} TEE_Event_TUI_Button;

typedef struct
{
    TEE_EVENT_TUI_BUTTON_ACTION     action;
    TEE_EVENT_TUI_BUTTON_TYPE       button;
} TEE_Event_TUI_Button_V1;
```

`TEE_Event_TUI_Button` events are produced by user interaction with physical or virtual buttons provided by the device hardware.

- `version`: The version of the structure – currently always 1.
- `action`: Whether the button is pressed or released; see section 4.3.4.
- `button`: The button that generated the event; see section 4.3.5.

If a button is under the control of the TEE, pressing the button generates a `TEE_Event_TUI_Button` and the REE receives no information about the button being pressed.

If a button is under the control of the REE, there is no `TEE_Event_TUI_Button` but the REE may generate an event (`TEE_Event_TUI_REE` – see section 4.5.10.3).

#### 4.5.10.2 TEE_Event_TUI_Keyboard

```
typedef struct
{
    uint32_t                    version;
    union  {
           TEE_Event_TUI_Keyboard_V1   v1;
           } u;
} TEE_Event_TUI_Keyboard;

typedef struct
{
    TEE_EVENT_TUI_KEY_ACTION   action;
    uint32_t                   mod;
    uint32_t                   key;
} TEE_Event_TUI_Keyboard_V1;
```

`TEE_Event_TUI_Keyboard` events are produced by user interaction with a physical or virtual keyboard incorporated into the device and under the control of the TEE.

This is unrelated to the use of a virtual keyboard displayed on a touch screen created by the TA itself (input from which would be reported as coordinates through touch events) or of any external keyboard connected via the REE (which would be reported as REE events).

Keyboard presses are split into one DOWN and one UP event (corresponding to a given key being pressed and released, respectively). Each event has a bitmap of modifier keys (shift, control, etc.) indicating which of these were depressed at the time of the event.

The TEE must manage any stateful key indicators (such as caps lock, num lock, etc.) – events for these keys must still be sent to TAs. This avoids all TAs having to deal with reading and writing the status LEDs for such devices.

Discovering and remapping key presses to match the markings for a given physical keyboard layout is the responsibility of the TEE.

- `version`: The version of the structure – currently always 1.
- `action`: The action performed, Key Down or Key Up.
- `mod`: The modifier key bitmap as given in the USB HID device class definition [USB HID]. Note that on devices where there is no specific localization, `0` (Not Supported) is returned.
- `key`: The key identified with the values given in the USB HID Keyboard/Keypad page [USB Key].

#### 4.5.10.3 TEE_Event_TUI_REE

```
typedef struct
{
    uint32_t                  version;
    union {
          TEE_Event_TUI_REE_V1 v1;
          } u;
} TEE_Event_TUI_REE;

typedef struct
{
    TEE_EVENT_TUI_REE_TYPE     event;
} TEE_Event_TUI_REE_V1;
```

`TEE_Event_TUI_REE` events enable the TEE to pass events from outside the TEE to the calling TA. This standard defines a limited number of such events. `TEE_Event_TUI_REE` events may also be used for other events from other entities outside this TEE. In that case, a Trusted OS would have to define its own event types in the implementation defined range.

Events originating in the REE transit through the Trusted OS. They are dispatched via the TEE pseudo-peripheral. The event type indicates the original source of the event.

- `version`: The version of the structure – currently always 1.
- `event`: The event generated by the REE.
  - The events `TEE_EVENT_TUI_REE_ROT`*xxx* and `TEE_EVENT_TUI_TEE_ROT`*xxx* enable the TEE to inform the TA that the display has been rotated clockwise by 90, 180, or 270 degrees. The TEE SHOULD send `TEE_EVENT_TUI_REE_ROT`*xxx* if the information comes from an untrusted source and `TEE_EVENT_TUI_TEE_ROT`*xxx* if the information is from a trusted source. These events are informational only. It is the duty of the TA to rotate the image if required.
  - The `TEE_EVENT_TUI_REE_DISPLAYSTOPPED` event informs the TA that a system event, such as a phone call, has occurred and the system has taken back control of the display.
  - `TEE_EVENT_TUI_REE_CANCEL` SHOULD be used to tell a TA that it must relinquish its control when (for example) the REE requires the user's immediate attention, either because of an external event or due to the Client Application sending the `TEEC_RequestCancellation` function. Upon receipt of this event, a TA should close the session as quickly as is safely possible. If the TA does not close the TUI session in response to the `TEE_EVENT_TUI_REE_CANCEL` event, the TEE will automatically close the TUI session after `gpd.tee.tui.low.session.timeout`.

### Informative Notes

A TA should not attempt to open a new session immediately after `TEE_EVENT_TUI_REE_CANCEL`, as it is likely to encounter a loop during which the REE event is still ongoing. It is recommended that the calling Client Application allow the user to explicitly restart the transaction.

#### 4.5.10.4 TEE_Event_TUI_TEE

```
typedef struct
{
    uint32_t                  version;
    union {
          TEE_Event_TUI_TEE_V1 v1;
          } u;
} TEE_Event_TUI_TEE;

typedef struct
{
    TEE_EVENT_TUI_TEE_TYPE     event;
} TEE_Event_TUI_TEE_V1;
```

`TEE_Event_TUI_TEE` events enable the TEE to pass events to the calling TA. They may also be used to pass events from other entities outside this TEE, but this specification does not define types for this purpose. A Trusted OS would have to define its own event types for this.

- `version`: The version of the structure – currently always 1.
- `event`: The event generated by the TEE.

The events `TEE_EVENT_TUI_REE_ROT`*xxx* and `TEE_EVENT_TUI_TEE_ROT`*xxx* enable the TEE to inform the TA that the display has been rotated clockwise by 90, 180, or 270 degrees. The TEE SHOULD send `TEE_EVENT_TUI_REE_ROT`*xxx* if the information comes from an untrusted source and `TEE_EVENT_TUI_TEE_ROT`*xxx* if the information is from a trusted source. These events are informational only. It is the duty of the TA to rotate the display.

#### 4.5.10.5 TEE_Event_TUI_Touch

```
typedef struct
{
    uint32_t                      version;
    union  {
           TEE_Event_TUI_Touch_V1  v1;
           } u;
} TEE_Event_TUI_Touch;

typedef struct
{
    uint32_t                      display;
    TEE_EVENT_TUI_TOUCH_ACTION    action;
    uint32_t                      finger;
    uint32_t                      pressure;
    uint32_t                      x;
    uint32_t                      y;
} TEE_Event_TUI_Touch_V1;
```

`TEE_Event_TUI_Touch` events are produced by user interaction with a touch screen display.

Touches are split into one DOWN event, zero or more MOVE events, and one UP event.

'Taps' on a touch screen will yield a DOWN and UP event. 'Swipes' or other complex gestures will yield a DOWN event, zero or more MOVE events tracing the finger movement, followed by a final UP event.

Unless the application is interrupted while a finger is touching the display, there should be exactly one UP event for every DOWN event.

Each touch event gives a position in display coordinates in units of whole pixels. The coordinates of MOVE events correspond to the position of the finger at the end of the move.

The production rate, movement threshold, and resolution of MOVE events are Trusted OS and/or hardware defined. Similarly, the TEE is not required to perform any debouncing or other processing, but may choose to do so.

- `version`: The version of the structure – currently always 1.
- `display`: A pointer to the display that was touched.
- `action`: The action performed, Up, Move, Down; see section 4.3.9.
- `finger`: Each touch event has a 'finger' identifier to allow tracking of multiple touches simultaneously. Devices not supporting multi-touch should only report events for finger zero. Fingers should be allocated from zero upwards, and must be stable for a given contiguous touch gesture or tap. A maximum of 256 simultaneous fingers are supported.
- `pressure`: Touch events have an optional 'pressure' indication. If the hardware and driver support is available, this is a value between 1 and 255 inclusive indicating the pressure of the touch or movement. If no such support is available, DOWN and MOVE events have the pressure set at 127. In any case, and by natural definition, UP events have the pressure set to zero. Variation of pressure (if supported) may be the source of MOVE events.
- `x,y`: The coordinates of the touch in pixels, with `0,0` as the top left of the display. It is the duty of the TEE to convert raw coordinates from the underlying system and report them in the current coordinate system used by the TA to address the display.

**Informative Note:** A TA wishing to provide a familiar mobile device user interface may choose to divulge selected details of some events (such as all touches, without their location) to the REE to allow the REE to replay its default action for touch screen input (such as haptic or sound feedback).

### 4.5.11 TEE_EventQueueHandle

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

A `TEE_EventQueueHandle` is an opaque handle for an event queue.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef struct __TEE_EventQueueHandle* TEE_EventQueueHandle;
#endif
```

A Trusted OS SHOULD ensure that the value of `TEE_EventQueueHandle` returned to a TA is not predictable and SHALL ensure that it does contain all or part of a machine address.

The value `TEE_INVALID_HANDLE` is used to indicate an invalid `TEE_EventQueueHandle`. All other values denote a valid `TEE_EventQueueHandle`.

### 4.5.12 TEE_EventSourceHandle

***Note:*** This data structure will be defined in [TEE Core API] v1.2 and later.

A `TEE_EventSourceHandle` is an opaque handle for a specific source of events, for example a biometric sensor.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  typedef struct __TEE_EventSourceHandle* TEE_EventSourceHandle;
#endif
```

The value `TEE_INVALID_HANDLE` is used to indicate an invalid `TEE_EventSourceHandle`. All other values denote a valid `TEE_EventSourceHandle`.

### 4.5.13 TEE_TUIDisplay

```
typedef struct __TEE_TUIDisplay TEE_TUIDisplay;
```

This is an opaque type.

### 4.5.14 TEE_TUIDisplayInfo

```
typedef struct
{
    uint32_t           version;
    union  {
           TEE_TUIDisplayInfo_V1   v1;
           } u;
} TEE_TUIDisplayInfo;

typedef struct
{
    uint32_t           bitDepth;
    uint32_t           flags;
    uint32_t           pixelWidth;
    uint32_t           pixelHeight;
    uint32_t           physicalWidth;
    uint32_t           physicalHeight;
    uint32_t           numPeripherals;
    TEE_PeripheralId*  associatedPeripherals;
} TEE_TUIDisplayInfo_V1;
```

The `TEE_TUIDisplayInfo` structure contains information about a physical display. It is returned by `TEE_TUI_GetDisplayInformation`, described in section 4.6.21.

- `version`: The version of the structure – currently always 1.
- `bitDepth`: The number of bits per pixel.
- `flags`: One or more `TEE_TUI_DISPLAY_INFO_FLAGS` XORed together; see section 4.3.13.
- `pixelWidth`: The number of pixels in a row across the display.
- `pixelHeight`: The number of rows of pixels in the display.
- `physicalWidth`: The nominal width of the display in millimeters – or zero if unknown.
- `physicalHeight`: The nominal height of the display in millimeters – or zero if unknown.
- `numPeripherals`: The number of associated peripherals.
- `associatedPeripherals`: An array of `TEE_PeripheralIds` listing the peripherals physically associated with this display.

### 4.5.15 TEE_TUIImage

```
typedef struct
{
    TEE_TUIImageSource  source;
    union
    {
        struct
        {
            [inbuf] void* image; size_t imageLength;
        }
        ref;

        struct
        {
            uint32_t storageID;
            [in(objectIDLength)] void* objectID; size_t objectIDLen;
        }
        object;
    };
    uint32_t    width;
    uint32_t    height;
} TEE_TUIImage;
```

This value is also defined in [TEE TUI].

The `TEE_TUIImage` structure defines a way to handle an image for label area and buttons. An image source can be a buffer or a Data Object in the Trusted Storage:

- `source`: The source of the image.
  - If set to `TEE_TUI_NO_SOURCE`, the structure does not refer to an image and all other fields are ignored.
  - If set to `TEE_TUI_REF_SOURCE`, the field `ref` SHALL be selected.
  - If set to `TEE_TUI_OBJECT_SOURCE`, the field `object` SHALL be selected.
- `image, imageLength`: A buffer containing the image. The referenced memory SHALL contain the whole of the image in PNG format.
- `storageID`: The storage to use. It SHALL be `TEE_STORAGE_PRIVATE` as defined in [TEE Core API].
- `objectID, objectIDLen`: The Object Identifier which refers to a Data Object containing the image in PNG format.
- `width`: The number of pixels of the width of the image.
- `height`: The number of pixels of the height of the image.

The `width` and `height` SHALL match the values set within the image in PNG format.

### 4.5.16 TEE_TUIImageSource

```
typedef uint32_t TEE_TUIImageSource;
```

This type is also defined in [TEE TUI]. If a TA is compiled with code that also includes [TEE TUI], then the [TEE TUI] definition takes precedence.

**Table 4-22: `TEE_TUIImageSource` Values**

| Constant Name | Value | Description |
|---|---|---|
| `TEE_TUI_NO_SOURCE` | `0x00000000` | No image is provided as input. |
| `TEE_TUI_REF_SOURCE` | `0x00000001` | The image source is provided as a memory reference. |
| `TEE_TUI_OBJECT_SOURCE` | `0x00000002` | The image source is provided as a Data Object in the Trusted Storage. |
| Reserved | `0x00000003 – 0x7ffffffe` | |
| `TEE_TUI_IMAGE_ILLEGAL_VALUE` | `0x7fffffff` | See below. |
| Implementation Defined | `0x80000000 – 0xffffffff` | |

`TEE_TUI_IMAGE_ILLEGAL_VALUE` is reserved for testing and validation. It SHALL NOT be generated or used in normal operation.

### 4.5.17 TEE_TUISurface

```
typedef struct __TEE_TUISurface  TEE_TUISurface;
```

This is an opaque type.

### 4.5.18 TEE_TUISurfaceBuffer

```
typedef struct
{
    uint32_t  version;
    union {
        TEE_TUISurfaceBuffer_V1 v1;
    } u;
} TEE_TUISurfaceBuffer;

typedef struct
{
    uint32_t  *buf;
} TEE_TUISurfaceBuffer_V1;
```

The `TEE_TUISurfaceBuffer` structure is used to provide a staging area in which images can be prepared before being sent to a display. The buffer can be accessed using standard functions such as `memcpy` and `memset`.

A *surface* is a pixel buffer available for reading and writing by a TA. Pixels are represented in RGBA 8888 format – that is, 8 bits each of red, green, blue, and alpha channels, in that order. The color channels are not pre-multiplied by the alpha channel. The RGB components are interpreted as being linear in the device color space.

In OpenGL terms, this pixel format is GL_RGBA8.

The alpha channel is used if the surface is composited onto another surface or existing image; however, when the image is finally written to a display – a surface obtained from `TEE_TUI_GetDisplaySurface` – the alpha channel is ignored (i.e. this is a straight blit of the color values to the display, not a composition of the image with pure black or another color).

Surfaces have a width and height (in units of whole pixels), and a horizontal stride from one row of pixels to the next (in units of whole pixels).

The stride may be greater than the display width for reasons of row alignment, or because the pixel buffer is a GPU-provided surface of power-2 dimensions.

The *start of a surface* refers to the top left pixel (X=0,Y=0), the *next pixel* to the pixel immediately to the right (X=1,Y=0) of this, and so on until one horizontal stride is completed, when the next row of the display commences at (X=0,Y=1).

The values of pixels unused in a stride (that is, pixels between the surface width and the whole stride) are ignored; however, it is expected that TAs will clip their drawing operations to the valid portion of the surface to avoid overflowing the pixel buffer.

The buffers associated with the screen surface is not accessible directly and the screen surface is always referenced via a handle of type `TEE_TUISurface`, never via a `TEE_TUISurfaceBuffer`.

- `version`: The version of the structure – currently always 1.
- `buf`: An array of 32-bit words containing pixel values.

### 4.5.19 TEE_TUISurfaceInfo

```
typedef struct
{
    uint32_t                  version;
    union  {
           TEE_TUISurfaceInfo_V1   v1;
           } u;
} TEE_TUISurfaceInfo;

typedef struct
{
    uint32_t  bitDepth;
    uint32_t  width;
    uint32_t  height;
    uint32_t  stride;
} TEE_TUISurfaceInfo_V1;
```

The `TEE_TUISurfaceInfo` structure provides information about a surface.

- `version`: The version of the structure – currently always 1.
- `bitDepth`: The bit depth supported by this surface.
- `width`: The width of the surface in pixels.
- `height`: The height of the surface in pixels.
- `stride`: The distance in pixels between the start of one row and the start of the next. This may be larger than the width.

## 4.6 Functions

[TEE Core API] v1.2 and later will define the functions required for processing events and peripherals.

Implementations of [TEE Core API] that define the functions required for processing events and peripherals will define the sentinel `TEE_CORE_API_EVENT`; therefore, implementations of this API SHALL check for this value before defining these functions.

Functions defined in [TEE Core API] will return debug values from that specification – that is, with Specification number 10 and not the values defined in this document.

### 4.6.1 TEE_Peripheral_Close

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_Close( TEE_PeripheralDescriptor *peripheral );
#endif
```

#### Description

The `TEE_Peripheral_Close` function is used by a TA to release a single peripheral. On successful return, the `peripheralHandle` and `eventSourceHandle` values pointed to by `peripheral` SHALL be `TEE_INVALID_HANDLE`.

**Specification Number:** 55 **Function Number:** 0x1E01

#### Parameters

- `peripheral`: A pointer to a `TEE_PeripheralDescriptor` structure.

#### Return Value

- `TEE_SUCCESS`: In case of success. At least one of `peripheralHandle` and `eventSourceHandle` points to a valid handle.
- `TEE_ERROR_BAD_STATE`: The calling TA does not have a valid open handle to the peripheral.
- `TEE_ERROR_BAD_PARAMETERS`: `peripheral` is `NULL`.

#### Panic Reasons

`TEE_Peripheral_Close` SHALL NOT panic.

### 4.6.2 TEE_Peripheral_CloseMultiple

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_CloseMultiple(
        const     uint32_t                 numPeripherals,
        [inout]  TEE_PeripheralDescriptor *peripherals
     );
#endif
```

#### Description

`TEE_Peripheral_CloseMultiple` is a convenience function which closes all the peripherals identified in the buffer pointed to by `peripherals`. In contrast to `TEE_Peripheral_OpenMultiple`, there is no guarantee of atomicity; the function simply attempts to close all the requested peripherals.

**Specification Number:** 55 **Function Number:** 0x1E02

#### Parameters

- `numPeripherals`: The number of entries in the `TEE_PeripheralDescriptor` buffer pointed to by `peripherals`.
- `peripherals`: A pointer to a buffer of `numPeripherals` instances of `TEE_PeripheralDescriptor`. The interpretation and treatment of each individual entry in the buffer of `descriptors` is as described for `TEE_Peripheral_Close` in section 4.6.1.

#### Return Value

- `TEE_SUCCESS`: In case of success, which is defined as all the requested `TEE_PeripheralDescriptor` instances having been successfully closed.
- `TEE_ERROR_BAD_STATE`: The calling TA does not have a valid open handle to at least one of the `peripherals`.
- `TEE_ERROR_BAD_PARAMETERS`: `peripherals` is `NULL` and/or `numPeripherals` is `0`.

#### Panic Reasons

`TEE_Peripheral_CloseMultiple` SHALL NOT panic.

### 4.6.3 TEE_Peripheral_GetPeripherals

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_GetPeripherals(
         [inout]    uint32_t*       version,
         [outbuf]   TEE_Peripheral* peripherals, size_t* size
      );
#endif
```

#### Description

The `TEE_Peripheral_GetPeripherals` function returns information about the peripherals known to the TEE. This function MAY list all peripherals attached to the implementation and SHALL list all peripherals visible to the calling TA. The TEE may not be able to control all the peripherals. Of those that the TEE can control, it may not be able to parse the events generated, so not all can be used as event sources.

**Specification Number:** 55 **Function Number:** 0x1E03

#### Parameters

- `version`:
  - On entry, the highest version of the `TEE_Peripheral` structure understood by the calling program.
  - On return, the actual version returned, which may be lower than the value requested.
- `peripherals`: A pointer to an array of `TEE_Peripheral` structures. This will be populated with information about the available sources on return. Each structure in the array returns information about one peripheral.
- `size`:
  - On entry, the size of `peripherals` in bytes.
  - On return, the actual size of the buffer containing the `TEE_Peripheral` structures in bytes. The combination of `peripherals` and `size` complies with the `[outbuf]` behavior specified in [TEE Core API] section 3.4.4.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OLD_VERSION`: If the version of the `TEE_Peripheral` structure requested is not supported.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to hold all the sources.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `version` is `NULL`.

- If `peripherals` is `NULL` and/or `*size` is not zero.
- See [TEE Core API] section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.4 TEE_Peripheral_GetState

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_GetState(
            const TEE_PeripheralId      id,
            const TEE_PeripheralStateId stateId,
        [out] TEE_PeripheralValueType*  periphType,
        [out] void*                     value
  );
#endif
```

#### Description

The `TEE_Peripheral_GetState` function enables a TA which knows the state ID of a peripheral state item to fetch the value of this directly. A TA does not need to have an open handle to a peripheral to obtain information about its state – this allows a TA to discover information about peripherals available to it before opening a handle.

**Specification Number:** 55 **Function Number:** 0x1E04

#### Parameters

- `id`: The unique peripheral identifier for the peripheral in which we are interested.
- `stateId`: The identifier for the state item for which the value is requested.
- `periphType`: On return, contains a value of `TEE_PeripheralValueType` which determines how the data pointed to by `value` should be interpreted.
- `value`: On return, points to the value of the requested state item.

**Note:** The caller SHALL ensure that the buffer pointed to by `value` is large enough to accommodate whichever is the larger of `uint64_t` and `char*` on a given TEE platform.

#### Return Value

- `TEE_SUCCESS`: State information has been fetched.
- `TEE_ERROR_BAD_PARAMETERS`: The value of one or both of `id` or `stateId` are not valid for this TA; `periphType` or `value` is `NULL`.

#### Panic Reasons

`TEE_Peripheral_GetState` SHALL NOT panic.

### 4.6.5 TEE_Peripheral_GetStateTable

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_GetStateTable(
      [in] TEE_PeripheralId id,
      [outbuf] TEE_PeripheralState* stateTable, size_t* bufSize
  );
#endif
```

#### Description

The `TEE_Peripheral_GetStateTable` function fetches a buffer containing zero or more instances of `TEE_PeripheralState`. These provide a snapshot of the state of a peripheral.

**Specification Number:** 55 **Function Number:** 0x1E05

#### Parameters

- `id`: The `TEE_PeripheralId` for the peripheral from which the TA wishes to read data
- `stateTable`: A buffer of at least `bufSize` bytes that on successful return is overwritten with an array of `TEE_PeripheralState` structures.
- `bufSize`:
  - On entry, the size of `stateTable` in bytes.
  - On return, the actual number of bytes in the array. The combination of `stateTable` and `bufSize` complies with the `[outbuf]` behavior specified in [TEE Core API] section 3.4.4.

#### Return Value

- `TEE_SUCCESS`: Data has been written to the peripheral.
- `TEE_ERROR_BAD_PARAMETERS`: The value of `id` or `stateTable` is `NULL` and/or `bufSize` is `0`.

#### Panic Reasons

- See [TEE Core API] section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.6 TEE_Peripheral_Open

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_Open( TEE_PeripheralDescriptor *peripheral );
#endif
```

#### Description

The `TEE_Peripheral_Open` function is used by a TA to obtain descriptor(s) enabling access to a single peripheral. If the TA needs to open more than one peripheral, it SHOULD use `TEE_Peripheral_OpenMultiple`.

If this function executes successfully and if `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` indicates that exclusive access is supported, then the Trusted OS guarantees that neither the REE, nor any other TA, has access to the peripheral. If `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` indicates that exclusive access is not supported, the calling TA SHOULD assume that it does not have exclusive access to the peripheral.

The Trusted OS returns handles which can be used by the TA to manage interactions with the peripheral. If `TEE_Peripheral_Open` succeeds, at least one of `peripheralHandle` and `eventSourceHandle` is set to a valid handle value.

It is an error to call `TEE_Peripheral_Open` for a peripheral which is already owned by the calling TA instance.

**Specification Number:** 55 **Function Number:** 0x1E06

#### Parameters

- `peripheral`: A pointer to a `TEE_PeripheralDescriptor` structure. The fields of the structure pointed to are used as follows:
  - `id`: This is the unique identifier for a specific peripheral, as returned by `TEE_Peripheral_GetPeripherals`. This field SHALL be set on entry, and SHALL be unchanged on return.
  - `peripheralHandle`: On entry, the value is ignored and will be overwritten.
    On return, the value is set as follows:
    - `TEE_INVALID_HANDLE`: This peripheral does not support the Peripheral API.
    - Other value: An opaque handle which can be used with the Peripheral API functions.
  - `eventSourceHandle`: On entry, the value is ignored and will be overwritten.
    On return, the value is set as follows:
    - `TEE_INVALID_HANDLE`: This peripheral does not support the Event API.
    - Other value: An opaque handle which can be used with the Event API functions.

#### Return Value

- `TEE_SUCCESS`: In case of success. At least one of `peripheralHandle` and `eventSourceHandle` points to a valid handle.
- `TEE_ERROR_BAD_PARAMETERS`: `peripheral` is `NULL`.

- `TEE_ERROR_ACCESS_DENIED`: If the system was unable to acquire exclusive access to a peripheral for which `TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS` indicates exclusive access is possible.

### Panic Reasons

- If `peripheral->id` is not known to the system.
- If `peripheral->id` is already owned by the calling TA instance.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.7 TEE_Peripheral_OpenMultiple

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_OpenMultiple(
         const     uint32_t                  numPeripherals,
         [inout]  TEE_PeripheralDescriptor *peripherals
      );
#endif
```

#### Description

The TEE_Peripheral_OpenMultiple function is used by a TA to atomically obtain access to multiple peripherals.

TEE_Peripheral_OpenMultiple behaves as though a call to TEE_Peripheral_Open is made to each TEE_PeripheralDescriptor in peripherals in turn, but ensures that all or none of the peripherals have open descriptors on return. This function should be used where a TA needs simultaneous control of multiple peripherals to operate correctly.

If this function executes successfully, the Trusted OS guarantees that neither the REE, nor any other TA, has access to the any of the requested peripherals for which exclusive access is supported (as indicated by TEE_PERIPHERAL_STATE_EXCLUSIVE_ACCESS). If an error is returned, the Trusted OS guarantees that no handle is open for any of the requested peripherals.

The Trusted OS returns handles which can be used by the TA to manage interactions with the peripheral. If TEE_Peripheral_OpenMultiple succeeds, at least one of peripheralHandle and eventSourceHandle fields in each descriptor is set to a valid handle value. If an error is returned, all the peripheralHandle and eventSourceHandle fields in each descriptor SHALL contain TEE_INVALID_HANDLE.

**Specification Number:** 55 **Function Number:** 0x1E07

#### Parameters

- numPeripherals: The number of entries in the TEE_PeripheralDescriptor buffer pointed to by peripherals.
- peripherals: A pointer to a buffer of numPeripherals instances of TEE_PeripheralDescriptor. The interpretation and treatment of each individual entry in the buffer of descriptors is as described for TEE_Peripheral_Open in section 4.6.6.

#### Return Value

- TEE_SUCCESS: In case of success. At least one of peripheralHandle and eventSourceHandle points to a valid handle in every entry in peripherals.
- TEE_ERROR_BAD_PARAMETERS: peripherals is NULL and/or numPeripherals is 0.
- TEE_ERROR_ACCESS_DENIED: If the system was unable to acquire exclusive access to all the requested peripherals.

### Panic Reasons

- If `peripherals[x].id` for any instance, `x`, of `TEE_PeripheralDescriptor` is not known to the system.
- If `peripherals[x].id` for any instance, `x`, of `TEE_PeripheralDescriptor` is already owned by the calling TA.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.8 TEE_Peripheral_Read

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_Read(
         [in]      TEE_PeripheralHandle peripheralHandle,
         [outbuf]  void *buf,   size_t *bufSize
      );
#endif
```

#### Description

The `TEE_Peripheral_Read` function provides a means to read data from the peripheral denoted by `peripheralHandle`. The `peripheralHandle` field of the peripheral descriptor must be a valid handle for this function to succeed.

The calling TA allocates a buffer of `bufSize` bytes before calling. On return, this will contain as much data as is available from the peripheral, up to the limit of `bufSize`. The `bufSize` parameter will be updated with the actual number of bytes placed into `buf`.

`TEE_Peripheral_Read` is designed to allow a TA to implement polled communication with peripherals. The function SHALL NOT wait on any hardware signal, and SHALL retrieve only the data which is available at the time of calling.

While some peripherals may support both the event queue and the polling interface, it is recommended that TA implementers do not attempt to use both the Peripheral API and the Event API to read data from the same peripheral. Peripheral behavior if both APIs are used simultaneously on the same peripheral is undefined.

**Specification Number:** 55 **Function Number:** 0x1E08

#### Parameters

- `peripheralHandle`: A valid `TEE_PeripheralHandle` for the peripheral from which the TA wishes to read data.
- `buf`: A buffer of at least `bufSize` bytes which, on successful return, will be overwritten with data read back from the peripheral.
- `bufSize`:
  - On entry, the size of `buf` in bytes.
  - On return, the actual number of bytes read from the peripheral. The combination of `buf` and `bufSize` complies with the `[outbuf]` behavior specified in [TEE Core API] section 3.4.4.

#### Return Value

- `TEE_SUCCESS`: Data has been read from the peripheral. The value of `bufSize` indicates the number of bytes read.
- `TEE_ERROR_EXCESS_DATA`: Data was read successfully, but the peripheral has more data available to read. In this case, `bufSize` is the same value as was indicated when the function was called. It is recommended that the TA read back the remaining data from the peripheral before continuing.
- `TEE_ERROR_BAD_PARAMETERS`: The value of `peripheralHandle` is `TEE_INVALID_HANDLE`; or `buf` is `NULL` and `bufSize` is not zero.

### Panic Reasons

- If the calling TA does not provide a valid `peripheralHandle`.
- See [TEE Core API] section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.9 TEE_Peripheral_SetState

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_SetState(
      const TEE_PeripheralHandle       handle,
      const TEE_PeripheralStateId      stateId,
      const TEE_PeripheralValueType    periphType,
      const void*                      value
  );
#endif
```

#### Description

The `TEE_Peripheral_SetState` function enables a TA to set the value of a writeable peripheral state item. Items are only writeable if the `ro` field of the `TEE_PeripheralState` for the state item is `false`.

TAs SHOULD call `TEE_Peripheral_GetStateTable` for the peripheral id in question to determine which state items are writeable by the TA.

Note that any previous snapshot of peripheral state will not be updated after a call to `TEE_Peripheral_SetState`.

**Specification Number:** 55 **Function Number:** 0x1E09

#### Parameters

- `handle`: A valid open handle for the peripheral whose state is to be updated.
- `stateId`: The identifier for the state item for which the value is requested.
- `periphType`: A value of `TEE_PeripheralValueType` which determines how the data pointed to by `value` should be interpreted.
- `value`: The address of the value to be written to the state item.

#### Return Value

- `TEE_SUCCESS`: State information has been updated.
- `TEE_ERROR_BAD_PARAMETERS`: The value of one or both of `handle` or `stateId` are not valid for this TA; or `periphType` is not a value defined in `TEE_PeripheralValueType`; or `value` is `NULL`; or the value which is being written is read-only.

#### Panic Reasons

`TEE_Peripheral_SetState` SHALL NOT panic.

### 4.6.10 TEE_Peripheral_Write

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Peripheral_Write(
         [in]      TEE_PeripheralHandle peripheralHandle,
         [inbuf]   void *buf,   size_t bufSize
      );
#endif
```

#### Description

The `TEE_Peripheral_Write` function provides a means to write data to the peripheral denoted by `peripheralHandle`. The `peripheralHandle` field of the peripheral descriptor must be a valid handle for this function to succeed.

The calling TA allocates a buffer of `bufSize` bytes before calling, and fills it with the data to be written.

**Specification Number:** 55 **Function Number:** 0x1E0A

#### Parameters

- `peripheralHandle`: A valid `TEE_PeripheralHandle` for the peripheral from which the TA wishes to read data.
- `buf`: A buffer of at least `bufSize` bytes containing data which has, on successful return, been written to the peripheral.
- `bufSize`: The size of `buf` in bytes.

#### Return Value

- `TEE_SUCCESS`: Data has been written to the peripheral.
- `TEE_ERROR_BAD_PARAMETERS`: `buf` is `NULL` and/or `bufSize` is `0`.

#### Panic Reasons

- `peripheralHandle` is not a valid open handle to a peripheral.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.11 TEE_Event_AddSources

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_AddSources(
               uint32_t                numSources,
         [in]  TEE_EventSourceHandle   *sources,
         [in]  TEE_EventQueueHandle    *handle
  );
#endif
```

#### Description

The `TEE_Event_AddSources` function atomically adds new event sources to an existing queue acquired by a call to `TEE_Event_OpenQueue`**.** If the function succeeds, events from this source are exclusively available to this queue.

If the function fails, the queue is still valid. The queue SHALL contain events from the original sources and MAY contain some of the requested sources. In case of error, the caller should use `TEE_Event_ListSources` to determine the current state of the queue.

It is not an error to add an event source to a queue to which it is already attached.

**Specification Number:** 55 **Function Number:** 0x1E0B

#### Parameters

- `numSources`: Defines how many sources are provided.
- `sources`: An array of `TEE_EventSourceHandle` that the TA wants to add to the queue.
- `handle`: The handle for the queue.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the handle does not represent a currently open queue.
- `TEE_ERROR_BUSY`: If any of the requested resources cannot be reserved.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `handle` is `NULL`.
- If the `sources` array does not contain `numSources` elements.
- If any of the pointers in `sources` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.12 TEE_Event_CancelSources

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_CancelSources(
             uint32_t                 numSources,
       [in]  TEE_EventSourceHandle    *sources,
       [in]  TEE_EventQueueHandle     *handle
);
#endif
```

#### Description

The `TEE_Event_CancelSources` function drops all existing events from a set of sources from a queue previously acquired by a call to `TEE_Event_OpenQueue`.

New events from these sources will continue to be added to the queue, unless the TA has released the sources using `TEE_Event_DropSources` or `TEE_Event_CloseQueue`.

It is not an error to cancel an event source that is not currently attached to the queue.

**Specification Number:** 55 **Function Number:** 0x1E0C

#### Parameters

- `numSources`: Defines how many sources are provided.
- `sources`: An array of `TEE_EventSourceHandle`. Events from these sources are cleared from the queue.
- `handle`: The handle for the queue.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.
- `TEE_ERROR_BAD_STATE`: If the handle does not represent a currently open queue.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `handle` is `NULL`.
- If the `sources` array does not contain `numSources` elements.
- If any of the pointers in `sources` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.13 TEE_Event_CloseQueue

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_CloseQueue( [in] TEE_EventQueueHandle *handle );
#endif
```

#### Description

The `TEE_Event_CloseQueue` function releases TUI resources previously acquired by a call to `TEE_Event_OpenQueue`.

All outstanding events on the queue will be invalidated.

**Specification Number:** 55 **Function Number:** 0x1E0D

#### Parameters

- `handle`: The handle to the `TEE_EventQueueHandle` to close.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the handle does not represent a currently open queue.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

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

### 4.6.14 TEE_Event_DropSources

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_DropSources(
             uint32_t                  numSources,
  [in]       TEE_EventSourceHandle     *sources,
  [in]       TEE_EventQueueHandle      *handle
  );
#endif
```

#### Description

The `TEE_Event_DropSources` function removes one or more event sources from an existing queue previously acquired by a call to `TEE_Event_OpenQueue`. No more events from these sources are added to the queue. Events from these sources will be available to the REE, until they are reserved by this or another TA using `TEE_Event_AddSources` or `TEE_Event_OpenQueue`.

Events from other event sources will continue to be added to the queue. It is permissible to have a queue with no current event sources attached to it.

It is not an error to drop an event source that is not currently attached to the queue.

**Specification Number:** 55 **Function Number:** 0x1E0E

#### Parameters

- `numSources`: Defines how many sources are provided.
- `sources`: An array of `TEE_EventSourceHandle`. Events from these sources are cleared from the queue.
- `handle`: The handle for the queue.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the handle does not represent a currently open queue.
- `TEE_ERROR_ITEM_NOT_FOUND`: If one or more sources was not attached to the queue. All other sources are dropped.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `handle` is `NULL`.
- If the `sources` array does not contain `numSources` elements.
- If any of the pointers in `sources` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.15 TEE_Event_ListSources

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_ListSources(
        [in]       TEE_EventQueueHandle *handle,
        [outbuf]   TEE_EventSourceHandle *sources, size_t* bufSize
  );
#endif
```

#### Description

The `TEE_Event_ListSources` function returns information about sources currently attached to a queue.

**Specification Number:** 55 **Function Number:** 0x1E0F

#### Parameters

- `handle`: The handle for the queue.
- `sources`: A buffer of at least `bufSize` bytes that on successful return is overwritten with an array of `TEE_EventSourceHandle` structures.
- `bufSize`:
  - On entry, the size of `sources` in bytes.
  - On return, the actual number of bytes in the array. The combination of `sources` and `bufSize` complies with the `[outbuf]` behavior specified in [TEE Core API] section 3.4.4.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.
- `TEE_ERROR_SHORT_BUFFER`: If the output buffer is not large enough to hold all the sources.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `handle` is `NULL`.
- If `bufSize` is `NULL`.
- If `sources` is `NULL`.
- See [TEE Core API] section 3.4.4 for reasons for `[outbuf]` generated panic.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.16 TEE_Event_OpenQueue

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_OpenQueue(
  [inout]   uint32_t                  *version,
            uint32_t                  numSources,
            uint32_t                  timeout,
  [in]      TEE_EventSourceHandle     *sources,
  [out]     TEE_EventQueueHandle      *handle
  );
#endif
```

#### Description

The `TEE_Event_OpenQueue` function claims an exclusive access to TUI resources for the current TA instance.

This function allows for multiple event sources to be reserved.

It is possible for multiple TAs to open queues at the same time provided they do not try to reserve any of the same resources.

An individual TA SHALL NOT open multiple queues; instead, the TA SHOULD use `TEE_Event_AddSources` and `TEE_Event_DropSources` to add and remove event sources from the queue.

The `TEE_EventQueue` will be closed automatically if no calls to `TEE_Event_Wait` are made for `timeout` milliseconds. This has the same guarantees as the `TEE_Wait` function in [TEE Core API].

**Specification Number:** 55 **Function Number:** 0x1E10

#### Parameters

- `version`:
  - On entry, the highest version of the `TEE_Event` structure understood by the calling program.
  - On return, the actual version of the `TEE_Event` structure that will be added to the queue, which may be lower than the value requested.
- `numSources`: Defines how many sources are provided.
- `timeout`: The timeout for this function in milliseconds.
- `sources`: An array of `TEE_EventSourceHandle`, as returned from `TEE_Event_ListSources`.
- `handle`: The handle for this session. This value SHOULD Be Zero on entry and is set if the session is successfully established and `numSources` is not zero.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BUSY`: If any of the requested resources cannot be reserved.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

- `TEE_ERROR_OLD_VERSION`: If the version of the `TEE_Event` structure requested is not supported.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

### Panic Reasons

- If `version` is `NULL`.
- If `handle` is `NULL`.
- If the `sources` array does not contain `numSources` elements.
- If any of the pointers in `sources` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.17 TEE_Event_Wait

***Note:*** This function will be defined in [TEE Core API] v1.2 and later.

```
#ifdef TEE_CORE_API_EVENT
// TEE_EVENT structure defined in TEE Core API
#else
  TEE_Result TEE_Event_Wait(
    [in]      TEE_EventQueue  *handle,
              uint32_t        timeout,
    [inout]   TEE_Event       *events,
    [inout]   uint32_t        *numEvents,
    [out]     uint32_t        *dropped
);
#endif
```

#### Description

The `TEE_Event_Wait` function fetches events that have been returned from a peripheral reserved by `TEE_Event_OpenQueue`.

The API allows one or more events to be obtained at a time to minimize any context switching overhead, and to allow a TA to process bursts of events en masse.

Obtaining events has a timeout, allowing a TA with more responsibilities than just user interaction to attend to these periodically without needing to use multi-threading.

The `TEE_Event_Wait` function opens the input event stream. If the stream is not available for exclusive access within the specified timeout, an error is returned. A zero timeout means this function returns immediately. This has the same guarantees as the `TEE_Wait` function in [TEE Core API].

Events are returned in order of decreasing age: `events[0]` is the oldest available event, `events[1]` the next oldest, etc.

On entry, `*numEvents` contains the number of events pointed to by `events`.

`*numEvents` can be `0` on entry, which allows the TA to query whether input is available. If `timeout==0`, the function should return `TEE_SUCCESS` if there are pending events and `TEE_ERROR_TIMEOUT` if there is no pending event.

On return, `*numEvents` contains the actual number of events written to `events`.

If the function returns with any status other than `TEE_SUCCESS`, `*numEvents = 0`.

If there are no events available in the given timeout, `*numEvents` is set to zero and this function returns an error.

If any events occur, the function returns as soon as possible, and does not wait until `*numEvents` events have occurred.

If `dropped` is non-`NULL`, the current count of dropped events is written to this location.

This function is cancellable. If the cancelled flag of the current instance is set and the TA has unmasked the effects of cancellation, then this function returns earlier than the requested timeout.

- If the cancellation occurs in response to an REE event (as discussed in section 3.8), the TEE SHOULD cancel the function and `TEE_ERROR_EXTERNAL_CANCEL` is returned.
- If the operation was cancelled by the client, `TEE_ERROR_CANCEL` is returned. See [TEE Core API] section 4.10 for more detail about cancellations.

**Specification Number:** 55 **Function Number:** 0x1E11

### Parameters

- `handle`: The handle for the queue
- `timeout`: The timeout in milliseconds
- `events`: A pointer to an array of `TEE_Event` structures
- `numEvents`:
  - On entry, the maximum number of events to return
  - On return, the actual number of events returned
- `dropped`: A pointer to a count of dropped events

### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If `handle` does not represent a currently open queue.
- `TEE_ERROR_TIMEOUT`: If there is no event to return within the timeout.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_CANCEL`: If the operation was cancelled by the client.

### Panic Reasons

- If `handle` is `NULL`.
- If `events` is `NULL`.
- If `numEvents` is `NULL`.
- If `dropped` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.18 TEE_TUI_BlitDisplaySurface

```
TEE_Result TEE_TUI_BlitDisplaySurface(
[in]       TEE_TUIDisplay *display
);
```

#### Description

The `TEE_TUI_BlitDisplaySurface` function blits the contents of the surface associated with a display onto that display.

This function blocks until the blit has completed (therefore the TA can start writing another frame as soon as control is returned).

Calling `TEE_TUI_BlitDisplaySurface` invalidates the surface associated with the display. The TA must call `TEE_TUI_GetDisplaySurface` again before updating the display again.

When the TA calls `TEE_TUI_BlitDisplaySurface`, the TEE SHALL turn on the security indicator if one is present.

**Specification Number:** 55 **Function Number:** 0x0101

#### Parameters

- `display`: The display to update.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the TUI session has expired.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `display` is `NULL` or does not point to a valid display.
- If `surface` is `NULL` or does not point to a valid surface.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.19 TEE_TUI_CloseSession

```
TEE_Result TEE_TUI_CloseSession(void);
```

#### Description

The TEE_TUI_CloseSession function releases TUI resources previously acquired. This function SHOULD be called as soon as possible to avoid a bad user experience.

All surfaces and surface buffers created in the session are invalidated.

This function will close a session opened with TEE_TUI_InitSessionLow.[1]

**Specification Number:** 55 **Function Number:** 0x0102

#### Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_BAD_STATE: If the current TA instance is not within a TUI session initially started by a successful call to TEE_TUI_InitSessionLow.
- TEE_ERROR_BUSY: If the TUI resources are currently in use, i.e. a TUI screen is displayed. Can only be returned by a Trusted OS supporting multi-threading within a TA and will occur when a thread tries to close a TUI session that is displaying a TUI screen in another thread.
- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

TEE_TUI_CloseSession SHALL NOT panic.

[1] A Trusted User Interface high-level session, opened with TEE_TUIInitSession() from [TEE TUI], is closed with TEE_TUICloseSession.

### 4.6.20 TEE_TUI_DrawRectangle

```
TEE_Result TEE_TUI_DrawRectangle(
[in]       TEE_TUISurface  *surface,
           uint32_t              x1,
           uint32_t              y1,
           uint32_t              x2,
           uint32_t              y2,
           uint32_t              color,
[in]       TEE_TUI_SURFACE_OPACITY opacity
);
```

#### Description

The `TEE_TUI_DrawRectangle` function draws a filled rectangle in a given surface.

Pixels are inclusive: Setting x1 = x2 will produce a vertical line one pixel wide. Similarly setting y1 = y2 produces a horizontal line one pixel wide.

There is no provision for different border and fill colors; this can be achieved by drawing multiple rectangles.

**Specification Number:** 55 **Function Number:** 0x0103

#### Parameters

- `surface`: A pointer to the surface on which the rectangle will be drawn.
- `x1,y1`: The coordinates of one corner of the rectangle (usually the top left).
- `x2,y2`: The coordinates of the diagonally opposite (usually the bottom right) corner of the rectangle.
- `color`: A 32-bit value containing 8-bit alpha, red, green, and blue values.
- `opacity`: The `TEE_TUI_SURFACE_OPACITY` to apply to this rectangle; see section 4.3.16 for more information.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If `surface` is `NULL` or does not point to a valid surface.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.21 TEE_TUI_GetDisplayInformation

```
TEE_Result TEE_TUI_GetDisplayInformation(
          uint32_t           displayNumber,
[inout]   uint32_t           *version,
[out]     TEE_TUIDisplayInfo *displayInfo,
[out]     TEE_TUIDisplay     *display
);
```

#### Description

The `TEE_TUI_GetDisplayInformation` function retrieves information about a display, including pixel size, physical size, and bit depth.

This function may be called outside a session.

**Specification Number:** 55 **Function Number:** 0x0104

#### Parameters

- `displayNumber`: The number of the display whose information is to be retrieved. Displays are numbered sequentially from `0`.
- `version`:

  On entry, the highest version of the `TEE_TUIDisplayInfo` structure understood by the calling program.

  On return, the actual version of the `TEE_TUIDisplayInfo` structure returned, which may be lower than the value requested.
- `displayInfo`: A pointer to a `TEE_TUIDisplayInfo` structure.

  This structure SHALL be allocated by the TA prior to calling the function.

  On return, this structure is populated by the TUI implementation.
- `display`: A pointer to the `TEE_TUIDisplay` structure for this display.

  This structure SHALL be allocated by the TA prior to calling the function.

  This structure is populated by the TUI implementation.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OLD_VERSION`: If the version of the `TEE_TUIDisplayInfo` structure requested is not supported.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_ITEM_NOT_FOUND`: If the requested display is not present.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `version` is `NULL`.
- If `displayInfo` is `NULL`.

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

### 4.6.22 TEE_TUI_GetDisplaySurface

```
TEE_Result TEE_TUI_GetDisplaySurface(
[in]       TEE_TUIDisplay *display,
[out]      TEE_TUISurface *surface
);
```

#### Description

The TEE_TUI_GetDisplaySurface function yields a surface which will be displayed when TEE_TUI_BlitDisplaySurface is called.

This surface belongs to the display; calling TEE_TUI_ReleaseSurface on it is an illegal operation.

This surface remains valid until the next call to TEE_TUI_BlitDisplaySurface on the corresponding display or until the end of the TUI session, whichever occurs sooner.

The TA does not have direct access to this surface – the TA SHALL call TEE_TUI_GetSurface to obtain a surface it can access.

The TA cannot create a Surface Buffer for this display, if the TA calls TEE_TUI_GetSurfaceBuffer on this surface, the TEE SHALL panic.

**Specification Number:** 55 **Function Number:** 0x0105

#### Parameters

- display: The display to be opened.
- surface: This surface associated with this display.
  - If the trusted application has previously called TEE_TUI_BlitDisplaySurface on this display in this session, the surface SHALL have the same content as the surface associated with the display as the result of the previous TEE_TUI_BlitDisplaySurface.
  - If the trusted application has not previously called TEE_TUI_BlitDisplaySurface on this display in this session, the surface SHALL be zero.

#### Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_BAD_STATE: If the TUI session has expired.
- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If display is NULL or does not point to a valid display.
- If surface is NULL.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.23 TEE_TUI_GetPNGInformation

```
TEE_Result TEE_TUI_GetPNGInformation(
[in]       TEE_TUIImage *png,
[out]      uint32_t     *width,
[out]      uint32_t     *height,
[out]      uint32_t     *colorType
);
```

#### Description

The `TEE_TUI_GetPNGInformation` function returns the height, width, and color type of a given PNG image.

**Specification Number:** 55 **Function Number:** 0x0106

#### Parameters

- `png`: A pointer to `TEE_TUIImage` containing an image.
- `width`: The width in pixels of the image.

  On entry, SHOULD be set to zero.

  Will be populated on return.
- `height`: The height in pixels of the image.

  On entry, SHOULD be set to zero.

  Will be populated on return.
- `colorType`: The color type of the image.

  On entry, SHOULD be set to zero.

  Will be populated on return:
  - Grayscale (0) up to 8 bits depth
  - Truecolor (2) up to 24 bits

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_FORMAT`: If the image format is not supported.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `png` is `NULL` or cannot be interpreted as a valid PNG image.
- If `width` is `NULL`.
- If `height` is `NULL`.
- If `colorType` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.24 TEE_TUI_GetSurface

```
TEE_Result TEE_TUI_GetSurface(
           uint32_t       width,
           uint32_t       height,
           uint32_t       stride,
[out]      TEE_TUISurface *out
);
```

#### Description

The `TEE_TUI_GetSurface` function creates a new surface of the given width and height.

Surfaces have a width and height (in units of whole pixels), and a horizontal stride from one row of pixels to the next (in units of whole pixels).

The stride may be greater than the display width for reasons of row alignment, or because the pixel buffer is a GPU-provided surface of power-2 dimensions.

The *start of a surface* refers to the top left pixel, the *next pixel* to the pixel immediately to the right of this, and so on until one horizontal width is completed, when the next row of the display commences.

The content of surface `*out` will be set to all zero.

This function may be called at any time; it does not require a TUI session.

The caller must call `TEE_TUI_ReleaseSurface` when it has finished using the surface `*out`.

**Specification Number:** 55 **Function Number:** 0x0107

#### Parameters

- `width`: The width of the surface in pixels.
- `height`: The height of the surface in pixels.
- `stride`: The stride of the surface in pixels.
- `out`: Pointer to the new surface. This may be `NULL` on entry.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `stride` is less than `width`.
- If `out` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.25 TEE_TUI_GetSurfaceBuffer

```
TEE_Result TEE_TUI_GetSurfaceBuffer(
[inout]   uint32_t                *version,
[in]       TEE_TUISurface         *surface,
[out]      TEE_TUISurfaceBuffer   *buf
);
```

#### Description

The `TEE_TUI_GetSurfaceBuffer` function yields the buffer for writing to, and reading from, this surface. This enables the TA to update the contents using standard functions such as `memcpy`.

A TA can only obtain a buffer for a surface created with `TEE_TUI_GetSurface`. `TEE_TUI_GetSurfaceBuffer` cannot be used for a display surface, one created by `TEE_TUI_GetDisplaySurface`.

For a given surface, the `TEE_TUISurfaceBuffer` is constant (in other words, the location and dimensions of the `TEE_TUISurfaceBuffer` do not vary with time), though the contents are variable.

This buffer remains valid for as long as the underlying surface is valid.

**Specification Number:** 55 **Function Number:** 0x0108

#### Parameters

- `version`:

  On entry, the highest version of the `TEE_TUISurfaceBuffer` structure understood by the calling program.

  On return, the actual version returned, which may be lower than the value requested.

- `surface`: An existing surface.
- `buf`: A buffer with the height and width of the given surface.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OLD_VERSION`: If the version of the `TEE_TUISurfaceBuffer` structure requested is not supported.
- `TEE_ERROR_BAD_STATE`: If the surface does not exist or has been released.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `version` is `NULL`.
- If `surface` is `NULL`.
- If `buf` is `NULL`.
- If `surface` is the surface associated with a display.

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

### 4.6.26 TEE_TUI_GetSurfaceInformation

```
TEE_Result TEE_TUI_GetSurfaceInformation(
[inout]   uint32_t            *version,
[in]      TEE_TUISurface      *surface,
[out]     TEE_TUISurfaceInfo  *info
);
```

#### Description

The `TEE_TUI_GetSurfaceInformation` function returns information about a surface.

**Specification Number:** 55 **Function Number:** 0x0109

#### Parameters

- `version`:

  On entry, the highest version of the `TEE_TUISurfaceInfo` structure understood by the calling program.

  On return, the actual version returned, which may be lower than the value requested.

- `surface`: A pointer to the surface whose information is to be retrieved.
- `info`: A pointer to the structure holding information about this surface.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_OLD_VERSION`: If the version the `TEE_TUISurfaceInfo` structure requested is not supported.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `version` is `NULL`.
- If `surface` is `NULL` or is not a pointer to a valid surface.
- If `info` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.27 TEE_TUI_InitSessionLow

```
TEE_Result TEE_TUI_InitSessionLow(
           uint32_t                    numDisplays,
           uint32_t                    timeout,
           uint32_t                    flags,
[in]       TEE_TUIDisplay              *displays
           const uint32_t              numPeripherals,
[inout]TEE_PeripheralDescriptor    *peripherals
);
```

#### Description

The `TEE_TUI_InitSessionLow` function atomically claims an exclusive access to TUI displays and TUI input peripherals for the current TA instance. It behaves as though the following actions were done:

- Claim exclusive ownership of the TUI displays.
- Claim exclusive ownership of the TUI input peripherals, as though `TEE_Peripheral_OpenMultiple` were called on these peripherals.

`TEE_TUI_InitSessionLow` ensures that either all or none of these actions are performed.

The list of peripherals in `peripherals` SHALL contain every peripheral that will pass events associated with the displays to be opened. It MAY contain additional input peripherals the TA wants exclusive access to, for instance to prevent or limit side channel attacks.

The following restrictions apply to opening multiple Trusted User Interface sessions:

- An individual TA may open a single session with either `TEE_TUI_InitSessionLow` or `TEE_TUIInitSession` (from [TEE TUI]), but may not have both open simultaneously.
- If any TA has a session opened with `TEE_TUIInitSession`, then no TA may open a session with `TEE_TUI_InitSessionLow`.
- Multiple TAs may open sessions with `TEE_TUI_InitSessionLow` provided they do not attempt to access any of the same resources.

Control of ALL displays listed SHALL be taken over by the TEE at this stage, though the TEE security indicator SHALL NOT be illuminated until the TA first calls `TEE_TUI_BlitDisplaySurface`.

As a limited resource, the TUI session will be closed automatically whenever the TA does not update the display with `TEE_TUI_BlitDisplaySurface` for a period. This period is equal to the value of the property `gpd.tee.tui.low.session.timeout` or the `timeout` value, whichever is lower. This has the same guarantees as the `TEE_Wait` function in [TEE Core API].

When the TUI session terminates, all event sources acquired by the TA by the call to `TEE_TUI_InitSessionLow` will be closed.

**Specification Number:** 55 **Function Number:** 0x010A

#### Parameters

- `numDisplays`: Defines how many display structures are provided – this SHALL be less than `gpd.tee.tui.low.numDisplays`.
- `timeout`: The timeout for this function in milliseconds – overrides `gpd.tee.tui.low.session.timeout` if `timeout` is smaller.
- `flags`: One or more `TEE_TUI_INIT_SESSION_LOW_FLAGS` XORed together; see section 4.3.14.

- displays: An array of TEE_TUIDisplay pointers; see section 4.5.13.
- numPeripherals: The number of entries in the TEE_PeripheralDescriptor buffer pointed to by peripherals.
- peripherals: A pointer to a buffer of numPeripherals instances of TEE_PeripheralDescriptor.

### Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_BUSY: If another TA has a lock on the resources needed.
- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- TEE_ERROR_OUT_OF_MEMORY: If the system ran out of resources.

### Panic Reasons

- If displays is NULL or does not contain numDisplays elements.
- If TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR is set when gpd.tee.tui.low.securityIndicator is false.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.28 TEE_TUI_ReleaseSurface

```
TEE_Result TEE_TUI_ReleaseSurface(
[in]       TEE_TUISurface *surface
);
```

#### Description

The TEE_TUI_ReleaseSurface function destroys memory associated with the surface.

This function must be called only on surfaces returned by TEE_TUI_GetSurface. It SHALL NOT be called on the surface returned by TEE_TUI_GetDisplaySurface; doing so will cause a panic.

**Specification Number:** 55 **Function Number:** 0x010B

#### Parameters

- surface: The surface to release.

#### Return Value

- TEE_SUCCESS: In case of success.
- TEE_ERROR_BAD_STATE: If the surface does not exist or has been released.
- TEE_ERROR_EXTERNAL_CANCEL: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.

#### Panic Reasons

- If surface is NULL.
- If surface is not releasable; that is, it was allocated by TEE_TUI_GetDisplaySurface.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.29 TEE_TUI_SetPNG

```
TEE_Result TEE_TUI_SetPNG(
[in]       TEE_TUISurface         *surface,
[in]       TEE_TUIImage           *png,
[in]       uint32_t                x,
[in]       uint32_t                y,
[in]       TEE_TUI_SURFACE_OPACITY opacity
);
```

#### Description

The `TEE_TUI_SetPNG` function renders a PNG image onto a surface at a given location. Any portions of the image that fall outside the surface will be ignored.

Note that `TEE_TUI_SetPNG` decompresses the image into an existing surface.

If the function returns anything other than `TEE_SUCCESS`, the content of the surface on return is undetermined.

**Specification Number:** 55 **Function Number:** 0x010C

#### Parameters

- `surface`: A pointer to the surface in which to draw the image.
- `png`: A pointer to a `TEE_TUIImage` structure containing a PNG image.
- `x,y`: The coordinates of the top left pixel of the image.
- `opacity`: The `TEE_TUI_SURFACE_OPACITY` to apply to the image; see section 4.3.16.

#### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_FORMAT`: If the image is not compliant with PNG format or if the image format is not supported.
- `TEE_ERROR_BAD_STATE`: If the surface is not valid.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

#### Panic Reasons

- If `surface` is `NULL`.
- If `png` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

### 4.6.30 TEE_TUI_TransformSurface

```
TEE_Result TEE_TUI_TransformSurface(
[in]      TEE_TUISurface              *source,
[inout]   TEE_TUISurface              *destination,
[in]      uint32_t                    x_source,
[in]      uint32_t                    y_source,
[in]      uint32_t                    width,
[in]      uint32_t                    height,
[in]      uint32_t                    x_dest,
[in]      uint32_t                    y_dest,
[in]      TEE_TUI_SURFACE_OPACITY     opacity,
[in]      TEE_TUIOperation            operation
);
```

#### Description

The `TEE_TUI_TransformSurface` function takes an image in the input surface and performs the specified transformation on the portion of the image whose top left corner is at `x_source, y_source` with width `width` and height `height`.

The top left corner of the resulting image will be positioned at `x_dest, y_dest` in the output buffer.

The image is cropped to the size of the output buffer. Any portion of the input that lies outside the boundary of the output buffer is unused.

Pixels in the target buffer that are not overwritten by the transformation are unchanged.

When pixels are overwritten, they are blended according to the Alpha channels of the two surfaces and the specified opacity.

The input and output surfaces may overlap either partially or entirely. It is up to the TEE to manage transposing pixels in place. However, a TEE may return a `TEE_ERROR_OUT_OF_MEMORY` error if any part of the input and output surfaces overlap. In this case the contents of both the input and output buffers are undetermined.

**Specification Number:** 55 **Function Number:** 0x010D

#### Parameters

- `source`: The `TEE_TUISurface` to read input data from.
- `destination`: The `TEE_TUISurface` to write data to. This SHALL NOT be the same surface as `source`.
- `x_source`: The x-coordinate in `*source` of the top left hand corner of the data to transform.
- `y_source`: The y-coordinate in `*source` of the top left hand corner of the data to transform.
- `width`: The width in pixels of the portion of the image to transform.
- `height`: The height in pixels of the portion of the image to transform.
- `x_dest`: The x-coordinate in `*destination` of the top left hand corner of the resultant image.
- `y_dest`: The y-coordinate in `*destination` of the top left hand corner of the resultant image.
- `opacity`: Determines how to blend pixels in the source and destination.
- `operation`: The operation to perform on the image. This is a bitwise OR of `TEE_TUI_OPERATION` values.

### Return Value

- `TEE_SUCCESS`: In case of success.
- `TEE_ERROR_BAD_STATE`: If the surface does not exist or has been released.
- `TEE_ERROR_EXTERNAL_CANCEL`: If the operation has been cancelled by an external event which occurred in the REE while the function was in progress.
- `TEE_ERROR_OUT_OF_MEMORY`: If the system ran out of resources.

### Panic Reasons

- If `source` is `NULL` or does not point to a valid surface.
- If `destination` is `NULL` or does not point to a valid surface.
- If `width` is `NULL`.
- If `height` is `NULL`.
- If the Implementation detects any error associated with the execution of this function which is not explicitly associated with a defined return code for this function.

# Annex A Panicked Function Identification

If this specification is used in conjunction with the TEE TA Debug Specification ([TEE TA Debug]), then the specification number is 55 and the values listed in Table A-1 SHALL be associated with the function declared.

[TEE Core API] v1.2 and later will define the functions required for processing events and peripherals.

Functions defined in [TEE Core API] will return debug values from that specification – that is, with specification number 10 and function numbers as defined in [TEE Core API], not the values defined in this document.

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

| Category | Function | Function Number in hexadecimal | Function Number in decimal |
|---|---|---|---|
| Peripheral API | TEE_Peripheral_Close | 0x1E01 | 7681 |
| | TEE_Peripheral_CloseMultiple | 0x1E02 | 7682 |
| | TEE_Peripheral_GetPeripherals | 0x1E03 | 7683 |
| | TEE_Peripheral_GetState | 0x1E04 | 7684 |
| | TEE_Peripheral_GetStateTable | 0x1E05 | 7685 |
| | TEE_Peripheral_Open | 0x1E06 | 7686 |
| | TEE_Peripheral_OpenMultiple | 0x1E07 | 7687 |
| | TEE_Peripheral_Read | 0x1E08 | 7688 |
| | TEE_Peripheral_SetState | 0x1E09 | 7689 |
| | TEE_Peripheral_Write | 0x1E0A | 7690 |
| Event API | TEE_Event_AddSources | 0x1E0B | 7691 |
| | TEE_Event_CancelSources | 0x1E0C | 7692 |
| | TEE_Event_CloseQueue | 0x1E0D | 7693 |
| | TEE_Event_DropSources | 0x1E0E | 7694 |
| | TEE_Event_ListSources | 0x1E0F | 7695 |
| | TEE_Event_OpenQueue | 0x1E10 | 7696 |
| | TEE_Event_Wait | 0x1E11 | 7697 |
| TUI Low-level API | TEE_TUI_BlitDisplaySurface | 0x0101 | 257 |
| | TEE_TUI_CloseSession | 0x0102 | 258 |
| | TEE_TUI_DrawRectangle | 0x0103 | 259 |
| | TEE_TUI_GetDisplayInformation | 0x0104 | 260 |
| | TEE_TUI_GetDisplaySurface | 0x0105 | 261 |
| | TEE_TUI_GetPNGInformation | 0x0106 | 262 |
| | TEE_TUI_GetSurface | 0x0107 | 263 |
| | TEE_TUI_GetSurfaceBuffer | 0x0108 | 264 |
| | TEE_TUI_GetSurfaceInformation | 0x0109 | 265 |

| Category | Function | Function Number in hexadecimal | Function Number in decimal |
|---|---|---|---|
| | TEE_TUI_InitSessionLow | 0x010A | 266 |
| | TEE_TUI_ReleaseSurface | 0x010B | 267 |
| | TEE_TUI_SetPNG | 0x010C | 268 |
| | TEE_TUI_TransformSurface | 0x010D | 269 |

# Annex B TEE TUI Low-level API Usage

The following example code is informative. No guarantee is made as to its quality or correctness.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <memory.h>
#include "tee_internal_api.h"
#include "tee_tui_low_api.h"

#define MAX_BUFFER              (256)
#define TA_GETPERIPHERALS (1)
#define TA_VERSIONFAIL    (2)
#define TA_GETSTATETABLE  (3)
#define TA_FAILBAUDRATE   (4)
#define TA_FAILOPEN       (5)
#define TA_FAILWRITE      (6)
#define TA_OPENFAIL       (7)

int main()
{
uint32_t                  version, timeout, numEvents, j, max, myflag,
                          numSources, numPD;
uint32_t                  *numPeripherals, *myWidth, *myHeight,
                          *myColortype, *numEvents, *dropped;
TEE_EventQueueHandle      *myQueue;
TEE_Peripheral            *myPeripherals;
TEE_PeripheralId           myAccId;      // Id for chosen fingerprint scanner
TEE_PeripheralId           myTouchId;    // Id for Touch event source
TEE_PeripheralId            myTeeId;     // Id for Event Source for TEE
TEE_PeripheralDescriptor   *myAccPD;     // Peripheral Descriptor
TEE_PeripheralDescriptor   *myTouchPD;   // Peripheral Descriptor
TEE_PeripheralDescriptor   *myTeePD;     // Peripheral Descriptor
TEE_PeripheralDescriptor    myPDArray[3]; // Array of Peripheral Decriptors
TEE_EventSourceHandle       myAccESH;    // Handle for accelerometer
TEE_EventSourceHandle       myTouchESH;      // Handle for Touch event source
TEE_EventSourceHandle       myTeeESH;         // Handle for TEE Event Source
TEE_EventSourceHandle       myDeviceArray[3]; // array of event sources
TEE_EventSourceHandle      *mySource;
TEE_TUIDisplay             *myDisplay;
TEE_Tuidisplayinfo         *myDisplayInfo;
TEE_TUISurface             *mySurface, *myDisplaySurface;
TEE_TUISurfaceInfo         *mySurfaceInfo;
TEE_TUIImage               *myPng;
```

```
TEE_Event                   *events;
TEE_Event                    MyTouchEvent;
TEE_Result                   myResult;
size_t                       bytes_in_Peripheral_array = 0;
size_t                       bytes_in_state_table = 0;
TEE_Peripheral               peripherals[1];
TEE_PeripheralState          myPeripheralstate[1];

// Trivial error handling
#define ta_assert(cond, val); if (!(cond)) TEE_Panic(val);

/* NOTE In order to clarify the flow, this code does not include error
   handling routines.
   Actual code would need to deal with the errors thrown by various calls.
*/

/* All structures are version 1 */

version = 1;
numSources = 0;
timeout = 1000;

if (TEE_Event_OpenQueue(&version,
    numSources,
    timeout,
    mySource,
    myQueue ) != TEE_SUCCESS)
    {
    /* Cannot open Event queue */
    };

/* get a list of peripherals */

/* find out how much space is needed for information on the available
peripherals */

  TEE_Peripheral_GetPeripherals(&version, NULL, &bytes_in_Peripheral_array)
  != TEE_ERROR_SHORT_BUFFER

  // defend against strange result
  ta_assert((myResult == TEE_ERROR_SHORT_BUFFER) &&
            (bytes_in_Peripheral_array != 0), TA_GETPERIPHERALS);

  peripherals = TEE_Malloc( bytes_in_Peripheral_array, TEE_MALLOC_FILL_ZERO);

  // fetch the information on the peripherals

  myResult = TEE_Peripheral_GetPeripherals(&version,
                                           peripherals,
                                           &bytes_in_Peripheral_array);
```

```
// check result is sensible
ta_assert((myResult == TEE_SUCCESS) && (&bytes_in_Peripheral_array != 0),
          TA_GETPERIPHERALS);

// assume we only have one display
// if the device has more displays, the TA must determine which to use.
// find touch source for display 1

if (TEE_TUI_GetDisplayInformation(1,
                                  &version,
                                  myDisplayInfo,
                                  myDisplay) != TEE_SUCCESS) {

    /* error obtaining Display Info buffer */
}

max = bytes_in_Peripheral_array / sizeof(TEE_Peripheral);

// loop through all peripherals
// find TEE Pseudo Peripheral
for (uint32_t i = 0; i < max; i++) {
  ta_assert(peripherals[i].version == 1, TA_VERSIONFAIL);
  if (peripherals[i].u.v1.periphType == TEE_PERIPHERAL_OS) {
    myTeePD->id = peripherals[i].u.v1.id;
  }

  // find accelerometer

  if (myPeripherals[i].u.v1.periphType == TEE_PERIPHERAL_ACCELEROMETER) {
      bytes_in_state_table = 0;
      myResult = TEE_Peripheral_GetStateTable(peripherals[i].u.v1.id,
                                              myPeripheralstate,
                                              &bytes_in_state_table);

       ta_assert((myResult == TEE_ERROR_SHORT_BUFFER)
                 && (&bytes_in_state_table != 0), TA_GETSTATETABLE);

      myResult = TEE_Peripheral_GetStateTable(peripherals[i].u.v1.id,
                                              myPeripheralstate,
                                              &bytes_in_state_table);
      ta_assert((myResult == TEE_SUCCESS)
                 && (bytes_in_state_table != 0), TA_GETSTATETABLE);

      /* check if the security indicator of the accelerometer is
      controllable. If it is, add it to the peripherals to control
      and turn on the security indicator */
```

```
        max = bytes_in_state_table / sizeof(TEE_PeripheralState);
        myflag = 0;
        numPD = 2;

        for (uint32_t j = 0; j < max; j++) {
            if ((myPeripheralstate[j].id == TEE_PERIPHERAL_STATE_FLAGS)
                && (myPeripheralstate[j].u.uint32Val
                    & TEE_PERIPHERAL_FLAG_TEE_CONTROLLABLE) ) {
                myflag = TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR;
                myAccPD->id = peripherals[i].u.v1.id;
                numPD = 3;
            };
        };
    }

    // find touch source for display 1

    if (myPeripherals[i].u.v1.periphType == TEE_PERIPHERAL_TOUCH) {
        for (uint32_t j = 0; j < myDisplayInfo->numPeripherals; j++){
            if (myPeripherals[i].u.v1.periphType
                == myDisplayInfo->associatedPeripherals[j]){
                myTouchPD->id = peripherals[i].u.v1.id;
            }
        }
    }
}

    // open the display and peripherals.
    if (myflag == TEE_TUI_INIT_SESSION_LOW_FLAG_REQUIREINDICATOR) {
        TEE_PeripheralDescriptor myPDArray[]= {*myTeePD, *myTouchPD, *myAccPD};
        }
    else {
        TEE_PeripheralDescriptor myPDArray[]= {*myTeePD, *myTouchPD} ;
        }
    myResult = TEE_TUI_InitSessionLow(1, 1000, myflag, myDisplay, numPD,
                                      myPDArray);
    switch (myResult){
        case TEE_SUCCESS:
            myTeeESH =    myPDArray[0].eHandle;
            myAccESH =    myPDArray[1].eHandle;
            myTouchESH =  myPDArray[2].eHandle;
            break;

        case TEE_ERROR_BUSY:
            // Touch event source is in use return an error
            return TEE_ERROR_BUSY;

        default:
            // some other error so assert
            TEE_Panic(myResult);
    }
```

```
 myDeviceArray[0] = myTeeESH;
 myDeviceArray[1] = myTouchESH;
 myDeviceArray[1] = myAccESH;

if (TEE_Event_AddSources(3, myDeviceArray, myQueue) != TEE_SUCCESS)
 {
 /* Error adding sources to queue */
 };

/* From here Touch events for display 1 are captured by the TEE. */

if (TEE_TUI_GetDisplaySurface(myDisplay, myDisplaySurface) != TEE_SUCCESS)
 {
 /* Error opening Display Surface */
 };

if (TEE_TUI_GetSurfaceInformation(&version,
                                  myDisplaySurface,
                                  mySurfaceInfo) != TEE_SUCCESS)
 {
 /* Error obtaining Surface information */
 };
/* Get a new surface to use to manipulate the image */
if (TEE_TUI_GetSurface(myDisplayInfo->pixelWidth,
                       myDisplayInfo->pixelHeight,
                       mySurfaceInfo->u.v1.stride,
                       mySurface) != TEE_SUCCESS)
 {
 /* error obtaining working surface */
 };
```

```
  /* Assume we have copied an image into myPng */
  /* Obtain the size of the image */

  if (TEE_TUI_GetPNGInformation(myPng,
                                myWidth,
                                myHeight,
                                myColortype) != TEE_SUCCESS)
    {
    /* error obtaining information from image */
    };
 /* write the image to a surface */
  if (TEE_TUI_SetPNG(mySurface, myPng, 0, 0, myWidth, myHeight,
      TEE_TUI_SURFACE_OPACITY_OPAQUE) != TEE_SUCCESS)
    {
    /* error writing PNG to surface */
    };

  /* Invert the colors in the PNG -
     writing the output to the surface belonging to the display */

  if (TEE_TUI_TransformSurface(mySurface,
                               myDisplaySurface,
                               0, 0,
                               *myWidth,
                               *myHeight,
                               0, 0,
                               TEE_TUI_SURFACE_OPACITY_OPAQUE,
                               TEE_TUI_OPERATION_INVERTRED |
                               TEE_TUI_OPERATION_INVERTBLUE |
                               TEE_TUI_OPERATION_INVERTGREEN)
                               != TEE_SUCCESS)
    {
    /* error transforming surface */
    };
  /* Call function to update screen */
  if (TEE_TUI_BlitDisplaySurface(myDisplay, mySurface) != TEE_SUCCESS)
    {
    /* error updating screen */
    };
  /* Wait for an event */
  *numEvents = 1;
  if (TEE_Event_Wait(myQueue,
                     1000,
                     events,
                     numEvents,
                     dropped) != TEE_SUCCESS)
    {
    /* deal with error */
    }
  else
    {
```

```
    /* deal with input */
    /* events contains one touch event */
    memcpy(&MyTouchEvent,
           events[0].u.v1.payload,
           sizeof(MyTouchEvent));

    /* MyTouchEvent contains a touch */
    /* Add code to deal with touch here */
    };

  /* Tidy up */
  /* Close the TUI session */

  TEE_TUI_CloseSession();

  /* REE can now access the display */

  /* Close the Queue */

  TEE_Event_CloseQueue(myQueue);

  /* Events now go to the REE */
  /* Free mysurface */

  TEE_TUI_ReleaseSurface(mySurface);
}
```