GlobalPlatform Device Technology
TEE Sockets API Specification
Version 1.0.1

Public Release
January 2017
Document Reference: GPD_SPE_100
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 ................................................................................................................................. 5
1.1 Audience ................................................................................................................................. 5
1.2 IPR Disclaimer ....................................................................................................................... 5
1.3 References .............................................................................................................................. 5
1.4 Terminology and Definitions ................................................................................................. 6
1.5 Abbreviations and Notations ................................................................................................. 7
1.6 Revision History .................................................................................................................... 7

2 Background ................................................................................................................................. 8

3 Requirements for TEE Sockets API .......................................................................................... 9
3.1 Assumptions and Scope .......................................................................................................... 9
3.1.1 Streams and Boundary of Trust ..................................................................................... 10
3.2 Implementation of Specific Protocols .................................................................................. 11
3.3 Layered Connections ........................................................................................................... 11

4 General Information .................................................................................................................. 12
4.1 Error Handling ....................................................................................................................... 12
4.2 Specification Version Number Property ............................................................................... 13
4.3 Protocol Identifier ................................................................................................................ 13
4.4 Panicked Function Identification ........................................................................................ 14

5 TEE_iSocket (The Generic Interface Socket) API .................................................................... 15
5.1 Header File Name .................................................................................................................. 15
5.2 Functionality ........................................................................................................................ 15
  5.2.1 TEE_iSocketHandle and Setup ..................................................................................... 17
  5.2.2 Fatal Errors .................................................................................................................... 17
  5.2.3 Timely Manner ................................................................................................................ 18
  5.2.4 Open ............................................................................................................................... 18
  5.2.5 Close .............................................................................................................................. 20
  5.2.6 Send ............................................................................................................................... 22
  5.2.7Recv ............................................................................................................................... 25
  5.2.8 Error .............................................................................................................................. 28
  5.2.9 ioctl .............................................................................................................................. 29
5.3 Global commandCode Definitions for ioctl .......................................................................... 31
5.4 Example of a TEE_iSocket Protocol Implementation ........................................................... 32
  5.4.1 The Header File ............................................................................................................. 32
  5.4.2 The C Implementation File ........................................................................................... 33

Note: Annexes to this specification are provided as separate documents. See section 3.2.
Tables

Table 1-1: Normative References......................................................................................................................5
Table 1-2: Informative References ..................................................................................................................6
Table 1-3: Terminology and Definitions .............................................................................................................6
Table 1-4: Abbreviations and Notations ............................................................................................................7
Table 1-5: Revision History ...............................................................................................................................7
Table 4-1: Error Codes Specific to TEE Sockets API..........................................................................................12
Table 4-2: Specification Version Number Property – 32-bit Integer Structure................................................13
Table 4-3: Protocol Identifier Ranges ..............................................................................................................13
Table 4-4: Function Identification Values ........................................................................................................14
Table 5-1: Structure of the commandCode Parameter ..................................................................................29
Table 5-2: Examples of commandCode Interpretations ...................................................................................29

Figures

Figure 3-1: TEE iSocket API Exposes Network Client Capabilities to TA........................................................9
Figure 3-2: Separation of Security Protocols and Pure Transport Protocols .................................................11
1 Introduction

This document, the GlobalPlatform TEE Sockets API Specification, specifies:

- The generic C interface used by a Trusted Application (TA) to establish and utilize network communications to a remote server using a socket style approach
- Generic error behavior of the functions

This general specification does not specify any particular protocol nor the data structures used to utilize a particular protocol. Each specific protocol, such as TCP and UDP, is described in detail in a separate Annex to this general specification.

1.1 Audience

This document is suitable for software developers implementing Trusted Applications running inside the Trusted Execution Environment (TEE) which need to make socket networking calls.

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

1.2 IPR Disclaimer

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

1.3 References

Table 1-1: Normative References

<table>
<thead>
<tr>
<th>Standard / Specification</th>
<th>Description</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPD_SPE_007</td>
<td>GlobalPlatform Device Technology TEE Client API Specification</td>
<td>[TEE Client]</td>
</tr>
<tr>
<td>GPD_SPE_009</td>
<td>GlobalPlatform Device Technology TEE System Architecture</td>
<td>[TEE Sys Arch]</td>
</tr>
<tr>
<td>GPD_SPE_010</td>
<td>GlobalPlatform Device Technology TEE Internal Core API Specification</td>
<td>[TEE Core]</td>
</tr>
<tr>
<td>GPD_SPE_025</td>
<td>GlobalPlatform Device Technology TEE TA Debug Specification</td>
<td>[TEE Debug]</td>
</tr>
<tr>
<td>Standard / Specification</td>
<td>Description</td>
<td>Ref</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>GPD_SPE_104</td>
<td>TEE Sockets API Specification Annex D: Example of Using TEE Sockets API Specification</td>
<td>[Socket Example]</td>
</tr>
<tr>
<td>IEEE Std 1003.1-2008</td>
<td>The Open Group Base Specifications Issue 7 (referred to as &quot;the POSIX standard&quot;)</td>
<td>[POSIX]</td>
</tr>
<tr>
<td>RFC 5246</td>
<td>The Transport Layer Security (TLS) Protocol</td>
<td>[TLS]</td>
</tr>
<tr>
<td>RFC 793 / RFC 791</td>
<td>Transmission Control Protocol and Internet Protocol (referred to as TCP/IP)</td>
<td>[RFC 793]</td>
</tr>
<tr>
<td>RFC 768 / RFC 791</td>
<td>User Datagram Protocol and Internet Protocol (referred to as UDP/IP)</td>
<td>[RFC 768]</td>
</tr>
<tr>
<td>RFC 2119</td>
<td>Key words for use in RFCs to Indicate Requirement Levels.</td>
<td>[RFC 2119]</td>
</tr>
</tbody>
</table>

Table 1-2: Informative References

<table>
<thead>
<tr>
<th>Standard / Specification</th>
<th>Description</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC 7498-1</td>
<td>Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model (referred to as “the OSI model”)</td>
<td>[ISO 7498]</td>
</tr>
<tr>
<td>IPsec</td>
<td>Base architecture for IPsec compliant systems.</td>
<td>[RFC 2401]</td>
</tr>
<tr>
<td>IKE</td>
<td>Describes version 2 of the Internet Key Exchange (IKE) protocol. IKE is a component of IPsec used for performing mutual authentication and establishing and maintaining Security Associations (SAs).</td>
<td>[RFC 5996]</td>
</tr>
</tbody>
</table>

### 1.4 Terminology and Definitions

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

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

Selected technical terms used in this document are included in Table 1-3. Additional technical terms are defined in TEE Internal Core API Specification [TEE Core].

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSocket</td>
<td>Interface Socket</td>
</tr>
</tbody>
</table>
1.5 Abbreviations and Notations

Selected abbreviations and notations used in this document are included in Table 1-4. Additional abbreviations and notations are defined in [TEE Core].

Table 1-4: Abbreviations and Notations

<table>
<thead>
<tr>
<th>Abbreviation / Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>IKE</td>
<td>Internet Key Exchange</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPsec</td>
<td>Internet Protocol Security</td>
</tr>
<tr>
<td>PSK</td>
<td>Pre-Shared Key</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
</tbody>
</table>

1.6 Revision History

Table 1-5: Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 2015</td>
<td>1.0</td>
<td>Public Release</td>
</tr>
<tr>
<td>January 2017</td>
<td>1.0.1</td>
<td>Public Release showing all non-trivial changes since v1.0. Changes include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added one error code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Expanded explanation of error handling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Redefined TEE_iSocketHandle structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Defined the open function as cancellable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clarified description of close, send, and recv functions</td>
</tr>
</tbody>
</table>
2 Background

TAs are commonly required to communicate with remote servers. Such communication is today typically performed by using the TEE Client API [TEE Client] to route Trusted Application (TA) data to or from a Client Application (CA), and the CA then manages a sockets interface in the Rich Execution Environment (REE) to the remote server.

Creating numerous TAs, all performing the same underlying tasks increases code size, risks, and effort and so it is seen as desirable to provide this common functionality inside the TEE.

Communication is normally depicted as layers of protocols (e.g. the OSI layer model [ISO 7498]), each performing specific tasks of the communication. For example, the Internet Protocol (IP) is a connectionless protocol, which enables network nodes to send messages across network boundaries to other nodes using the IP address. This protocol does not guarantee in-order delivery nor does it provide retransmission facilities in case of corrupted messages. To provide such features, the Transmission Control Protocol (TCP) can be utilized. On top of TCP we may choose to add encryption of the messages, and hence a new layer must be used, for example the Transport Layer Security (TLS) protocol. All these different layers basically provide the same functions to the user: open a connection, send and receive data, and close the connection.

Of course, there are many means of communication other than the Internet type of protocols. Modern devices often include short-range communication such as Bluetooth and near field communication but essentially, all the user needs is a set of functions performing the basic communication tasks.

From this point of view, GlobalPlatform has created a specification that is small and easily layered as the needs of the user change and new communication media are added to the device.

This version of the TEE Sockets API targets Client functionality. The API specified enables a TA developer to include basic network client functionality in a Trusted Application.
3 Requirements for TEE Sockets API

3.1 Assumptions and Scope

The intent of this specification is to provide a common homogeneous interface for the TA to communicate with other network nodes. The POSIX Sockets interface [POSIX], as provided by many operating systems, has limitations when it comes to layering different protocols. This specification modularizes the interface to provide a more flexible solution while sustaining the simplicity of the POSIX interface.

The modular approach also provides a simpler path to upgradable security and additional protocols. Each protocol is handled by an instance of an “Interface Socket” (iSocket), which can be plugged in on top of another iSocket to add protocol layers.

In the current version, this API defines mechanisms that allow the TA to act as a network client (i.e. connecting to external servers via the network).

Figure 3-1: TEE iSocket API Exposes Network Client Capabilities to TA
3.1.1 Streams and Boundary of Trust

The fundamental operation that a network connection performs is to transfer a bi-directional stream of data bytes (or packets) between entities that are located anywhere in a network.

The code that performs this transport functionality is distributed across the network and hence is not under the control of the TEE and cannot be trusted. This means that the data received from the network cannot be trusted and software executing in the TEE must treat it as such.

There is no reason why the network hardware and software (i.e. that which implements low level transport mechanisms such as TCP/IP and similar) should execute within the TEE\(^1\): to do so would not improve the trusted status of the data (since the data would still originate outside the TEE) and would add significant complexity to the system.

Network software that changes the level of trust of the data stream SHALL execute within the TEE. Other network software MAY execute within the TEE. For example, the implementation of the SSL/TLS algorithm ensures that the data received is that sent by the originator and hence improves its level of trust. Thus, TLS (or any other security protocol) SHALL be implemented within the Trusted OS component, while pure transport protocols, which do not change the level of trust of the data stream, MAY be implemented in the REE. See for example Figure 3-2.

If we remove the mechanics of network transport from the TEE all that remains is a mechanism for transport of streams of bytes into and out of TAs. Such a mechanism can be used for purposes other than networking (e.g. DRM data handling) and as such is defined in a non-networking specific manner.

The only part of the design that is network specific is stream setup, where network addresses and protocols must be specified.

Two types of stream are considered:

- A byte stream where the data is described as a stream of bytes (e.g. TCP/IP [RFC 793])
- A message stream where the data is described as a stream of self-contained messages defined by the network transfer mechanism (e.g. UDP/IP [RFC 768]), hereafter referred to as datagrams

Both of these stream types are unprotected and additional layers are needed to provide confidentiality and/or integrity. An example of a protocol that provides this is TLS [TLS].

\(^1\) There is an exception to this, which is when IPsec is used to protect the transferred data. Because IPsec is embedded within the IP framework, it is necessary for the whole IP stack to be embedded within the TEE. It is also necessary to embed the Internet Key Exchange (IKE) protocol implementation. This is a large volume of code, so support for IPsec is not specified currently.
3.2 Implementation of Specific Protocols

This specification provides the general API for accessing and handling client sockets of various kinds. For each specific protocol that GlobalPlatform decides to support, an Annex document to this specification will be created, in which all the protocol specific details are specified. In one sense, the current specification can be viewed as a virtual class definition, of which the specific protocol specifications are instances. Each specific protocol (or sometimes a group of closely related protocols) will have its own header file in addition to the generic one defining the general API.

This specification does not provide support for configuring carrier layer connections and hence, an REE application (or a proprietary TEE interface) may have to instigate a session before the TEE TA can start a socket layer protocol session.

This API does not include establishment of a transport layer. This is covered in an iSocket instance defined in the Annexes to this document.

3.3 Layered Connections

This specification is intended to provide an easy way for protocols to be layered. This means that, for example, a security protocol such as TLS can be layered on top of a transport protocol such as TCP. To decouple the different layers, the developer is free to choose the transport layer as appropriate. In a layered configuration, each protocol is opened and set up individually, starting at the lowest layer. Each protocol is also closed individually, starting at the highest layer. It is only during send and receive that the higher layers utilize the lower layers in order to send and receive data packages. Stacks of socket layers are opened in low to high order, and closed in high to low order. The highest layer is the layer that the TA will mainly interact with, so in the case of TCP/TLS the high layer is TLS. Once the layering is established it is expected the TA developer will interact with the highest layer, unless there is an error.
4 General Information

4.1 Error Handling

This API follows the TEE Internal Core API philosophy that any programmer avoidable error results in a TA level Panic. See section 2.2 of [TEE Core].

In addition to the error codes defined in [TEE Core], the following error codes are used throughout this API.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_ISOCKET_ERROR_PROTOCOL</td>
<td>0xF1007001</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_REMOTE_CLOSED</td>
<td>0xF1007002</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_TIMEOUT</td>
<td>0xF1007003</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_OUT_OF_RESOURCES</td>
<td>0xF1007004</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_LARGE_BUFFER</td>
<td>0xF1007005</td>
</tr>
<tr>
<td>TEE_ISOCKET_WARNING_PROTOCOL</td>
<td>0xF1007006</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_HOSTNAME</td>
<td>0xF1007007</td>
</tr>
</tbody>
</table>

Since protocols can be layered, and each protocol may have specific error codes that are only valid for that protocol, a generic TEE_ISOCKET_ERROR_PROTOCOL and TEE_ISOCKET_WARNING_PROTOCOL error code is defined. The error function defined in section 5.2.8 can be used to retrieve the specific error or warning from the last operation. All protocol specific errors SHALL be defined in the instance header file and SHALL follow the general guidelines in [TEE Core].

As described in section 3.3, Layered Connections, the TA developer is expected to interact with the highest layer unless there is an error. If a TEE_ISOCKET_ERROR_PROTOCOL is returned from any function, lower layer investigation may be required. Calling the error function on a context that has returned a TEE_ISOCKET_ERROR_PROTOCOL may return TEE_SUCCESS if the error code was propagated and not caused in that context. Only the context causing a TEE_ISOCKET_ERROR_PROTOCOL shall store the protocol specific error. If the TA encounters a protocol specific error at the top context, it may use the error function to investigate each open context in order to determine which context caused the error. See [Socket Example][Socket Example] for an error handling example.
4.2 Specification Version Number Property

This specification defines a TEE property containing the version number of the specification the implementation conforms to. The property can be retrieved using the normal Property Access Functions defined in [TEE Core]. The property SHALL be named “gpd.teesockets.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 supported specification is not a maintenance version, this byte SHALL be set to zero.
- The least significant byte is reserved for future use. Currently this byte SHALL be set to zero.

<table>
<thead>
<tr>
<th>Bits 24-31 (MSB)</th>
<th>Bits 16-23</th>
<th>Bits 8-15</th>
<th>Bits 0-7 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major version number of the specification.</td>
<td>Minor version number of the specification.</td>
<td>Maintenance version number of the specification. If not a maintenance version, SHALL be set to zero.</td>
<td>Reserved for future use. Currently SHALL be set to zero.</td>
</tr>
</tbody>
</table>

The specification version number is also defined in the iSocket interface (described in section 5.2), stored in the 32-bit integer TEE_iSocketVersion, interpreted as described in Table 4-2. For a specific protocol, TEE_iSocketVersion does not need to have the same value as the property gpd.teesockets.version. A lower version number is acceptable and is interpreted as indicating that the specific protocol uses an earlier version of the iSocket interface.

4.3 Protocol Identifier

Each specific protocol will have a unique protocol identifier. This identifier is a byte and the value is specified by GlobalPlatform. This identifier is used to identify the protocol to which the TEE_iSocket structure belongs. Table 4-3 specifies the protocol identifier ranges.

<table>
<thead>
<tr>
<th>Protocol Identifier Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 – 0x63 (0 – 99)</td>
<td>Reserved protocol identifier.</td>
</tr>
<tr>
<td>0x64 – 0xFF (100 – 255)</td>
<td>Protocol identifiers assigned by GlobalPlatform. The protocol description associated with each value can be found in Annexes to this document.</td>
</tr>
</tbody>
</table>
4.4 Panicked Function Identification

The TEE TA Debug Specification [TEE Debug] mandates that each API function be assigned a specification number and a function number to be reported through the PRM and Debug functionality. Since TEE iSocket API is divided into this general API description and Annexes containing the protocol specifics, the functions defined in this specification are assigned only a function number. The specification number can be derived from the Annex document reference number.

Example

The open function of this specification is assigned the function number 0x101. If the TLS protocol is defined in the Annex with document reference number “GPD_SPE_103”, then the Panic identifier of the TLS open function will be: Specification Number: 103, Function Number: 0x101

If this specification is used in conjunction with the TEE TA Debug Specification ([TEE Debug]), then the specification number is based on the Annex that specifies the protocol in use (as described above) and the values listed in Table 4-4 SHALL be associated with the function declared.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Function Number in hexadecimal</th>
<th>Function Number in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Socket</td>
<td>open</td>
<td>0x101</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>close</td>
<td>0x102</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>send</td>
<td>0x103</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>recv</td>
<td>0x104</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>0x105</td>
<td>261</td>
</tr>
<tr>
<td></td>
<td>ioctl</td>
<td>0x106</td>
<td>262</td>
</tr>
</tbody>
</table>
5 TEE_iSocket (The Generic Interface Socket) API

The TEE_iSocket API is designed to supply a generic table of functions regardless of networking layer or client functionality and hence, the core of the API is a structure, containing function pointers.

5.1 Header File Name

The header file of the TEE_iSocket interface SHALL be named “tee_isocket.h”.

5.2 Functionality

The basic TEE_iSocket interface consists of six functions; open, close, send, recv, error, and ioctl. Implementations may add other fields provided that the structure begins with the field names, order, and types as specified here. Implementers of proprietary extensions should note that future versions of this specification may require additional fields as part of the standard initial segment.

The declaration can be seen below.
typedef const struct TEE_iSocket_s
{
    uint32_t   TEE_iSocketVersion;
    uint8_t     protocolID;

    TEE_Result (* open)(
        TEE_iSocketHandle *ctx,
        void *setup,
        uint32_t *protocolError );

    TEE_Result (* close)(
        TEE_iSocketHandle ctx );

    TEE_Result (* send)(
        TEE_iSocketHandle ctx,
        const void *buf,
        uint32_t *length,
        uint32_t timeout );

    TEE_Result (* recv)(
        TEE_iSocketHandle ctx,
        void *buf,
        uint32_t *length,
        uint32_t timeout );

    uint32_t (* error) (  
        TEE_iSocketHandle ctx );

    TEE_Result (* ioctl)(
        TEE_iSocketHandle ctx,
        uint32_t commandCode,
        const void *buf,
        uint32_t *length );
} TEE_iSocket;

The structure contains constant function pointers that will implement the specifics of one particular layer of the communication stack. For a specific supported protocol layer, the Implementation SHOULD provide the following in the instance specific header file:

- A declaration of the corresponding setup structure as described in section 5.2.1.
- A declaration of a pointer to the specific instance of the TEE_iSocket.
5.2.1 TEE_iSocketHandle and Setup

The context that needs to be maintained varies for each connection and protocol, so a generic opaque type TEE_iSocketHandle is used. This is defined as a pointer to an implementation defined structure:

```c
typedef struct void TEE_iSocketHandle * TEE_iSocketHandle;
```

In a specific protocol implementation, the handle must keep all data that is needed to maintain the connection from the time it is opened until it is closed.

The setup parameter given to open is also protocol specific and must contain all the data to open the connection. For example, a TCP/IP socket must know the address and port of the server to connect to. The setup parameter must be of a pointer to structure type, known as the setup structure. If no setup data is necessary for a particular protocol, the value of the setup parameter SHALL be NULL.

If a protocol is intended to be “on top” of a transporting protocol, for example such as TLS over TCP, the setup structure SHALL contain a pointer to the underlying TEE_iSocket and the instance handle.

5.2.2 Fatal Errors

Errors and the protocol specific errors returned by the functions of the TEE sockets interface can be “fatal” or “non-fatal”.

- If the open function returns anything other than TEE_SUCCESS, then the TEE_iSocketHandle equals TEE_HANDLE_NULL. The only valid operation on TEE_HANDLE_NULL is close.

- When a socket returns a fatal error in response to any other function, the only valid operations on that TEE_iSocketHandle instance are close and error.

- The return values table of each function description specifies whether each error is fatal.

- As TEE Sockets API structures are layered, it is recommended that if the lower layer returns a fatal error, then so should the current layer. This way the TA can be alerted of the error and can manage it accordingly. See section 3.3, Layered Connections, for a description of the layered socket structure.

5.2.3 Timely Manner

A number of actions in this API must be performed in a timely manner. It is implementation defined as to what this may be but the implementation should take into account any defined timeouts of the protocol in question.
5.2.4 Open

```c
TEE_Result (* open) (  
    [out] TEE_iSocketHandle *ctx,  
    [in] void *setup,  
    [out] uint32_t *protocolError  
);
```

### Description

The `open` function tries to open the connection according to the prerequisites passed in the `setup` parameter (see section 5.2.1). If the connection is successful, it returns a handle to the socket in the `ctx` parameter. If the connection is not successful, the `ctx` parameter has the value `TEE_HANDLE_NULL` after this function has been successfully called, any changes to the `setup` parameter SHALL NOT alter the behavior of the protocol in subsequent calls to the instance `TEE_iSocket` functions. In other words, the functionality and behavior of the protocol instance after a successful `open` function call SHALL only depend on the handle.

The `open` function call is blocking and should execute in a timely manner (see section 5.2.3, Timely Manner), which may vary depending on the communication channel and protocol. If the connection cannot be opened in a time frame appropriate to the protocol, the function returns `TEE_ISOCKET_ERROR_TIMEOUT`. If a configurable timeout is appropriate it MAY be specified in the setup structure.

This function is cancellable; i.e. if the current task’s cancelled flag is set and the TA has unmasked the effects of cancellation, then this function returns earlier than the requested action with the return code `TEE_ERROR_CANCEL`. If an annex defines a non-blocking sockets layer, then this function need not be cancellable.

Protocol specific errors are returned by returning the `TEE_ISOCKET_ERROR_PROTOCOL` or `TEE_ISOCKET_WARNING_PROTOCOL` and updating the `protocolError` parameter with the protocol-specific error code. If the return code is not `TEE_ISOCKET_ERROR_PROTOCOL` or `TEE_ISOCKET_WARNING_PROTOCOL` the contents of `protocolError` will not be changed.

In a layered configuration, each layer must be allocated and initialized starting from the lowest layer, i.e. the `open` function must be called for each layer, using the returned handle in the setup of the next layer.

### Specification Number: See section 4.4  
### Function Number: 0x101

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle *ctx</td>
<td>Handle to an implementation specific context.</td>
</tr>
<tr>
<td>void *setup</td>
<td>Protocol specific setup parameter.</td>
</tr>
<tr>
<td>uint32_t *protocolError</td>
<td>In case of a protocol specific error or warning, this parameter holds the error value. See specific protocol Annex specifications for the possible values.</td>
</tr>
</tbody>
</table>
Return Values

See section 5.2.2 for the meaning of fatal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>In case of success.</td>
</tr>
<tr>
<td>TEE_ERROR_CANCEL</td>
<td>No</td>
<td>The TA has unmasked the effects of cancellation and the Client Application has set the task cancelled flag.</td>
</tr>
<tr>
<td>TEE_ERROR_OUT_OF_MEMORY</td>
<td>Yes</td>
<td>Failed to allocate memory for the socket handle.</td>
</tr>
<tr>
<td>TEE_ERROR_BAD_PARAMETERS</td>
<td>Yes</td>
<td>Error in the setup parameter. The requested setup cannot be achieved.</td>
</tr>
<tr>
<td>TEE_SOCKET_ERROR_OUT_OF_RESOURCES</td>
<td>Yes</td>
<td>Failed to allocate resources for the socket.</td>
</tr>
<tr>
<td>TEE_SOCKET_ERROR_TIMEOUT</td>
<td>No</td>
<td>The connection cannot be opened in a timely manner.</td>
</tr>
<tr>
<td>TEE_ERROR_COMMUNICATION</td>
<td>No</td>
<td>There is no route to the requested host or the host did not accept the connection.</td>
</tr>
<tr>
<td>TEE_SOCKET_ERROR_PROTOCOL</td>
<td>Yes</td>
<td>Protocol specific error. Occurs under conditions defined in the relevant Annex. The error value is returned in the *protocolError parameter.</td>
</tr>
<tr>
<td>TEE_SOCKET_WARNING_PROTOCOL</td>
<td>No</td>
<td>Occurs under conditions defined in the relevant Annex. The error value is returned in the *protocolError parameter.</td>
</tr>
</tbody>
</table>

If a value of TEE_SUCCESS is not returned, then the socket will not have been opened and so the fatal state of these errors (as defined in section 5.2.2) is irrelevant.

Panic Reasons

The open function SHALL panic if the following occurs:

- Any parameter is NULL.

The open function MAY panic if the following occurs:

- The setup parameter pointer does not point to a valid instance specific setup type.
- If the Implementation detects any error which cannot be represented by any defined or Implementation defined error code.
### 5.2.5 Close

```c
TEE_Result (* close) (     
    TEE_iSocketHandle ctx,  
);
```

**Description**

The `close` function closes the socket, de-allocates the context, and frees any resources held by the handle, after which the handle is not valid anymore. **Calling this function with an already closed context is a programming error and will cause a Panic.** Calling this function with a `TEE_HANDLE_NULL` parameter will return `TEE_SUCCESS`.

The `close` function call is blocking and should execute in a timely manner. In a layered configuration, the layers must be closed starting from the highest layer. Higher layers SHALL NOT close underlying layers in this function.

The handle `ctx` should be considered to be in a state where the only valid operation on it is `close` or `error`.

If the `close` function cannot close the context in a timely manner, it SHALL return `TEE_ISOCKET_ERROR_TIMEOUT`. The handle `ctx` should be considered to be in a state where the only valid operation on it is `close` or `error`.

If the `close` function encounters an error specific to the protocol, the return value SHALL be `TEE_ISOCKET_ERROR_PROTOCOL`.

The `close` function may be called on an already closed handle. If it is, the error state SHALL be retained and the same return value SHALL be returned each time.

**Specification Number:** See section 4.4  
**Function Number:** 0x102

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle ctx</td>
<td>Initialized implementation specific handle or <code>TEE_HANDLE_NULL</code>.</td>
</tr>
</tbody>
</table>

**Return Values**

See section 5.2.2 for the meaning of `fatal`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>In case of success. The <code>ctx</code> handle may be invalid or already closed.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_TIMEOUT</td>
<td>Yes</td>
<td>The connection could not be closed in a timely manner. A timeout may be caused by a remote party that failed to receive all data or failed to transmit all data in a timely manner.</td>
</tr>
<tr>
<td>TEE_ERROR_COMMUNICATION</td>
<td>Yes</td>
<td>The route to the host is down or an internal network interface is down.</td>
</tr>
<tr>
<td>Name</td>
<td>Fatal</td>
<td>Reason</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_REMOTE_CLOSED</td>
<td>Yes</td>
<td>The remote host has closed the connection. Some instances will never return this error due to statelessness. TEE_ISOCKET_ERROR_REMOTE_CLOSED is possible only if a prior data exchange was reported as successful but the connection was forcibly closed by the remote host.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_PROTOCOL</td>
<td>Yes</td>
<td>Protocol specific error. See the error function for further information. All protocol specific errors returned from close are fatal errors.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_HOSTNAME</td>
<td>Yes</td>
<td>Unable to reach the requested remote host.</td>
</tr>
</tbody>
</table>

**Panic Reasons**

The close function SHALL panic if either of the following occurs:

- The handle is not initialized.
- The handle is not a valid handle of the specific protocol. TEE_HANDLE_NULL is a valid handle.
- The Implementation detects any error which cannot be represented by any defined or implementation defined error code.
5.2.6 Send

```
TEE_Result (* send) (  
    TEE_iSocketHandle ctx,  
    [in]  const void  *buf,  
    [inout] uint32_t  *length,  
    uint32_t  timeout
);
```

Description

The `send` function transmits data to the remote network node. If `TEE_iSocket` instances are layered, the `send` function of the topmost protocol calls the `send` function of its lower layer after processing the message according to its protocol.

The parameter `buf` points to a memory region containing the bytes to be transmitted. The number of bytes to be sent is passed in `*length`. Unless `*length` is zero, on return, `*length` is updated with the actual number of bytes transmitted or passed on to the next lower layer in the stack.

The `send` function is a blocking function with a timeout. It tries to send the buffer (or at least propagate the data to lower level buffers) for a specific amount of time. After it sends or propagates the data, or after the specified time elapses (whichever comes first), it unconditionally returns. If no data was sent, it returns the error code `TEE_ISOCKET_ERROR_TIMEOUT`. If some data was transmitted, it returns `TEE_SUCCESS` and the `*length` parameter SHALL be updated with the number of successfully sent bytes. Note that a successful transmission does not guarantee that the remote node has received the data, unless the protocol makes specific guarantees. An error in transmission may be reported as a fatal error in a subsequent call to `send`, `recv`, `ioctl`, or `close`.

For a partially sent message, the `send` function SHALL NOT buffer the remaining bytes. It is the responsibility of the user to re-send those bytes.

For protocols that operate on a data stream, the `send` function SHOULD return as soon as at least part of the buffer has been sent. For protocols that operate on blocks of data, such as some encrypted protocols, it is the user's responsibility to provide large enough chunks of data to the `send` function in order to minimize fragmentation. It is protocol specific and implementation specific whether the `send` function internally buffers data in order to avoid fragmentation and small chunks.

For datagram-based protocols, the `send` function SHALL map one call to `send` to one datagram. If the buffer to transmit is too large for a single datagram, the `send` function returns `TEE_ISOCKET_ERROR_LARGE_BUFFER`, and sets the `*length` parameter to the maximum number of bytes that the protocol datagram can send. Furthermore, the `send` function SHALL NOT transmit any of the bytes in the buffer, when the error `TEE_ISOCKET_ERROR_LARGE_BUFFER` is returned.

For protocols that operate on a data stream, the `send` function SHOULD return as soon as at least part of the buffer has been sent. For protocols that operate on blocks of data, such as some encrypted protocols, it is the user's responsibility to provide large enough chunks of data to the `send` function in order to minimize fragmentation. It is protocol specific and implementation specific whether the `send` function internally buffers data in order to avoid fragmentation and small chunks.

The `send` function is a cancellable function. If a cancel request is received during execution of the `send` function, the implementation SHALL behave as if the timeout has been reached but the return code SHALL be `TEE_ERROR_CANCEL` and SHALL update the `*length` with the number of received bytes. If the transmission was completed, the return code SHALL be `TEE_SUCCESS`.
Specification Number: See section 4.4    Function Number: 0x103

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle ctx</td>
<td>Initialized implementation specific handle.</td>
</tr>
<tr>
<td>const void *buf</td>
<td>The buffer containing the data to be sent or passed on to the next lower layer in the stack, unless *length is zero.</td>
</tr>
<tr>
<td>uint32_t *length</td>
<td>A pointer to the length in bytes of the buffer to be sent. On return, this parameter is updated with the number of bytes sent.</td>
</tr>
<tr>
<td>uint32_t timeout</td>
<td>The timeout of the operation in milliseconds. If timeout == 0, the function makes one attempt to send data and returns immediately. If timeout == TEE_TIMEOUT_INFINITE (defined in [TEE Core] TEE_Wait), the function blocks until all bytes are sent or a fatal error occurs.</td>
</tr>
</tbody>
</table>

Return Values

See section 5.2.2 for the meaning of fatal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>In case of success.</td>
</tr>
<tr>
<td>TEE_ERROR_CANCEL</td>
<td>No</td>
<td>The function was cancelled.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_TIMEOUT</td>
<td>No</td>
<td>The complete buffer could not be sent during the specified timeout period.</td>
</tr>
<tr>
<td>TEE_ERROR_COMMUNICATION</td>
<td>No</td>
<td>The route to the host is down or an internal network interface is down.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_REMOTE_CLOSED</td>
<td>Yes</td>
<td>The remote host has closed the connection. Some instances will never return this error due to statelessness.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_PROTOCOL</td>
<td>Yes</td>
<td>Protocol specific error. See the error function for further information. Each protocol specific error is defined in the corresponding Annex. Underlying protocols can be tested for errors recursively by applying the struct_setup (see the Annexes) underlying ctx to the error function. The error can be parsed individually depending on the ProtocolID from the underlying socket.</td>
</tr>
<tr>
<td>Name</td>
<td>Fatal</td>
<td>Reason</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TEE_SOCKET_WARNING_PROTOCOL</td>
<td>No</td>
<td>Protocol specific warning. See the error function for further information. Each protocol specific warning is defined in the corresponding Annex. Underlying protocols can be tested for warnings recursively by applying the struct_setup (see the Annexes) underlying ctx to the error function. The warning can be parsed individually depending on the ProtocolID from the underlying socket.</td>
</tr>
<tr>
<td>TEE_SOCKET_ERROR_LARGE_BUFFER</td>
<td>No</td>
<td>The protocol is datagram-based and the buffer is too large to be sent in one datagram. No bytes from the buffer are sent.</td>
</tr>
<tr>
<td>TEE_SOCKET_ERROR_OUT_OF_RESOURCES</td>
<td>Yes</td>
<td>Failed to allocate resources for the operation.</td>
</tr>
</tbody>
</table>

**Panic Reasons**

The `send` function SHALL panic if any of the following occurs:

- The handle is not initialized or is `NULL`.
- The handle is not a valid handle of the specific protocol.
- The parameter `buf == NULL`.
- The parameter `length == NULL`.
- The Implementation detects any error which cannot be represented by any defined or Implementation defined error code.
5.2.7 Recv

```c
TEE_Result (* recv) (
    TEE_iSocketHandle    ctx,
    [out]   void    *buf,
    [inout] uint32_t    *length,
    uint32_t     timeout
);
```

**Description**

The `recv` function receives data from the remote network node. If TEE_iSocket instances are layered, the `recv` function of a given protocol layer calls the `recv` function of the next lower layer, and then processes the received message according to its protocol.

The parameter `buf` points to a memory region that will hold the received bytes. The maximum number of bytes to receive is passed in `*length`. On return, `*length` is updated with the actual number of bytes received.

The `recv` function is a blocking function with a timeout. It tries to receive data for the specific amount of time. After it receives the data, or after the specified time elapses (whichever comes first), it unconditionally returns. If no data was available during the timeout period, the function returns the code TEE_ISOCKET_ERROR_TIMEOUT. If some data was received during the timeout period, the `*length` parameter is updated and the return code is TEE_SUCCESS. The timeout parameter SHALL be passed on to the lower layer without increasing its value.

If `*length == 0`, the timeout parameter has no effect and `buf` may equal NULL, and the function immediately returns the number of bytes available for transfer in `*length`.

For datagram-based protocols, the `recv` function SHALL map one call to `recv` to one received datagram. This means that calling the `recv` function with `*length` less than the size of the datagram, will not return the total number of bytes received, but only the length of the next datagram to be delivered. If the `recv` function is called with `*length` less than the size of the datagram and returns `*length == 0`, that means that there are no more datagrams to be delivered.

For datagram-based protocols, the `recv` function SHALL map one call to `recv` to one received datagram. This means that calling the `recv` function with `*length == 0` will not return the total amount of bytes received, but only the length of the next datagram to be delivered. When calling the `recv` function with `*length == 0`, and it returns `*length == 0`, it means that there are no more datagrams to be delivered.

Datagram handling in this version of TEE Sockets API is different from the handling of conventional datagram sockets, as it does not handle truncated packets. The TA implementation has to allocate the full buffer for the entire packet to map one `recv` call to one received datagram. In order to detect the desired buffer size for receiving a datagram, call `recv` with `length == 0` and use the [out] length value for buffer memory allocation. If the defined buffer is too short for the incoming datagram, the superfluous bytes in the incoming datagram will be lost and `recv` will return TEE_SUCCESS. By comparing the [out] length value containing the number of received bytes with the size of the current buffer, an implementer can detect whether and how many bytes have been lost.
For protocols that operate on a data stream, the recv function SHOULD return as soon as at least part of the buffer has been received. For protocols that operate on blocks of data, such as some encrypted protocols, the recv function SHALL internally buffer data from lower layers until a complete block is received to operate on. In such cases, the recv function SHALL act as though the complete block of data to operate on is received in one package from the lower layers.

If there is a mismatch between the received data and the protocol specification, the function SHALL discard the data, set *length = 0, and return the error code TEE_ISOCKET_ERROR_PROTOCOL.

The recv function is a cancellable function. If a cancel request is received during execution of the recv function, the implementation SHALL behave as if the timeout has been reached but the return code SHALL be TEE_ERROR_CANCEL. If the complete buffer was received, the return value SHALL be TEE_SUCCESS.

### Specification Number: See section 4.4    Function Number: 0x104

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle ctx</td>
<td>Initialized implementation specific handle.</td>
</tr>
<tr>
<td>void *buf</td>
<td>A pointer to an allocated memory buffer where the received bytes will be stored. If *length == 0, this buffer will not be touched and can be NULL.</td>
</tr>
<tr>
<td>uint32_t *length</td>
<td>A pointer to an integer holding the requested number of bytes to receive. The buffer must have allocated at least this number of bytes. On return, this parameter holds the actual number of bytes read. If the function is called with *length == 0, it returns the number of bytes that are ready to be received in *length.</td>
</tr>
<tr>
<td>uint32_t timeout</td>
<td>The timeout of the operation in milliseconds. If timeout == TEE_TIMEOUT_INFINITE (defined in TEE Core TEE_Wait), the function blocks until all requested bytes are received or a fatal error occurs. If timeout == 0, the function receives the data that is available but at most *length bytes.</td>
</tr>
</tbody>
</table>
Return Values

See section 5.2.2 for the meaning of *fatal*.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>In case of success.</td>
</tr>
<tr>
<td>TEE_ERROR_CANCEL</td>
<td>No</td>
<td>The function was cancelled.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_TIMEOUT</td>
<td>No</td>
<td>Nothing was received during the timeout period. *length is set to zero.</td>
</tr>
<tr>
<td>TEE_ERROR_COMMUNICATION</td>
<td>Yes</td>
<td>The route to the host is down or an internal network interface is down.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_REMOTE_CLOSED</td>
<td>Yes</td>
<td>The remote host has closed the connection. Some instances will never return this error due to statelessness.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_PROTOCOL</td>
<td>Yes</td>
<td>Protocol specific error. See the error function for further information. Each protocol specific error is defined in the corresponding Annex.</td>
</tr>
<tr>
<td>TEE_ISOCKET_WARNING_PROTOCOL</td>
<td>No</td>
<td>Protocol specific warning. See the error function for further information. Occurs under conditions defined in the relevant Annex.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_OUT_OF_RESOURCES</td>
<td>Yes</td>
<td>Failed to allocate resources for the operation.</td>
</tr>
</tbody>
</table>

Panic Reasons

The `recv` function SHALL panic if any of the following occurs:

- The handle is not initialized or is `NULL`.
- The handle is not a valid handle of the specific protocol.
- The parameter `length` == `NULL`.
- The parameter `*length` != 0 and `buf` == `NULL`.
- The Implementation detects any error which cannot be represented by any defined or implementation defined error code.
5.2.8  Error

```
uint32_t (*error)(
    TEE_iSocketHandle ctx
);
```

Description

The `error` function reports any protocol specific error. Some functions in the TEE_iSocket interface must return protocol specific errors or warnings and in such a case, the function returns the generic error TEE_ISOCKET_ERROR_PROTOCOL or TEE_ISOCKET_WARNING_PROTOCOL. The `error` function can be used to retrieve a more detailed error code. The definitions of the error codes are protocol specific. In a layered TEE_iSocket configuration, the TA developer needs to call the `error` function for each layer to find out where the protocol specific error occurred.

A protocol error is valid and may be retrieved until another operation on the socket has been performed, in which case the `error` function returns the status of the last operation. If no protocol specific error occurred during the last socket operation in that specific layer, the `error` function SHALL return TEE_SUCCESS.

Specification Number:  See section 4.4  Function Number:  0x105

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle ctx</td>
<td>Initialized implementation specific handle.</td>
</tr>
</tbody>
</table>

Return Values

See section 5.2.2 for the meaning of `fatal`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>The last operation on this <code>ctx</code> did not encounter any protocol errors.</td>
</tr>
<tr>
<td>Protocol specific error or</td>
<td>Depends</td>
<td>Each protocol specific error or warning is defined in the corresponding</td>
</tr>
<tr>
<td>warning value</td>
<td></td>
<td>Annex. If a function returns a TEE_ISOCKET_ERROR_PROTOCOL, it is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>considered fatal and the socket is closed. If a function returns a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEE_ISOCKET_WARNING_PROTOCOL, a warning occurred that is not fatal and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the command was successful but with a warning.</td>
</tr>
</tbody>
</table>

Panic Reasons

The `error` function SHALL panic if either of the following occurs:

- The handle is not initialized or is NULL.
- The handle is not a valid handle of the specific protocol.
- The Implementation detects any error which cannot be represented by any defined or implementation defined error code.
5.2.9 ioctl

```c
TEE_Result (* ioctl) (
    TEE_iSocketHandle ctx,
    uint32_t commandCode,
    [inout] void *buf,
    [inout] uint32_t *length
);
```

Description

The `ioctl` function is a way of extending the interface and providing protocol specific functionality to the interface. The parameter `commandCode` identifies what functionality is requested and depending on that, the `buf` parameter may be an input parameter, an output parameter, or both.

The most significant byte of the parameter `commandCode` indicates which protocol the `ioctl` function targets. If the most significant byte is zero, the `ioctl` function call is propagated throughout the stack of layered protocols.

The `commandCode` is constructed from a `protocolID` and a `commandID` as described in Table 5-1 and exemplified in Table 5-2.

Table 5-1: Structure of the `commandCode` Parameter

<table>
<thead>
<tr>
<th>Bits [31:24]</th>
<th>Function</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocolID</td>
<td></td>
<td>0x000000 – 0xEFFFFF Reserved for use in GlobalPlatform specifications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF00000 – 0xFFFFFF Reserved for Implementation specific commands.</td>
</tr>
</tbody>
</table>

Table 5-2: Examples of `commandCode` Interpretations

<table>
<thead>
<tr>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00123456</td>
<td>A general command that propagates through the protocol stack. It has a <code>commandID</code> of 0x123456, which indicates that GlobalPlatform defined it.</td>
</tr>
<tr>
<td>0x05F00001</td>
<td>A targeted command for the protocol having <code>protocolID</code> equal to 0x05. It has a <code>commandID</code> of 0xF00001, which indicates that it is Implementation specific.</td>
</tr>
</tbody>
</table>
The functionality of the `ioctl` function is as follows:

1. The byte `protocolID == 0`.
   a. If the protocol recognizes the `commandID` (has an internal function for the command), `ioctl` executes the corresponding internal function for the command.
      i. If the execution returns no error, `ioctl` invokes the `ioctl` function of the next lower protocol in the stack, using the same set of parameters (exchanging the `ctx`). The `ioctl` function returns with the return code of the lower protocol. If there is no lower layer protocol, it returns `TEE_SUCCESS`.
      ii. If the execution of the internal function returns an error, `ioctl` returns with that error.
   b. If the protocol does not recognize the `commandID`, `ioctl` invokes the `ioctl` function of the next lower protocol in the stack, using the same set of parameters (exchanging the `ctx`). The `ioctl` function returns with the return code of the lower protocol. If there is no lower layer protocol, it returns `TEE_SUCCESS`.

2. The byte `protocolID != 0`.
   a. If the parameter `protocolID` matches the protocol identifier of this protocol:
      i. If the protocol recognizes `commandID` (has an internal function for the command), `ioctl` executes the corresponding internal function for the command and returns `TEE_SUCCESS`, `TEE_ISOCKET_ERROR_PROTOCOL`, or `TEE_ISOCKET_WARNING_PROTOCOL` depending on the internal function.
      ii. If the protocol does not recognize `commandID`, it SHALL panic.
   b. If the parameter `protocolID` does not match the protocol identifier of this protocol, `ioctl` invokes the `ioctl` function of the next lower protocol in the stack, using the same set of parameters (exchanging the `ctx`). The `ioctl` function returns with the return code of the lower protocol. If there is no lower layer protocol, it SHALL panic.

Depending on the `commandCode`, the parameter `*buf` may be an input buffer, an output buffer, or both.

- If `*buf` is an input buffer, the parameter `*length` holds the length in bytes of the data in the buffer.
- If `*buf` is an output buffer, the parameter `*length` holds the number of bytes available in `*buf` for storing the output data, and upon return the `*length` parameter is updated with the actual number of bytes written into `*buf`.
- If `*buf` is both an input and an output buffer, the parameter `*length` acts as if it were a pure output buffer, and it is the responsibility of the internal function to interpret the correct number of bytes in the buffer to use as input data.

**Specification Number:** See section 4.4  
**Function Number:** 0x106
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_iSocketHandle ctx</td>
<td>Initialized implementation specific handle.</td>
</tr>
<tr>
<td>uint32_t commandCode</td>
<td>The requested command to execute. See section 5.3 and the accompanying text for a complete description.</td>
</tr>
<tr>
<td>void *buf</td>
<td>Pointer to a buffer holding input or output data of length *length. May be NULL if *length == 0.</td>
</tr>
<tr>
<td>uint32_t *length</td>
<td>Length of input data or allocated size of buffer for output data in bytes. For output data it is updated on return with the number of bytes written into *buf.</td>
</tr>
</tbody>
</table>

Return Values

See section 5.2.2 for the meaning of fatal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fatal</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEE_SUCCESS</td>
<td>No</td>
<td>In case of success.</td>
</tr>
<tr>
<td>TEE_ISOCKET_ERROR_PROTOCOL</td>
<td>Yes</td>
<td>In case of an error. See the error function for further details. Each protocol specific error is defined in the corresponding Annex.</td>
</tr>
<tr>
<td>TEE_ISOCKET_WARNING_PROTOCOL</td>
<td>No</td>
<td>In case of a warning. See the error function for further details. Each protocol specific warning is defined in the corresponding Annex.</td>
</tr>
</tbody>
</table>

Panic Reasons

The ioctl function SHALL panic if any of the following occurs:

- The handle is not initialized or is NULL.
- The handle is not a valid handle of the specific protocol.
- The parameter length == NULL.
- The parameter buf == NULL and *length > 0.
- The byte protocolID != 0 and the corresponding protocol or commandID is not found within the current stack of protocols.

The Implementation detects any error which cannot be represented by any defined or Implementation defined error code.

5.3 Global commandCode Definitions for ioctl

In this version of the specification, no global ioctl commands are defined.
5.4 Example of a TEE_iSocket Protocol Implementation

This section provides a short abbreviated example of an implementation of the Foo protocol. It is only meant as a guideline for how to declare the protocol specific structures and functions required by the TEE_iSocket interface.

5.4.1 The Header File

```c
/* Header file tee_foosocket.h */
#ifndef TEE_ISOCKET_PROTOCOLID_FOO
#include "tee_isocket.h" /* provides definition for TEE_iSocket */

#ifndef ISOCKET_PROTOCOLID_FOO
#define ISOCKET_PROTOCOLID_FOO 0x35

typedef struct TEE_fooSocket_Setup_s {
    /*
     * All things needed to setup the FOO protocol.
     */
    } TEE_fooSocket_Setup;

extern TEE_iSocket * const TEE_fooSocket;
#endif
```

5.4.2 The C Implementation File

```c
/* Implementation file tee_foosocket.c */
#include "tee_internal_api.h"
#include "tee_foosocket.h"

typedef struct fooSocket_Context_s {
    /*
    * All things needed to maintain the context
    */
    uint32_t          *protocolError;
} fooSocket_Context;

/*
 * FOO_Open
 */
static TEE_Result FOO_open(TEE_iSocketHandle *ctx,
    void              *setup,
    uint32_t          *protocolError)
{
    fooSocket_Context *context;
    TEE_fooSocket_Setup *mysetup;

    /*
    * Check for panic criteria.
    * NOTE that in real code the correct function ID and
    * document ID need to be returned by the panics.
    */
    if ((ctx == NULL) || (setup == NULL) || (protocolError == NULL) ) {
        TEE_Panic(TEE_ERROR_BAD_PARAMETERS);}
    }

    mysetup = (TEE_fooSocket_Setup *)setup;
    /*
    * Here we should check the correctness of the mysetup struct
    * and MAY Panic if it's not correct
    */

    /*
    * Allocate memory for context, and get a POINTER
    * in this case the pointer also consitutues the handle
    * used by iSocketHandle.
    */
    context = (fooSocket_Context *) TEE_Malloc(sizeof(fooSocket_Context),
        TEE_MALLOC_FILL_ZERO);
    if (context == NULL) {
        *ctx = TEE_HANDLE_NULL;
        return TEE_ERROR_OUT_OF_MEMORY;
    }
```
/*
 * Populate the fields in context to maintain the
 * communication channel.
 */
context = (TEE_fooSocket_Context *) ctx;

/* Check what is currently in the ctx by assigning it to context. */
*ctx = (TEE_iSocketHandle) context;

/* one example is to store the address used to report protocol errors */
context->protocolError = *protocolError;

/*
 * Open the communication channel according to the
 * parameters in mysetup.

 * [ Done here ]

 * There may be more context to be stored in relation
 * to the open channel before we:
 */
return TEE_SUCCESS;
}

/*
 * FOO_Close
 */
static TEE_Result FOO_close(TEE_iSocketHandle *ctx)
{
    if (ctx == TEE_HANDLE_NULL)
        return TEE_SUCCESS;

    /*
     * Check for panic criteria using helper functions.
     * NOTE that in real code the correct function ID and
     * document ID need to be returned by the panics.
     */
    if (FOO_CtxInvalidInProtocol(ctx) || FOO_CtxNotInitialized(ctx)) {
        TEE_Panic(TEE_ERROR_BAD_PARAMETERS);
        return TEE_SUCCESS;
    }

    /*
     * Code to clean up context goes here
     */

    TEE_Free(ctx);
    return TEE_SUCCESS;
}
/* Truncated definitions for the remaining functions */

```c
static TEE_Result FOO_send(
    TEE_iSocketHandle  *ctx,
    const void       *buf,
    uint32_t     *length,
    uint32_t      timeout ) { /* body */ }

static TEE_Result FOO_recv(
    TEE_iSocketHandle  *ctx,
    void           *buf,
    uint32_t     *length,
    uint32_t      timeout ) { /* body */ }

static uint32_t  FOO_error(
    TEE_iSocketHandle  *ctx )  { /* body */ }

static TEE_Result FOO_ioctl(
    TEE_iSocketHandle *ctx,
    uint32_t  commandCode,
    void   *buf,
    uint32_t     *length )  { /* body */ }

/* Instance declaration of the FOO protocol functions */

TEE_iSocket fooSocketInstance = {
    TEE_iSocketVersion,
    ISOCKET_PROTOCOLID_FOO,
    &FOO_open,
    &FOO_close,
    &FOO_send,
    &FOO_recv,
    &FOO_error,
    &FOO_ioctl
};

TEE_iSocket * const TEE_fooSocket = &fooSocketInstance;
```