TrustZone API Specification

Version 3.0

Document number: PRD29-USGC-000089 3.1

Date of Issue: 20 February 2009

Abstract

This document provides the specification of the TrustZone® API, a software application programming interface that enables application code running in a general purpose operating system to efficiently and robustly communicate with a dedicated security environment.

This API originated as a mechanism to enable efficient communications between a rich operating system and a security middleware environment running on an ARM processor implementing the ARM Architecture Security Extensions. Recognizing that development of security software has been hindered by the lack of common standards for software development, ARM is releasing this API as a public specification that can be implemented by any security platform vendor as an interface to their underlying security solution.

Contents

1 PREFACE

1.1 Document confidentiality

This document is confidential and may only be distributed to parties agreeing to the terms of the TrustZone Software API license, as stated below:

1.1.1 TrustZone Software API license

Subject to the provisions of Clauses 2 and 3, ARM hereby grants the LICENSEE a perpetual, non-exclusive, nontransferable, royalty free, worldwide license to use and copy the TrustZone Software API for the purpose of developing, having developed, manufacturing, having manufactured, offering to sell, selling, supplying or otherwise distributing products which comply with the TrustZone Software API.

THE TRUSTZONE SOFTWARE API IS PROVIDED “AS IS” WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.

No license, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM trade name in connection with the TrustZone Software API or any products based thereon. Nothing in Clause 1 shall be construed as authority for LICENSEE to make any representations on behalf of ARM in respect of the TrustZone Software API.

1.2 Disclaimer

ARM reserves the right to modify and revise this specification and technology described here. All ARM technology is subject to contract.

2 DOCUMENT STATUS

2.1 Change history

Version	Date	Status	Comments
1.0	29.06.2005	Issued	First public release.
2.0	22.09.2006	Issued	Major revision to simplify implementation. Addition of asynchronous API option to support event driven operating systems.
2.0.1	28.03.2008	Issued	No functional changes; minor corrections.
3.0	20.02.2009	Issued	Major revision to make implementation smaller, faster and more memory efficient.

2.2 References

Ref	Author	Description
[ISO/IEC 9899:1999]	ISO	ISO/IEC C99 Language Specification.
[RFC4122]	P. Leach, M. Mealling, R. Salz	RFC4122: A Universally Unique IDentifier (UUID) URN Namespace.
[TReqStatus]	Symbian Limited	Symbian OS v9.1 Reference Manual for TRequestStatus.
[ARM IHI 0036A]	ARM Limited	Application Binary Interface for the ARM Architecture.

3 OVERVIEW

Many embedded platforms now include a secure environment which may, for example, be implemented using a smartcard, a dedicated security processor within the device, or within a single processor implementing the ARM Security Extensions. Software applications running outside of the secure environment often need to communicate with services running within the security environment, although the most efficient mechanism for this communication is often varied.

The TrustZone API (TZAPI) is a programming interface that allows client applications to communicate with an abstract security service in a manner which is independent of the implementation of both the security environment and the security service itself. An implementation of the API can ensure that the communications interfaces are implemented on top of the most suitable underlying physical mechanism available on the hardware platform.

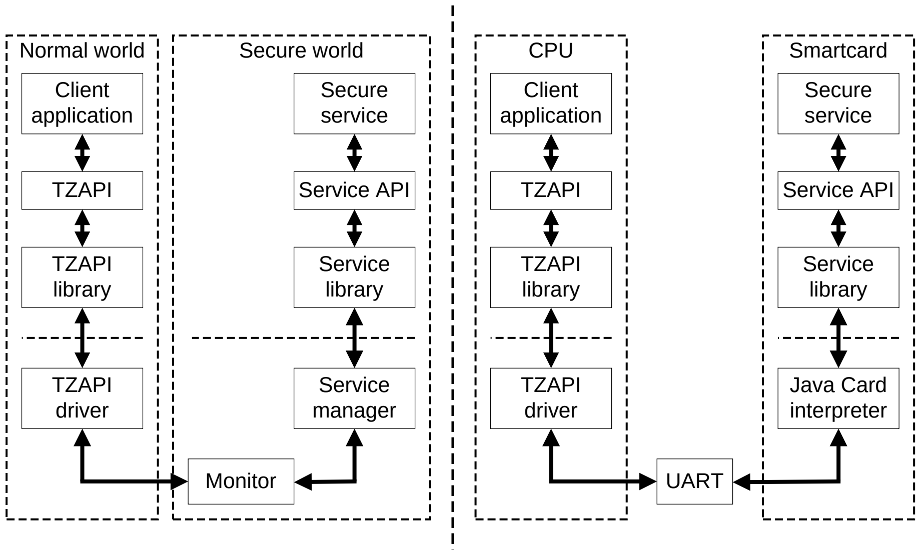

Figure 1: Two possible systems using the TrustZone API

Figure 1 shows two possible platforms using implementations of the TrustZone API: the one on the left is implemented on an ARM processor using the ARM Security Extensions, the one on the right is an implementation communicating with a smartcard-based security environment.

3.1 The TrustZone API

The TZAPI is a programming interface that enables a client application to access a security environment for managing and using secure services. It enables a client which has appropriate privileges to connect to a service and issue commands to that service. A command is an abstract message which instructs the service to perform some useful work on behalf of the client. A client can also query installed services and, if the implementation allows it, install new services at run-time.

A client is typically composed of an application and a service stub. The service stub is an abstraction layer which implements a service-specific AP, and translates these high level calls in to appropriate TZAPI calls. Both of these client components can use the TZAPI for managing services or accessing service functionality; however it is more common for a service stub to completely abstract the TZAPI from an application, as shown in Figure 2.

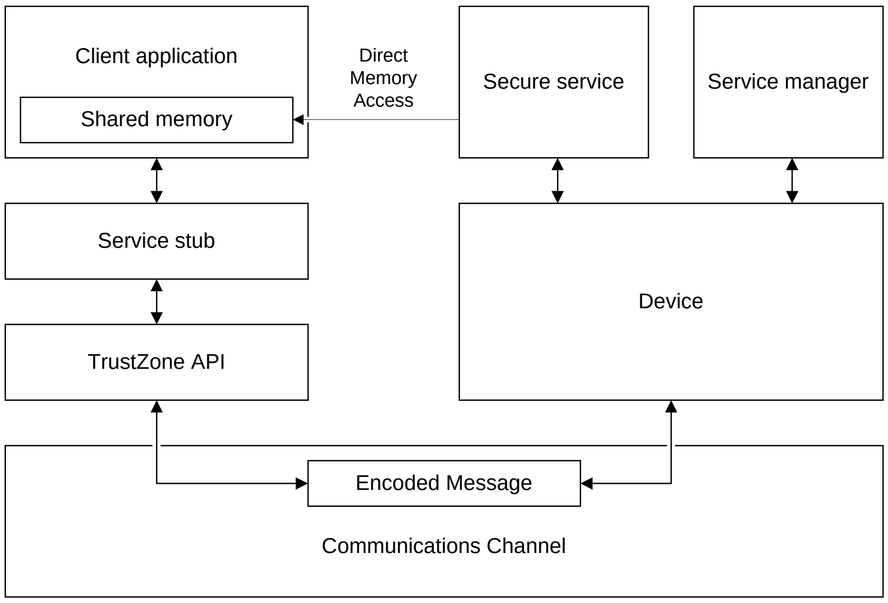

Figure 2: TrustZone API architecture overview

From the client's point of view, the TZAPI decomposes the security environment into four functional blocks:

- TrustZone API implementation:
  - The nature of the TZAPI implementation is platform-specific, but will typically include a component that exists as library code within the client memory space and a portion within the privileged memory of the host operating system. The component within the operating system, commonly called a driver, is more trusted than the component in the client memory space and may be capable of making some limited security decisions, for example the derivation of client identity to provide login credentials.
- Device:
  - A single logical device forms the basis of each security environment available on the system. The device provides the entry point to the security environment and manages all connections between the client and other components of the environment.
- Service manager:
  - A logical management component that can be used to query installed services, and change the services that are installed within the device.
- Services:
  - Any number of services running within the security environment, exposing high level security functionality to client applications.

Most of the functions in the TZAPI are designed to enable clients and services to communicate in a robust and efficient manner. Clients and services can communicate through two mechanisms: structured messages, and shared memory. Shared memory is client memory that is mapped directly into the memory space of service. Structured messages can be used to robustly pass small quantities of data, while shared memory is particularly useful for minimizing the overhead when a service has to deal with large data buffers.

Note that the availability of genuine shared memory depends on the nature of the security environment; some hardware platforms, such a smartcard-based secure environment, cannot support direct mapping of client memory. In these systems the implementation of shared memory may require a memory copy.

3.2 Client application

Client applications run within the normal operating system and are developed in the same manner as any other applications. Clients can either be stand-alone, such as a media player or an internet browser, or libraries that enable third-party applications to access critical security functionality, such as a cryptography library.

Client applications use the TZAPI to manage and access services, usually via an intermediate API provided by a service stub library, in order to access security functionality through a standardized interface.

In most operating systems a client application represents an operating system process. Most operating systems support multi-threaded processes, so the implementation of the TZAPI may need to be thread-to allow concurrent accesses from multiple threads within a single client application. Implementation of multi-threaded calling support is implementation-defined.

Some operating systems support an asynchronous programming interface; the TZAPI therefore includes asynchronous variants of some functions in order to support these platforms. The asynchronous functions are optional and may not be available in all implementations of the TZAPI.

3.3 Service stub

A service stub is an abstraction layer that is provided by the service developer. It sits above the TZAPI and implements a higher-level service-specific API which hides the low-level TZAPI message processing from the application.

A service stub is responsible for communicating with the service through client sessions, and encodes and decodes messages exchanged with the service. In addition, the service stub should handle errors returned by the TZAPI, including errors returned by the service via the TZAPI.

For some operations a service stub can control the service response time by specifying a time limit after which the operation will be aborted if it has not completed. The service stub can also explicitly cancel the operation if it is no longer required.

Note: The boundary between client applications and service stubs is only a logical convenience. There is a single level of trust on the client side, and the TZAPI does not make any distinction between client applications and service stubs, or their roles and responsibilities.

3.4 Device

Security execution environments are known as devices and, along with their resources and available operations, form the foundation of the security architecture. Several devices may co-exist on the same platform – each device being identified by a platform-unique name. The name of a device is implementation-dependent.

It is often the case that only one device is present on the platform, or that a particular device is designated as the “default” device. To improve interoperability, the default device must be accessible using a NULL identifier.

A device is persistent on a platform and is made accessible through one or more device contexts – a logical container in the TZAPI implementation that contains all of the information about a client and the sessions that it has open with services. When clients connect to a device, a device context is created and a reference to it is returned by the TZAPI implementation.

A client may be able to make multiple connections to a device, and as such may have multiple device contexts. On some systems it may also be possible to connect to multiple distinct devices.

A device context encompasses all the information required for linking a client to a specific device. Consequently the TZAPI implementation may store information within the device context that allows a binding with an equivalent context in the security environment. This creates a logical connection between the client and the secure environment.

The Service manager and custom services are only accessible through an instantiated device context, which means that a client must create a device context before being able to access the service manager or any available services.

When a device context is destroyed, the client can no longer access the device and the entire state defined by that device context is lost.

3.5 Service

A service is a dedicated program that runs within the security environment to provide security-related functionality, such as an implementation of a DRM or payment agent. A service is command-oriented, which means it receives command invocation requests from a client, processes the request and then sends back the responses.

A service is globally identified in the system by a Universally Unique Identifier (UUID, see **[RFC4122]**), known as the service identifier, which is specified by the service provider or the security environment.

Clients must open a session with a service before invoking service commands. To open a session the client must issue an ‘open’ operation, specifying the service UUID of the target service. The UUID is either hard-coded into the client or determined by interrogating the Service manager for a set of desired service properties.

Multiple clients may attempt to open sessions with the same service concurrently, but a single session cannot be shared between clients. A client can open multiple sessions with a single service if so desired. However, the underlying service may not support multiple sessions, and may reject ‘open session’ requests after the first. This feature is service-dependent and should be specified in the service documentation.

Once a client has opened a session it can send commands to the service. The available command identifiers, parameters and return values are service-dependent and should be specified in the service documentation.

A client may be able to invoke multiple commands simultaneously within the same session, either from multiple threads or by invoking operations asynchronously. A specific service may support multiple simultaneous commands, and may reject command invocations after the first. This feature is service-dependent and should be specified in the service documentation.

A service possesses a number of properties as name:value pairs. These properties include the service identifier, the human-readable name of the service, the human-readable name of the service vendor, and any implementation-defined properties provided by the service.

3.6 Service manager

The Service manager allows a client to inspect the installed services and to manage them. The Service manager exposes functions that enumerate all the services installed in the device and that return their properties. The client can use these functions to dynamically select a service at run-time, based on detecting a known UUID or by selecting a service based on features identified by entries in its property list.

The Service manager also provides functions to download new services into the execution environment and to remove existing services. The Service manager may restrict access to these functions, based on client login credentials provided when the connection to the Service manager is opened by the client. The required level of client login, and the format of the installable service binaries, is implementation-defined.

3.7 Login

When a client opens a new session with a service, or with the Service manager, it can specify some login parameters. Some operations may require that the client is identified or authenticated with a particular set of login information; attempting to access these operations with insufficient login credentials will result in an error.

A client can login using the TZ_LOGIN_PUBLIC login type, which implies that no login is attempted, or can prompt the underlying TZAPI implementation to generate one or more login tokens. These additional tokens can provide information about the identity of the user, the calling client, and even a cryptographic digest of the client application binary itself. A secure service can decide whether or not to allow a client connection, based on these tokens.

3.8 Structured messages

The data exchanged between clients and services as part of an operation request is marshaled and encoded in a data structure known as a structured message. A client can exchange structured messages with a service when it opens or closes a session or when it invokes a command within a session. These messages contain operation parameters sent to a service by a client or the response data returned to the client by the service (once the operation completes).

Data items are encoded into, and decoded from, structured messages using encoders and decoders provided by the TZAPI. New encoder and decoder instances are created by the TZAPI implementation as needed. The available space for message encoding is implementation-defined.

3.9 Shared memory

Structured messages are not the only way to communicate between clients and services. Buffers of memory in the client memory space can also be shared with a service. Shared memory blocks can be created in two ways:

- Registration of an existing client memory region with the service.
- Allocation of new client memory region by the TZAPI implementation.

On many platforms shared memory allows fast, zero-copy, unstructured communication between a client and a service. This feature is particularly useful for minimizing the overhead incurred when a service has to deal with large data buffers, such as multimedia files or encrypted disk partitions.

4 USER GUIDE

This section explains the typical usage of the TZAPI in order to manage and use services, providing a step-by-step approach to using the TZAPI for common tasks. It must be noted that clients need not adhere to the exact procedures detailed here, providing that the TZAPI is used correctly according to the specification provided later in this document.

This section, along with sections 5 and 6, details the dependencies between the different components of the TZAPI and the contexts in which functions can be called. Misuse of the TZAPI by a client application must not affect the execution of the security environment. The implementation of the security environment, and the services it contains, must be aware that clients are considered untrusted (as they exist outside of the security environment). Services must take appropriate measures to defend against misuse of the API by clients.

4.1 Device usage

In order to access the security environment a client should first open a connection with the underlying device. To achieve this, the client must call the TZDeviceOpen function, specifying the name of the device that it wants to open a connection with. The name of the supported devices is implementation-defined, but every implementation of the TZAPI library must support a default device, which the client can access using a NULL device name. A device may accept an optional configuration block which specifies parameters which configure the device for use. The configuration block is specified when the client opens the device, and the format of the configuration block is implementation-defined. Every device implementation must support a NULL configuration, which selects a sensible default configuration.

The function TZDeviceOpen returns a populated structure of type tz_device_t, which can be passed into other API functions when a reference to the device is needed. Subsequent operations involving the Service manager or third-party services can only be performed once the device connection has been opened.

When the client has completed its task using the secure environment it must call TZDeviceClose to close the connection. When TZDeviceClose completes the connection is irrevocably lost. If any service sessions are still open when the client attempts to close the connection with the device, the close function will return an error code. The client must therefore ensure that all service and Service manager sessions relating to a given device connection are properly closed before attempting to close the device connection.

```
/* Declare variables */
tz_return_t uiReturn;
tz_device_t sDevice;

/* Create a device connection: */
uiReturn = TZDeviceOpen(
               NULL,              /* Use default device. */
               NULL,              /* Use default configuration. */
               &sDevice);         /* New device connection structure. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Failed to open device! */
    goto exit;
}

/* Do something useful here. */
...

/* Close the device connection: */
exit:

uiReturn = SMDeviceClose(
               &sDevice);        /* Device to close. */
```

```
/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Device is not closed! */
}
```

4.2 Service usage

In order to use a service, a client must first open a session with it. The client may then invoke one or more service commands and receive the responses thereof. Finally, when all commands have been issued, the client must close the session.

4.2.1 Identifying services

Services are identified using UUIDs (Universally Unique IDentifiers). A UUID is a persistent identifier and can be assigned to the service at any point during development or installation.

The UUID 79B77788-9789-4A7A-A2BE-B60155EEF5F3 can be hard-coded into a client using the following C code:

```
static const tz_uuid_t ksMyUUID =
{
    0x79B77788, 0x9789, 0x4A7A,
    {
        0xA2, 0xBE, 0xB6, 0x01, 0x55, 0xEE, 0xF5, 0xF3
    }
};
```

Alternatively, the client can dynamically inspect the properties of the services installed on the device and select a service based on the returned values. For example, a service might advertise its conformance to a specific service interface, or describe the functionalities it implements, using its properties. In order to perform this inspection the client must open a session with the Service manager and use the functions TZManagerGetServiceList and TZManagerGetServiceProperty or TZManagerGetServicePropertyList.

4.2.2 Opening a client session

Once the client has the UUID of the service it needs to connect to, it can open a client session with the service by performing the following steps:

1. Prepare the open operation by calling the function TZOperationPrepareOpen. This function requires an open device connection, a valid UUID for the target service and any additional login and timeout information. Calling this function allows the implementation to create an operation context, returning any state in the `tz_operation_t` structure passed in by the client. If successful, this function returns an operation in the state TZ_STATE_ENCODE, which allows the encoder API to be used on the operation.
2. Add arguments to the open operation's structured message using the API encoder functions. It is acceptable to send an empty message to the service; in this case it is not necessary to call any encoder functions.
3. Call the function TZOperationPerform, or the asynchronous equivalent, to connect to the service and send the opening message. The service will process the arguments in the message and can optionally return a new message containing output from the open operation. Once either of the operation perform function has been called it is not valid to use the encoder API functions. It is not valid to use the decoder API functions until this operation perform operation completes.
4. Decode the service's answer, if any, using the API decoder functions.
5. Finally, release the operation context using the function TZOperationRelease. This step frees the decoder used by this operation, allowing it to be used for new commands. The client must have read all data it requires from the decoder memory space, including array content, before calling this function.

If the session creation is successful, the client structure returned by TZOperationPrepareOpen can subsequently be used to create new commands to exchange with the service.

A service may support multiple concurrent sessions; this feature is service-dependent. The TZOperationPerform function will return the error TZ_ERROR_BUSY or TZ_ERROR_ACCESS_DENIED if a client tries to open a session with a service that is capable only of single session operation and which already has a session open.

Some services may require the implementation to identify or authenticate the client – for example a banking service may restrict client connections to those clients able to provide acceptable credentials. When opening a session, the client and/or the implementation can provide 'login' information for the purpose of authentication. From the client's point of view the information is composed of a structure of type `tz_login_t`. This structure contains flags that inform the TZAPI implementation of which login credentials to provide and an optional pointer to a buffer of data provided by the client. Login can be used to provide information about the user running the application, the file system path of the application, or a cryptographic hash of the application itself. The details of each of these login credentials is implementation-defined.

The following code example shows the sequence of operations required to prepare, perform and release an open session operation within a service.

```
/* The UUID of the service to connect to. */
static const tz_uuid_t ksUUID =
{
    0x79b77788, 0x9789, 0x4a7a,
    { 0xa2, 0xbe, 0xb6, 0x1, 0x55, 0xee, 0xf5, 0xf3 }
};

tz_device_t    sDevice           /* Device connection. */

tz_return_t    uiReturn;         /* Return value from system. */
tz_return_t    uiSvcReturn;      /* Return value from service. */
tz_session_t   sSession;         /* Uninitialized session. */
tz_operation_t sOperation;       /* Uninitialized operation. */

uint32_t       uiAnswer;         /* Variable to hold service answer. */

/* Open device connection. */
...

/* Prepare the Open operation. */
uiReturn = TZOperationPrepareOpen(
               &sDevice,         /* An open device. */
               &ksUUID,          /* The service UUID. */
               NULL,             /* TZ_LOGIN_PUBLIC login. */
               NULL,             /* No time limit. */
               &sSession,        /* Session structure. */
               &sOperation);     /* Operation structure. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Could not create an open operation for some reason. */
    goto exit;
}

/* Encode structured message. */
/* This step may be omitted if the service requires no message. */
TZEncodeUint32(&sOperation, 0x01234567);
```

```
/* Send the open command to the service. */
uiReturn = TZOperationPerform(
               &psOperation,     /* Operation structure. */
               &uiSvcReturn);     /* Return if service error occurs. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    if(uiReturn == TZ_ERROR_SERVICE)
    {
        /* Service threw an error – stored in uiSvcReturn. */
        /* Can use decoder functions if it sent a message. */
    }

    else
    {
        /* System threw an error – stored in uiReturn. */
        /* Cannot use decoder functions. */
    }
    goto exit;
}

/* If here then service returned success - session is open. */
/* Decoder an answer message, if we expect any. For example: */
uiAnswer = TZDecodeUint32(&sOperation);

/* Check decoder has not errored. */
uiReturn = TZDecodeGetError(&sOperation);
if(uiReturn !=  TZ_SUCCESS)
{
    /* A decode error occurred - for example, a type mismatch. */
}

/* Release the operation context: */
exit:

TZOperationRelease(&sOperation);
```

4.2.3 Invoking service commands

Once a client session is opened, a client can invoke service commands. A command is composed of a command identifier, which is an unsigned 32-bit integer, and an optional structured message. The service's response to a command consists of an unsigned 32-bit return code and an optional answer message.

The protocol that defines the available command identifiers, and the required contents of the messages for each command, are specific to each service – these must be documented by the service provider.

To invoke a service command, the client must perform the following steps:

1. Prepare the invoke operation using the TZOperationPrepareInvoke function. The client must pass a pointer to an existing session opened with the service and the required command identifier. Calling this function allows the implementation to allocate the operation context locally; the implementation will not generally send anything to the device or service at this stage. When this function returns, the client code can call the encoder functions to encode a structured message to send to the service.
2. Fill in the command message using the encoder functions. It is acceptable to send an empty message to the service if no arguments are required; in this case, it is not necessary to call any encoder functions.
3. Call the function TZOperationPerform, or the asynchronous equivalent, to send the command to the service. The service will process the arguments in the message and can optionally return a new message containing

output from the invoke operation. Once operation perform stage has been started it is not valid to use the encoder API functions. It is not valid to use the decoder API functions until the perform stage has completed.

4. Decode the service's answer, if any, using the decoder functions and the decoder handle.
5. Finally, release the operation context using the function TZOperationRelease. This step frees the decoder used by this operation, allowing it to be used for new commands. The client must have read all data it requires from the decoder memory before calling this function. The client must call this function even if the perform function fails, or if the service's response contains no answer message.

The following code example shows the sequence of operations required to prepare, perform and release a perform command operation within a service.

```
tz_device_t    sDevice;          /* Device connection. */
tz_session_t   sSession;         /* Service session. */

tz_return_t    uiReturn;         /* Return value from system. */
tz_return_t    uiSvcReturn;      /* Return value from service. */
tz_operation_t sOperation;       /* Uninitialized operation. */

uint32_t       uiAnswer;         /* Variable to hold service answer. */

/* Open device connection. */
...

/* Open a service session. */
...

/* Prepare the Invoke operation. */
uiReturn = TZOperationPrepareInvoke(
               &sSession,        /* An open session. */
               42,               /* The ID of the command to invoke. */
               NULL,             /* No time limit. */
               &sOperation);     /* Operation structure. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Could not create an Invoke operation for some reason. */
    goto exit;
}

/* Encode structured message. */
/* This step may be omitted if the service requires no message. */
TZEncodeUint32(&sOperation, 0x76543210);

/* Send the Invoke command to the service. */
uiReturn = TZOperationPerform(
               &psOperation,     /* Operation structure. */
               &uiSvcReturn);    /* Return if service error occurs. */
```

```
/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    if(uiReturn == TZ_ERROR_SERVICE)
    {
        /* Service threw an error – stored in uiSvcReturn. */
        /* Can use decoder functions if it sent a message. */
    }
    else
    {
        /* System threw an error – stored in uiReturn. */
        /* Cannot use decoder functions. */
    }
    goto exit;
}

/* If here then service returned success - session is open. */
/* Decode an answer message, if we expect any. For example: */
uiAnswer = TZDecodeUint32(&sOperation);

/* Check decoder hasn not errored. */
uiReturn = TZDecodeGetError(&sOperation);
if(uiReturn !=  TZ_SUCCESS)
{
    /* A decode error occurred; for example a type mismatch. */
}

/* Release the operation context. */
exit:

TZOperationRelease(&sOperation);
```

4.2.4 Closing a client session

When a client no longer needs to access a service it can close the session. To close a session, the client must perform the following steps:

1. Prepare the close operation using the TZOperationPrepareClose function. The client must pass an open session to this function. Calling this function allows the implementation to allocate the operation context locally; the implementation will not generally send anything to the device or service at this stage.
2. It is not valid to use the encoder functions when constructing a close operation as the operation cannot carry a payload.
3. Call the function TZOperationPerform, or the asynchronous equivalent, to connect to the service and send the close message.
4. It is not valid to use the decoder functions when receiving the response to a close operation; it may carry no payload.
5. Finally, release the operation context using the function TZOperationRelease. This final stage also releases the client session; the session is no longer valid after this point.

4.3 Encoding and decoding structured messages

A client uses the TZAPI to encode and decode structured messages exchanged with a service. Structured messages are a convenient way to develop a robust protocol between the client and a service – enabling an implementation to provide type safety and defensive protection against buffer overflow issues. Structured messages can contain three data types: 32-bit unsigned integers, binary arrays, and a complex type representing a shared memory reference.

Structured messages are strongly-typed; whenever an attempt is made to decode a parameter in a message, only the decode function corresponding to the actual type of the parameter can succeed. Strongly-typed structured messages are an effective and centralized protection to type mismatch or buffer overflow attacks, where a client attempts to send an ill-formed message to a service.

Messages created using the TZAPI are encoded and decoded in first-in-first-out order. That is, messages must be decoded in the same order as that in which they were encoded.

4.3.1 Encoding data

The TZAPI automatically creates an encoder when it is valid for a client to send a message to a service, i.e. whenever the client calls one of the TZOperationPrepare* functions. An encoder is created for the operation when these prepare operations return success; this is represented by a state change in the operation object. The client can then make use of the API encoder functions to construct the message sent to the service.

A specific function is defined for each data type to be encoded. The names of the encoder functions start with the prefix TZEncode*. Array encoding allows a zero-copy implementation by allowing a client to encode a space of known size in the encoder memory, a pointer to which is then returned to the client. The client can then generate the data to place into the encoded space in the array before issuing the command to the service.

4.3.2 Decoding data

When an operation has completed, the decoder functions allow the client to decode the service's response, subject to the following constraints:

- Where the implementation returns TZ_SUCCESS or TZ_ERROR_SERVICE, the TZAPI implementation must return a valid decoder even if the service's response did not include any message – a decoder containing no data must be returned if this is the case.
- Where the implementation returns an error code which is neither TZ_SUCCESS, nor TZ_ERROR_SERVICE, the decoder is not generated and the decoder functions must not be called by the client.

A specific function is defined in the TZAPI for each data type which can be decoded. As mentioned previously, in order for a message to be decoded successfully, the sequence of data types extracted from a message must match the order in which the message was originally encoded.

The names of the decoder functions start with the prefix TZDecode*. The function TZDecodeArraySpace returns a pointer to a memory buffer owned by the decoder. The client code must finish using the data in the array slot, or copy it to a safe location in client owned memory, before releasing the operation for which the decoder was created.

4.3.3 Empty and NULL arrays

Memory arrays can have zero elements (empty) and can also be NULL. The TZAPI encoders and decoders treat each of these and an independent case and these can be identified uniquely on decode.

4.3.4 Handling encoder and decoder errors

The TZAPI optimizes error management for encoders and decoders by storing an error state in the encoder or decoder instance.

The encode functions return no error code. If an error occurs, for example an invalid parameter or an out-of-memory error, the TZAPI implementation sets the encoder error state. When the client then tries to perform the operation the API will return the error immediately without issuing the command to the secure environment. There is no mechanism to reset an encoder once an error has occurred – the operation must be released and a new one created.

The decode functions also return no error code, and similarly set the decoder error state if an error condition occurs. When the client wants to check that no decoding error has yet occurred, it can call the function TZDecodeGetError to retrieve the decoder error state. This helps to optimize the client code size because the client typically needs to check the decoder error state only when the whole message has been decoded.

The following code sequence decodes a sequence of parameters, and checks the error state at the end of the decode sequence:

```
tz_operation_t sOperation;       /* Refers to a performed operation. */

uint32_t uiA;                    /* Message item 1. */
uint32_t uiB;                    /* Message item 2. */
uint8_t  * pbC                   /* Message item 3 (buffer) .*/
uint32_t uiLenC;                 /* Length of item 3 (buffer). */

/* Attempt to decode the contents of the decoder: */
uiA = TZDecodeUint32(&sOperation);
uiB = TZDecodeUint32(&sOperation);
pbC = TZDecodeArraySpace(&sOperation, &uiLenC);

/* Check the error code: */
if(TZDecodeGetError(&sOperation) != TZ_SUCCESS)
{
    /* Decoder error occured. */
}
```

There is no function available to reset the decoder error state as it is expected that any error occurring should be treated as fatal.

4.4 Shared memory blocks

Using structured messages may add a significant overhead when transferring large quantities of data between a client and a service. This may be unacceptable in some use-cases where a minimum data bandwidth is required to achieve data streaming. To overcome this bandwidth problem, the TZAPI provides the capability to designate blocks of memory that are shared between the client and the service and directly accessible to both. It is recommended that this feature be used to exchange bulk data, while still using structured messages to exchange control information.

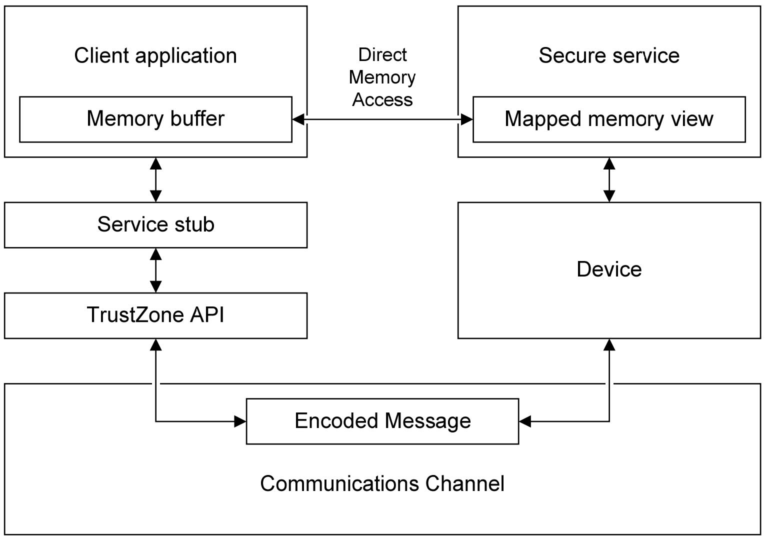

Figure 3: Shared memory mapping using a memory reference

4.4.1 Allocation of shared memory blocks

Shared memory blocks can be allocated by the TZAPI implementation; clients must call the function TZSharedMemoryAllocate to achieve this. This allows the underlying allocation operation to provide the most efficient memory for sharing while also giving opportunity for it to perform any critical tasks required by the platform to make the memory sharable, such as page locking. Once a shared memory block is allocated it is manipulated through a shared memory structure returned by the allocation function. These memory structures are associated with a single client session and can only be used in the context of that session.

When a shared memory block is allocated, the client can specify the intended service access semantics. Shared memory blocks are flagged with service read/write permissions which are used to inform the underlying implementation of the direction of data flow. This enables the implementation to perform any synchronization operations that are required when a reference to the shared memory is exchanged.

These access permissions are specified using flags defined when the memory is allocated:

- Service permissions – one of:
    - TZ_MEM_SERVICE_RO (Service is read-only)
    - TZ_MEM_SERVICE_WO (Service is write-only)
    - TZ_MEM_SERVICE_RW (Service is read and write)

To summarize, when allocating a shared memory block the client specifies:

- The client session in which the block is to be allocated.
- The size of the block.
- The access flags for the block.

4.4.2 Registration of shared memory blocks

Shared memory blocks may be created by registering existing portions of the client memory space within a session with a Secure world service, enabling direct access to the portions that have been registered. This is intended to be used when a client needs to map a buffer of data which has already been allocated by a library or another process in the system, and removes the need to copy the data into a special block of allocated memory.

When a shared memory block is registered the client must also specify the intended service access semantics, in the same manner as access permissions specified when a block is allocated.

4.4.3 Use of shared memory blocks

Once the client has allocated a shared memory block, it can pass memory references to portions of it to a service using structured messages. Structured message entries can be encoded by using the function TZEncodeMemoryReference. This function encodes a reference which can be decoded to give the service access to a contiguous interval of bytes within the block, known as a memory range.

Memory references are automatically released when the operation completes. This means that references are valid only during the lifetime of the operation and cannot be used by the service when the operation ends. If a block must be used in multiple operations then a new reference must be written for each operation. Writing a new memory reference is typically significantly less expensive than allocating a new shared memory block.

4.4.3.1 Efficiency of TZSharedMemoryAllocate and TZSharedMemoryRegister

Developers should be aware that the virtual memory architecture of some operating systems will require the TZAPI library implementation of TZSharedMemoryRegister to make a copy of any exchanged memory regions. The implementation of TZSharedMemoryAllocate will not require an internal copy as memory can be allocated with the appropriate configuration that is required by the host operating system for correct behavior.

Client developers should therefore apply the following rules when deciding whether to use TZSharedMemoryAllocate or TZSharedMemoryRegister in their implementation:

- If the client locally allocates memory and then generates the data which fills the buffer, it is most efficient to allocate the memory buffer using TZSharedMemoryAllocate. This is the most likely method to generate a zero-copy protocol with the Secure world if the system can support zero-copy at all.
- If the client code receives a pointer to an already allocated buffer from a library or another process it is most efficient to register the shared memory using TZSharedMemoryRegister. If the implementation can support zero-copy memory transfer the system will provide it, otherwise the required copy will be inserted, which is no worse than the client manually copying into a buffer allocated using the TZAPI.

For both of these operations the client developer should be aware that the access flags determine the copies that are required when passing data to, or receiving data from, the service. The client should specify the minimum set of access permissions required for each piece of memory in order to minimize the number of copies that might occur.

Note it is also generally more efficient to specify multiple memory references with restricted permissions than it is to specify a single larger region with a union of the required permissions. A simple protocol which passes a 4KB contiguous buffer to a service, where the bottom 2KB is used for input and the top 2KB is used for output, should encode the data using two memory references. The lower 2KB memory reference is specified with the flag

TZ_MEM_SERVICE_RO, and the upper 2KB memory reference is specified with the flag TZ_MEM_SERVICE_WO. This requires half of the memory copies that a single 4KB memory reference configured with the flags TZ_MEM_SERVICE_RW would require.

4.4.4 Releasing shared memory blocks

When a shared memory block is no longer needed, the client must call the function TZSharedMemoryRelease to release the block. After freeing the block the client must not access the memory again, or *UNDEFINED* behavior will result.

A shared memory block cannot be released while it is still referenced – any active memory references associated with the block must be retired first. In other words, the corresponding call or calls to TZOperationPerform or TZOperationAsyncGetResult must already have returned before releasing the memory block.

4.4.5 Data synchronization

Some implementations may not be able to provide continuous synchronization between the client and the service environment's view of the memory block. The implementation may need to perform various explicit actions in order to achieve synchronization, such as flushing caches or copying data. The implementation is allowed to synchronize data at any time when the range is 'live'. A range is considered 'live' between the following two moments in time:

- The moment any client thread enters the TZEncodeMemoryReference function to create a memory reference to that range.
- The moment any client thread receives notification that the operation into which the memory reference was written has completed.
  - Depending on whether the synchronous or asynchronous interface is being used, this may be the moment at which the corresponding call to the TZOperationPerform returns or the moment at which the corresponding call to the TZOperationAsyncGetResult returns a result other than TZ_PENDING.

The manner in which the implementation synchronizes the data with the service is implementation-dependent, but the semantics of the implementation should in all cases be equivalent to the semantics of the following sequence of operations:

- Each range referenced by the operation and flagged with TZ_MEM_SERVICE_RO or TZ_MEM_SERVICE_RW is copied from client memory into a separate service-visible buffer.
- The service performs the requested operation.
- Each range referenced by the operation and flagged with TZ_MEM_SERVICE_WO or TZ_MEM_SERVICE_RW is copied from the corresponding service-visible buffer back into the corresponding range of client memory.

Note that this does not imply that there should be a separate memory buffer for the service; the most efficient implementations of shared memory will not use a separate buffer and allow true direct access.

In order to ensure correct operation, the implementation must obey the following restrictions:

- Neither the implementation, nor the service, must write to a range except while it is live.
- Where a client makes modifications to memory which is not part of any live range, including any adjacent bytes, the data read or written by the implementation or the service via shared memory blocks must not be affected.
- The implementation must guarantee that any client-to-service synchronization of data takes place before any service-to-client synchronization.

4.4.5.1 Memory alignment hints

In some implementations a service may pass some of the shared memory given to it by a client to a piece of hardware, such as a cryptographic accelerator or a video engine. In these cases the system may function more efficiently if memory references are naturally aligned with hardware features of the memory system, such as the cache line configuration. A system property, TZ_PROPERTY_MEMORY_ALIGN, provides a hint to the client as to the optimal memory alignment and size for the hardware platform on which it is running. If the client can pass a memory range which has a base address that is naturally aligned on a multiple of this value, and the size of the memory range (in bytes) is also multiple of this value, then the Secure world may be able to process commands more efficiently.

4.4.6 Restrictions on clients

While the design of this specification provides a lot of flexibility to the implementation, it also means that clients must not assume behavior beyond guarantees stated above. In particular, it is implementation-defined what order memory is accessed during an operation.

To ensure correct operation and data consistency under all implementations, the client must also obey the following restrictions:

- The client must not read or write to any memory address visible in a live range.
- The client must not create multiple concurrent references to overlapping memory ranges.

Failure to observe these rules may result in incorrect data being read from shared memory by the client or the service at some later point in the software execution.

4.4.7 Example

The following code example shows how shared memory might be used:

```
tz_session_t   sSession;         /* An open client session. */
tz_operation_t sOperation;       /* A prepared invoke or close op. */

uint32_t       uiReturn;
uint32_t       uiSvcReturn;
uint32_t       uiI;

tz_shared_memory_t sShMem;       /* Structure for shared memory block. */

/* Allocate a Shared memory block attached to the session: */
/*   Read-only for service. */
sShMem.uiFlags  = TZ_MEM_SERVICE_RO;

/*   16KB buffer allocated.*/
sShMem.uiLength =  4096 * 16;

/*   Perform allocation. */
uiReturn = TZSharedMemoryAllocate(
               &sSession;
               &sShMem);

/* Handle errors. */
if (uiReturn != TZ_SUCCESS)
{
    /* Handle error... */
    ...
}

/* Write some client data into the block... */
for(uiI = 1024; uiI < 5120; uiI++)
{
    ((uint8_t*)pShMem.pBlock)[uiI] = 0xFF & uiI;
}

/* Encode a memory reference into previously prepared operation: */
/*  Range is 4096-byte long range, starting at offset 1024 bytes. */
TZEncodeMemoryReference(
    &sOperation,
    &sShMem,
    1024,
    4096,
    TZ_MEM_SERVICE_RO);
```

```
/* Invoke service operation. */
uiReturn = TZOperationPerform(
               &sOperation,
               &uiSvcReturn);

/* Handle errors. */
if (uiReturn != TZ_SUCCESS)
{
    /* Handle error... */
    ...
}

/* Release the operation context. */
uiReturn = TZOperationRelease(&sOperation);

/* Handle errors. */
if (uiReturn != TZ_SUCCESS)
{
    /* Handle error... */
    ...
}

/* After operation has completed, the block can be released. */
uiReturn = TZSharedMemoryRelease(&sShMem);

/* Handle errors. */
if (uiReturn != TZ_SUCCESS)
{
    /* Handle error... */
    ...
}
```

4.5 Timeouts and cancelling operations

The client can explicitly cancel any outstanding operation. Additionally, operations may be given an explicit timeout period when they are prepared – after this time, provided the operation has not already completed, it will be cancelled automatically. These features are useful for interrupting long operations, either automatically or when some external event such as a user-driven cancellation occurs.

The client application can explicitly cancel an operation by calling the function TZOperationCancel with the operation structure. When the client requests cancellation of an operation, or when an operation timeout expires, the implementation should take action to accelerate the completion of the operation. However, there is no guarantee that the request will be acted upon. In particular, the service may not support cancellation and refuse to consider the request. Also, the operation may have already completed by the time at which the request is received. In any case, the operation must be deemed completed only when the function TZOperationPerform returns. If the cancel request was acted upon by the service, the function TZOperationPerform must return the error code TZ_ERROR_CANCEL.

A timeout value is represented by an absolute time limit. The use of an absolute limit is useful to apply a timeout to a sequence of several operations. The client can call the function TZDeviceGetTimeLimit to convert a relative timeout value to an absolute time limit encoded in a structure of type `tz_timelimit_t`. Each TZAPI function that accepts a timeout parameter expects a pointer to such a structure. The client can pass the value NULL to denote the absence of a time limit. When the timeout expires, the implementation must automatically cancel the operation. If the timeout request is acted upon the operation must return the error code TZ_ERROR_CANCEL, exactly as if it had been cancelled explicitly.

4.6 Service manager

The Service manager API provides functions that allow a client to enumerate the services installed on the device, to obtain their properties, and (optionally) to download or remove services at run-time.

4.6.1 Opening the Service manager

To use the Service manager a client must call the function TZManagerOpen using an open device connection. There is a single Service manager per device, although a client may open multiple connections with it.

Some functionality of the Service manager, such as the install and removal features, may be accessible only to certain clients. When opening a connection with the Service manager the client can provide "login" information for the purpose of authentication in a similar manner to clients connecting to third-party services. The required login credentials are implementation-defined.

4.6.2 Inspecting services

A client can use the Service manager to inspect the services installed in the device and to retrieve their properties. The function TZManagerGetServiceList returns the list of all of the UUIDs of the services currently installed on the device.

Each service installed on the device has an associated set of properties, some of which are standard and some of which are defined by the service implementation. Each property is a name:value pair, where the name is a 32-bit unsigned integer, and the value is a binary block.

The standard properties which all services must implement are:

- TZ_PROPERTY_SVC_UUID – the UUID of the service
- TZ_PROPERTY_SVC_NAME – the name of the service.
- TZ_PROPERTY_SVC_VENDOR – the supplier of the service.

Additional properties may be defined by the service provider and can be used to advertise characteristics of the service. For example, a property could be used to indicate the version of the client-service protocol that the service exposes. The client can use this information to select a service in a more intelligent manner than by prior knowledge of the service identifier alone.

4.6.3 Download and removal of services

The Service manager may allow clients to download new services to the secure environment and to remove existing services. To access these operations the client must connect to the Service manager with any required login credentials – these are implementation-defined.

The TZManagerDownloadService function downloads a binary buffer, containing the service application, to the secure environment. The data format of the download is implementation-defined and will typically include additional authentication such as a cryptographic signature of the code package. The Service manager may refuse to download the service –, for example if the code signature is invalid, the client login is not suitable, or there is not enough memory in the secure environment. In such a case the function returns TZ_ERROR_ACCESS_DENIED. On successful completion, the identifier of the newly installed service is returned.

A client may use the function TZManagerRemoveService to remove an installed service from the device. When this function succeeds the service no longer appears in the service list and new connections cannot be made to it. The service removal process may imply collaboration between the security environment and the service, although this is implementation-defined. The Service manager may refuse to remove the service. In such a case the function returns TZ_ERROR_ACCESS_DENIED.

4.6.3.1 Detecting support for asynchronous operations

Support for run-time service download and removal operations should be considered a compile time option for the library and the client application. The following macro should be defined in the library header if the implementation provides these functions, and must not be defined if they are unavailable: TZ_BUILD_RUNTIME_SERVICE_CONTROL.

4.7 Memory management

Some functions of the TZAPI return a result which may be of variable size. In such cases the client can invoke a function with a buffer which it knows is big enough, or it can invoke the function with a NULL buffer which prompts the implementation to return the required size of the buffer.

Here is an example that shows how to retrieve the list of services installed on the system:

```
tz_session_t sSvcMgrSession;     /* An open Service manager session. */

tz_return_t uiReturn;
tz_uuid_t * psUUIDList = NULL;
uint32_t    uiListLen, uiI;

/* Call with a NULL list to query the number of installed services. */
uiReturn = TZManagerGetServiceList(
               &sSvcMgrSession,  /* The Service manager session. */
               NULL,             /* NULL list, so only query length. */
               &uiListLen);      /* Adx of var to hold list length. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Something went wrong! */
    goto exit;
}

psUUIDList = malloc(uiListLen * sizeof(tz_uuid_t));

/* Call with a NULL list to query the number of installed services. */
uiReturn = TZManagerGetServiceList(
               &sSvcMgrSession,  /* The Service manager session. */
               psUUIDList,       /* List of length to hold values. */
               &uiListLen);      /* Adx of var to hold list length. */

/* Handle error. */
if (uiReturn != TZ_SUCCESS)
{
    /* Something went wrong – handle error. */
    goto exit;
}

for(uiI = 0; uiI < uiListLen; uiI++)
{
    /* Do something useful with the service UUIDs. */
    ...
}

/* Tidy up. */
exit:

free(psUUIDList);
```

Unlike previous version of the API, this version of the TZAPI is designed so that the client is responsible for any memory allocation (excluding internal structures which are not visible to the client at all). This allows a client to optimally place structures and buffers in memory, including placing them on the calling stack.

4.8 Asynchronous operations

In some operating systems multi-threading is not the preferred way to perform concurrent operations. In these systems any concurrent operations are started by a non-blocking function call, and their completion is signaled using an event or by signaling a semaphore object. This design allows multiple operations to be started and controlled from a single thread of execution, which can improve performance and memory footprint.

4.8.1 Definition of the asynchronous interface

To support this style of programming the TZAPI provides the optional functions TZOperationAsyncStart and TZOperationAsyncGetResult to perform operations asynchronously. The TZOperationAsyncStart function must start the operation in the background and return promptly. When the operation completes, the implementation must signal a platform-specific synchronization object to notify the client that the operation is complete. The client can then call the function TZOperationAsyncGetResult to retrieve the result of the operation.

To call the function TZOperationAsyncStart, the client must pass a pointer to a platform-specific synchronization object in the parameter `pSyncObj`. The implementation must signal this synchronization object only once for each asynchronous operation. The type of this parameter in the function prototype is `void*` but this generic pointer must point to a synchronization object the type of which must be precisely defined for each operating system on which the TZAPI is implemented:

- On SymbianOS, the client must pass a pointer to an object of class `TRequestStatus`; see **[TReqStatus]**. When the operation started by TZOperationAsyncStart completes, the implementation stores the value KErrNone in the request status and signals the requesting thread's semaphore.
- An implementation may support asynchronous operations on other operating systems, but this specification does not yet provide the definition of the parameter `pSyncObj` for these cases.

Other than being asynchronous, calling TZOperationAsyncStart followed by a call to TZOperationAsyncGetResult is functionally equivalent to a call to TZOperationPerform. In particular, operations started with TZOperationAsyncStart can be canceled or timed out, subject to the same conditions that apply to blocking operations.

The function TZOperationAsyncStart returns no error code. All errors are returned by the function TZOperationAsyncGetResult. This function has an input parameter which is the operation handle that was passed to the previous call to TZOperationAsyncStart. When the operation completes the value returned, and the output parameters set by TZOperationAsyncGetResult, are the same as those the function TZOperationPerform would have returned.

4.8.2 Implementation choices

Support for asynchronous operations is optional, subject to fulfillment of the following requirements:

- All implementations must support synchronous operations.
- All implementations running on SymbianOS must support asynchronous operations.

4.8.3 Detecting support for asynchronous operations

Support for asynchronous operations should be considered a compile time option for the library and the client application. The following macro should be defined in the library header if the implementation provides asynchronous options, and must not be defined if they are unavailable: TZ_BUILD_ASYNCHRONOUS.

4.8.4 Usage example

The following example demonstrates the usage of asynchronous operations on SymbianOS:

```
tz_session_t sOperation;        /* A previously prepared operation. */
tz_return_t  uiReturn;          /* System return value. */
tz_return_t  uiSvcReturn;       /* Service return value. */

TRequestStatus sStatus;         /* SymbianOS request status object. */
TBool          yOpActive;       /* SymbianOS operation state. */

/* Start the operation. */
TZOperationAsyncStart(
    &sOperation,                /* A prepared operation. */
    &sStatus);                  /* SymbianOS synchronization object. */
```

```
yOpActive = ETrue;

/* Run other tasks in this thread... */
...

-------------------------------------------------------------------------
/* Program's main event-handling loop: */

FOREVER
{
    User::WaitForAnyRequest();

    if ( yOpActive && (sStatus != KRequestPending) )
    {
        yOpActive = EFalse;
        uiReturn = TZOperationAsyncGetResult(
                       &sOperation,
                       &uiSvcReturn);
        if (uiReturn == TZ_SUCCESS)
        {
            /* Process the service's response... */
            ...
        }
        else
        {
            /* Handle error... */
            ...
        }
    }

    /* Handle any other events which the program may be expecting... */
    ...
}
```

4.9 Error handling

This section describes the error management implemented by the TZAPI.

All return values and error codes returned by the API are defined to be of type `tz_return_t`.

The TZAPI implementation may return a limited set of return codes, defined by the table in section 5.5.2. These cover common run-time errors, programmer errors, and specific return codes for encoder and decoder errors. Note that this specification does not define values for each of these constants; only the macro name is defined. An implementation is free to assign the underlying numeric value of each of these constants.

Most of the functions in the TZAPI return a single error code, but the operation invocation commands also return a service return code. The service return code will be set if the main function return is TZ_ERROR_SERVICE. The value of this service return code is defined by the service protocol and is therefore implementation-defined.

4.10 Multi-threading and concurrency

Support for multi-threaded operation should be considered a compile time option for the library and the client application. The following macro should be defined in the library header if the implementation allows TZAPI functions to be called from multiple threads concurrently, and must not be defined if they cannot: TZ_BUILD_MULTITHREADED.

4.11 Robustness and security

The implementation must not assume that the client application is safe or stable. The security model assumed by TZAPI is that anything outside the security environment is not trusted; it must be therefore assumed that the client will be hostile and deliberately attack the implementation of the TZAPI, and the security services, by all possible means.

This specification defines the behavior of the implementation when the client calls the functions in a valid sequence and with valid parameters. Otherwise, the behavior is said to be *UNDEFINED*; in these cases the implementation can compromise the stability of the client environment but it is required that it still preserve the robustness and security of the security environment. For example, calling a function with a NULL pointer may provoke a client crash, but cannot cause the secure environment to crash or become unstable.

5 TYPES AND CONSTANTS

5.1 Notation

This specification uses the notation outlined in this section.

5.1.1 Hungarian notation

This specification uses the following Hungarian notation prefixes for variables:

Prefix	Description
`i`	Integer
`s`	Structure
`p`	Pointer
`a`	Array
`k`	Constant
`u`	Unsigned

Table 1: TrustZone API Hungarian notation

In addition to the Hungarian notation noted above, pointer parameters for API functions are also tagged with the `IN`, `OUT` and `INOUT` macros. The contents of the variable pointed to by an `IN` or `INOUT` pointer must contain some usable data on function entry. The contents of a variable pointed to by a pointer tagged with `OUT` or `INOUT` will be modified by the function implementation and will be written on output. By definition a pointer tagged with `IN` will also be a pointer to constant data, although the `const` qualifier is separately stated for clarity.

5.1.2 Basic types

This specification makes use of a subset of the integer types defined in the C99 standard **[ISO/IEC 9899:1999]**. The following basic types are used:

- `uint32_t`: unsigned 32-bit integer.
- `uint16_t`: unsigned 16-bit integer.
- `uint8_t`: unsigned 8-bit integer.

5.1.3 Reserved naming

To enable compatibility with future versions of this specification the function prefix "`TZ`", the constant prefix "`TZ_`" and the type prefix "`tz_`" should be considered reserved for future use. Implementations of the TZAPI and client applications using the TZAPI should not use these prefixes for their locally defined extensions to the API.

It is suggested that the "`TZI_`" prefix is used for implementation-defined constants and "`tzi_`" is used for implementation-defined types.

5.2 Managing TZAPI data structures

The TZAPI makes use of a number of data structures, introduced in the following sections of this specification which are related to, and dependent on, each other. Many designs will store pointers to cross-reference these structures; a client application cannot copy a structure and assume that it will function correctly. Any attempt to move an implementation-defined structure specified in section 6.4 will result in *UNDEFINED* behavior.

Structures in section 5.3 may be copied, but must remain valid while they are in use – i.e. a function in which they are a parameter is still executing.

5.3 Pre-defined types

This specification makes use of the following types that are completely defined by this specification:

5.3.1 tz_return_t

The `tz_return_t` type is used to contain all of the return codes used by the API.

```
typedef uint32_t tz_return_t;
```

5.3.2 tz_state_t

The `tz_state_t` type is used to contain all of the states used by the API. Note that not all structures can use all of the defined states.

```
typedef uint32_t tz_state_t;
```

It should be noted that TZ_STATE_UNDEFINED is not be an explicit state in the state machine. A structure defined as TZ_STATE_UNDEFINED can have any value for the state field; for example, the implementation cannot assume that it will be zero. For this reason when the specification states that a function "*sets the state the TZ_STATE_UNDEFINED*" it is not actually necessary to set the state field at all, although many implementations may choose to do this for client robustness and debug reasons.

5.3.3 tz_uuid_t

The `tz_uuid_t` type is used to contain the Universally Unique IDentifier (UUID) type as defined in **[RFC4122]**. A UUID is the mechanism by which a service is identified.

```
typedef struct tz_uuid_t
{
    uint32_t   uiTimeLow;
    uint16_t   uiTimeMid;
    uint16_t   uiTimeHiAndVersion;
    uint8_t    auiClockSeqAndNode[8];
} tz_uuid_t;
```

Client code can either hard-code the UUID of the service it wants to connect to, or find the service by querying the Service manager. The UUID 79B77788-9789-4A7A-A2BE-B60155EEF5F3 can be hard-coded using the following C code:

```
static const tz_uuid_t ksMyUUID =
{
    0x79B77788, 0x9789, 0x4A7A,
    {
        0xA2, 0xBE, 0xB6, 0x01, 0x55, 0xEE, 0xF5, 0xF3
    }
};
```

5.3.4 tz_login_t

When a client opens a session with a service it may be required to provide evidence of its identity before the service gives access to some parts of its functionality – this concept is called login. The login phase is controlled by the contents of a `tz_login_t` structure specified by the client when preparing the open session operation; this prompts the implementation to make the requested credentials available to the service. As the client is an untrusted entity, the system must provide the credentials by a side channel which cannot be tampered with by the client.

```
typedef struct tz_login_t
{
    uint32_t   uiType;
    void *     pBuff;
    uint32_t   uiBuffLen;
} tz_login_t;
```

Field description

`uiType`: A bit field specifying which login credentials must be provided to the service. This must be one of the following options:

- TZ_LOGIN_PUBLIC: no credentials are provided.

OR

- One or more of the following flags:
    - TZ_LOGIN_CLIENT_DATA: supply the client buffer specified by `pBuff` and `uiBuffLen`.
    - TZ_LOGIN_USER: supply the identity of the "user" executing the client.
    - TZ_LOGIN_GROUP: supply the identity of the "group" executing the client.
    - TZ_LOGIN_NAME: supply the "name" of the client executable. This may include path information to strengthen the differentiation between executables with the same name.
    - TZ_LOGIN_DIGEST: supply the cryptographic "digest" of the currently running client process to the service. This enables the service to compare the digest with a known good value to ensure that the client has not been tampered with.

OR

- TZ_LOGIN_ALL: A utility constant which indicates that all of the available login flags have been specified.

`pBuff`: A buffer of login information sent to the service when the TZ_LOGIN_CLIENT_DATA login flag is specified. The required content of this buffer is defined by the client-service protocol defined by the service that the client is attempting to connect to.

`uiBuffLen`: The length of the buffer in bytes. This field should be zero if `pBuff` is NULL.

All implementations of the TrustZone API must accept each of the login credential flags described above; however the client developer must be aware that in some systems these flags may have little or no value. For example, many embedded operating systems have no notion of "user" or "path" as all applications are simply compiled into the operating system image and executed at boot time. The precise function of each of the credential flags is therefore implementation defined.

It most cases it will be desirable for the service to check that the client is authentic in a manner that is agnostic to the non-trusted operating environment. This allows a development paradigm where the security environment and its services can be implemented and validated once before being subsequently deployed alongside any rich operating environment. Each operating system will provide different credentials for each login type, so the service cannot check against static values. The design principle here is that the client can provide a set of known-good values for the login which have been signed by a trusted authority, and send them to the service within this buffer using TZ_LOGIN_CLIENT_DATA. The service can hard code the public key of the trusted authority and as such can check the signature of the client provided data. The service can compare the authenticated client data against the actual login credentials provided by the operating environment in order to make a safe OS-agnostic decision about the validity of the login. When deploying the client to a new non-trusted operating environment a new service with retargeted login properties suitable for the host OS is no longer required, the developer can now simply deploy an updated signed data file as part of the client installation.

5.3.5 tz_property_t

The `tz_property_t` type is used to contain the name:value pairs which represent system and service properties. Each property consists of two parts, a fixed size 32-bit numeric name and a variable length binary block which represents the property's value. The value of a property can be dynamically retrieved by a client by sending the property name to TZManagerGetServiceProperty function, or the TZManagerGetServicePropertyList function.

```
typedef struct tz_property_t
{
    uint32_t   uiNamespace;
    uint32_t   uiName;
    void *     pValue;
    uint32_t   uiLength;
} tz_property_t;
```

Field description

`uiNamespace`: The numeric namespace of properties.

Refer to section 5.3.5.1 for further details and namespace range usage.

`uiName`: The numeric name of the property.

`pValue`: A binary buffer of containing the property value. The content of this buffer is defined by the specification of the property that is loaded.

`uiLength`: The length of the binary buffer in bytes, or zero if `pValue` is NULL.

5.3.5.1 Namespaces

Namespaces are used to allow multiple overlapping sets of names to exist in a single service implementation and to enable client code to disambiguate these names.

Namespace	Description
`0x00000000 – 0xEFFFFFFF`	This namespace range is free for use for any property.
`0xF0000000`	This namespace, defined by the constant TZ_NAMESPACE_SYSTEM, is used by all System properties returned by TZSystem functions. Services must not use this namespace for their own properties.
`0xF0000001`	This namespace, defined by the constant TZ_NAMESPACE_SERVICE, is used to contain service-specific properties that are exposed to the client application.
`0xF0000002`	This namespace, defined by the constant TZ_NAMESPACE_CLIENT, is used to contain client properties exposed to the service or the system by an implementation-defined mechanism. Services must not use this namespace for their own properties.
`0xF0000003`	This namespace, defined by the constant TZ_NAMESPACE_CONFIG, is used to contain service-specific configuration properties which can be interpreted by the Secure world system code or by the service itself. Properties allocated in this namespace are not exposed to client applications, and must never be returned by the TZManagerGetServicePropertyList or TZManagerGetServiceProperty functions.
`0xF0000004 - 0xFFFFFFFF`	*Reserved for future use.* Services must not use this namespace for their own properties.

Table 2: TrustZone API property namespaces

It is recommended that a service only uses TZ_NAMESPACE_SERVICE and TZ_NAMESPACE_CONFIG for its properties unless it exposes a standardized API which is queried and managed using properties. For example, a standardized cryptographic interface which is implemented by multiple services may expose the protocol version and functions it supports within the namespace `0x01234567`.

5.3.6 tz_property_name_t

The `tz_property_name_t` structure is used by the functions which list the properties exposed to a client by the system or by a service.

```
typedef struct tz_property_name_t
{
    uint32_t   uiNamespace;
    uint32_t   uiName;
} tz_property_name_t;
```

Field description

`uiNamespace`: The numeric namespace of properties.

Refer to section 5.3.5.1 for further details and namespace range usage.

`uiName`: The numeric name of the property.

5.4 Implementation-defined types

The following structures are used by the TrustZone API, but are only partially defined by this specification. All of these structures contain a portion that is implementation-defined. A specific implementation of the TrustZone API must fully define these structures and client code must be compiled against the correct definitions for their implementation.

Client code must never modify the contents of these structures directly unless explicitly specified in the specification. The only cases where this is allowed are:

- Setting the state of a device, session, operation or shared memory structure to TZ_STATE_INVALID before using them for the first time. This is useful for simplifying error handling – see section 5.6.2 for details.
- Setting the required input state for the shared memory structure when calling the functions that allocate a new block.

Modifying these fields at any other time, including moving them in memory or accessing any of the implementation-defined fields, will result in *UNDEFINED* behavior.

5.4.1 tz_timelimit_t

The `tz_timelimit_t` structure is used to contain an absolute time since an arbitrary origin. This time value is used to provide a timeout facility to the Open and Invoke operations of the API.

```
typedef struct tz_timelimit_t
{
   /* Implementation-defined. */
   ...
} tz_timelimit_t;
```

All instances of this structure must be created through the use of the function TZDeviceGetTimeLimit, otherwise the behavior is undefined. All time values contained in these structures are absolute values relative to the time that the function TZDeviceGetTimeLimit was called to create them.

The arbitrary origin for the timeout is local to each device. The TZAPI implementation must guarantee that this time origin does not change once the device has been created. Use of a time limit structure in a device other than in the one in which it was created results in undefined behavior.

Note: Time limits in the TrustZone API must only be considered as hints to the underlying implementation; the client must not assume that the system honors the time limit.

5.4.2 tz_device_t

The `tz_device_t` structure is used to contain control information related to the device. Some fields are defined by this specification, but the implementation may add additional data to these structures to suit its needs.

```
typedef struct tz_device_t
{
    tz_state_t uiState;
    struct
    {
        /* Implementation-defined. */
        ...
    } sImp;
} tz_device_t;
```

The device structure conforms to a state machine as the client calls the TZAPI functions. This state machine is outlined in Figure 4. If a client calls functions which are outside the scope of the state machine, the behavior is *UNDEFINED*.

The following notation is used for all state machines in this specification:

- Cornered rectangles = device states at any instance in time.
- Tapered rectangles = explicit client actions – i.e. calling functions.
- Diamond = decision points when transitioning states.
- Rounded rectangles = terminating state.

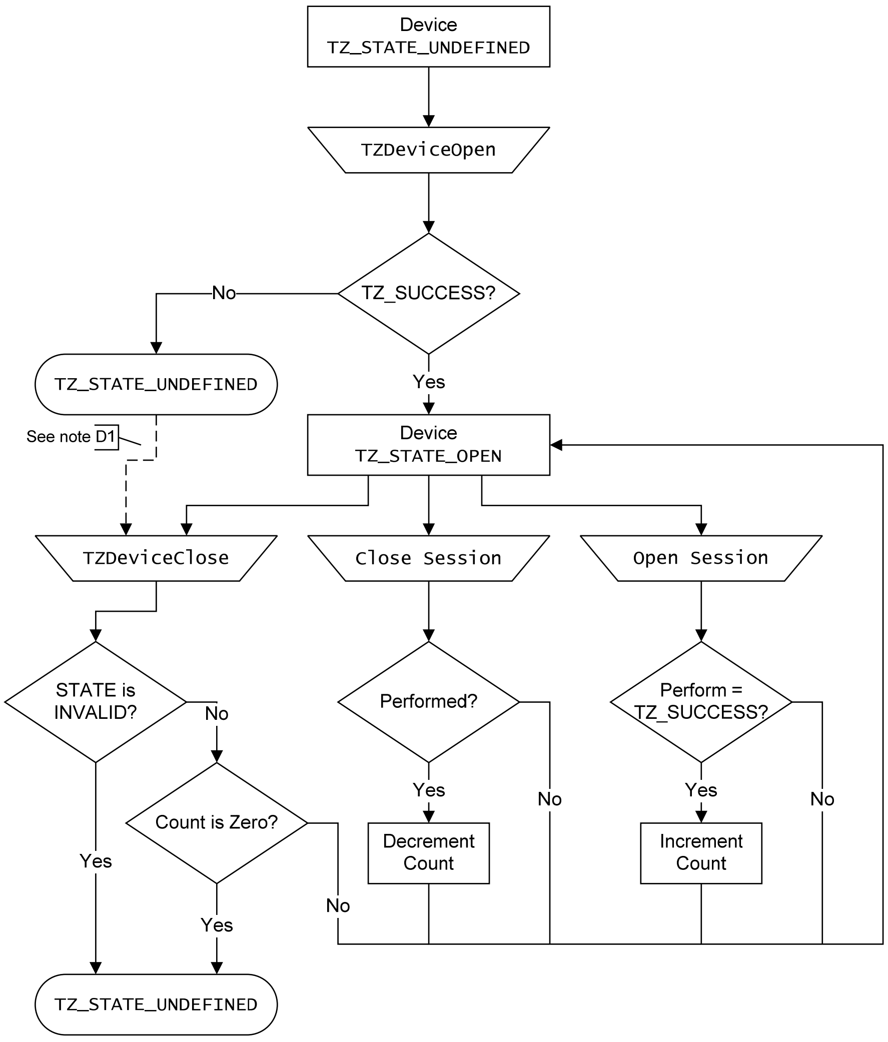

Figure 4: Device state machine

Note D1: At this point the client can throw away the device if it so wishes. For programmer convenience the device structure may optionally be passed to TZDeviceClose, but this is not mandatory.

Note D2: The "Close Session" and "Open Session" actions are comprised of a number of client TZAPI calls; in this case only the perform operation (synchronous or asynchronous) impacts the device state machine.

5.4.3 tz_session_t

The `tz_session_t` structure is used to contain control information related to a session between a client and a service. Some fields are defined by this specification, but the implementation may add additional data to these structures to suit its needs.

```
typedef struct tz_session_t
{
    tz_state_t uiState;
    struct
    {
        /* Implementation-defined. */
        ...
    } sImp;
} tz_session_t;
```

The session structure conforms to a state machine as the client calls the TZAPI functions. This state machine is outlined in Figure 5. If a client calls functions which are outside the scope of the state machine, the behavior is *UNDEFINED*.

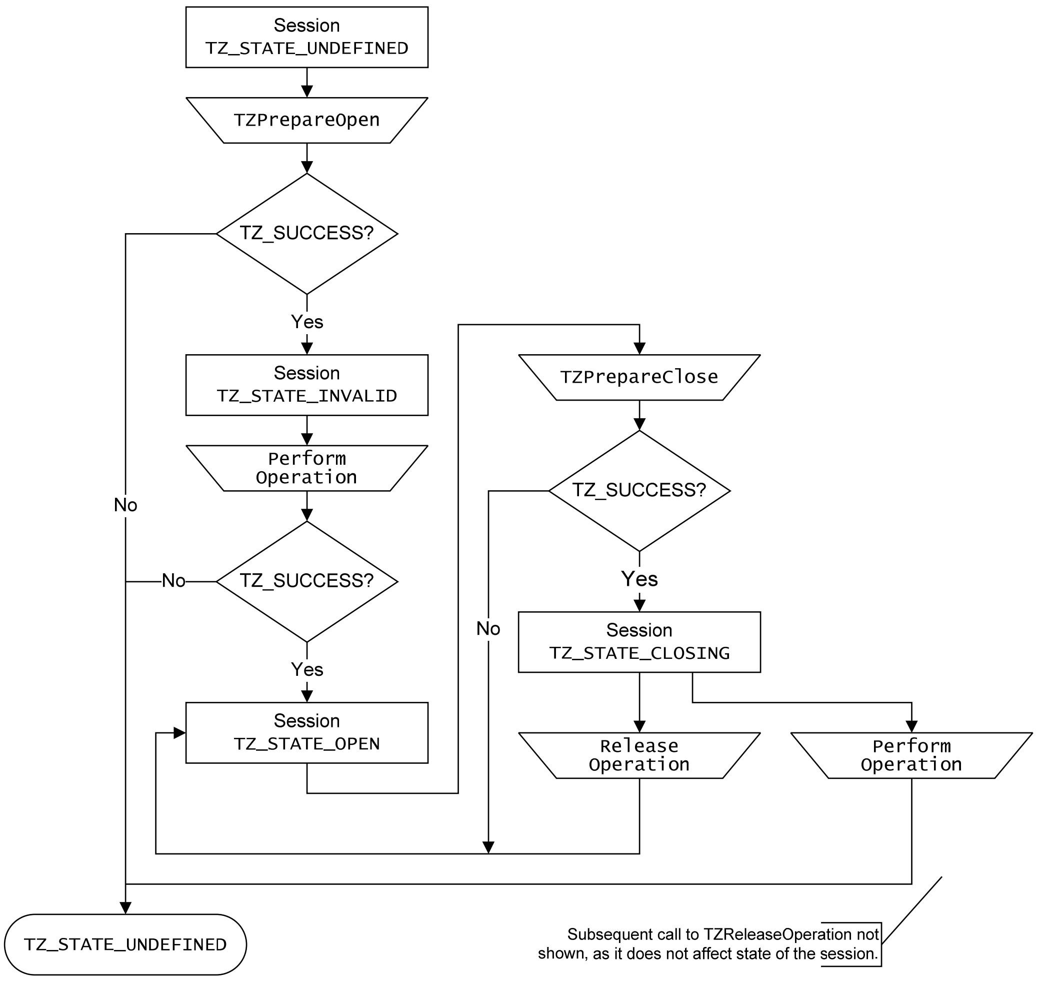

Figure 5: Session state machine

5.4.4 tz_operation_t

The `tz_operation_t` structure is used to contain control information related to an operation that is to be invoked with the security environment. Some fields are defined by this specification, but the implementation may add additional data to these structures to suit its needs.

```
typedef struct tz_operation_t
{
    tz_state_t uiState;
    struct
    {
        /* Implementation-defined. */
        ...
    } sImp;
} tz_operation_t;
```

The operation structure conforms to a state machine as the client calls the TZAPI functions. There are two state machines: one for open and invoke operations, and one for close operations.

5.4.4.1 Open and invoke operations

The state machine for open and invoke operations is outlined in Figure 6.

If a client calls functions which are outside the scope of this state machine, the behavior is *UNDEFINED*.

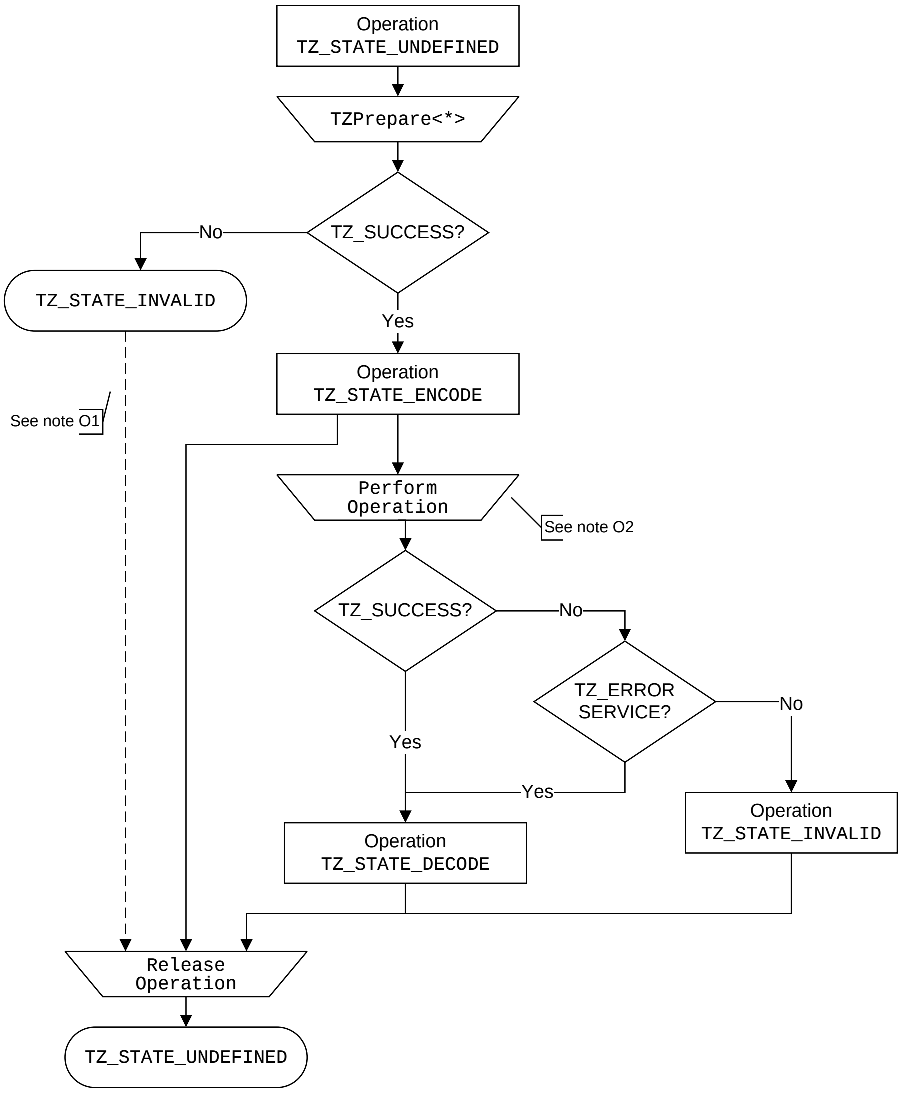

Figure 6: Open operation and invoke operation state machine

Note O1: At this point the client can throw away the operation if it so wishes. For programmer convenience the operation structure may optionally be passed to TZOperationRelease, but this is not mandatory.

Note O2: During the execution of the perform function, or pair of functions in the case of the asynchronous API, the state of the operation is set to TZ_STATE_RUNNING or TZ_STATE_RUNNING_ASYNC depending on the perform function used. In the synchronous case the running state is not visible to the calling thread; by the time the function returns the state will have transitioned again, but the running state may be visible to other client threads using the TZAPI.

5.4.4.2 Close operations

The close operation has a different state machine, which removes the ability to call the encoder and decoder functions. This state machine is given in Figure 7.

If a client calls functions which are outside the scope of the state machine, the behavior is *UNDEFINED*.

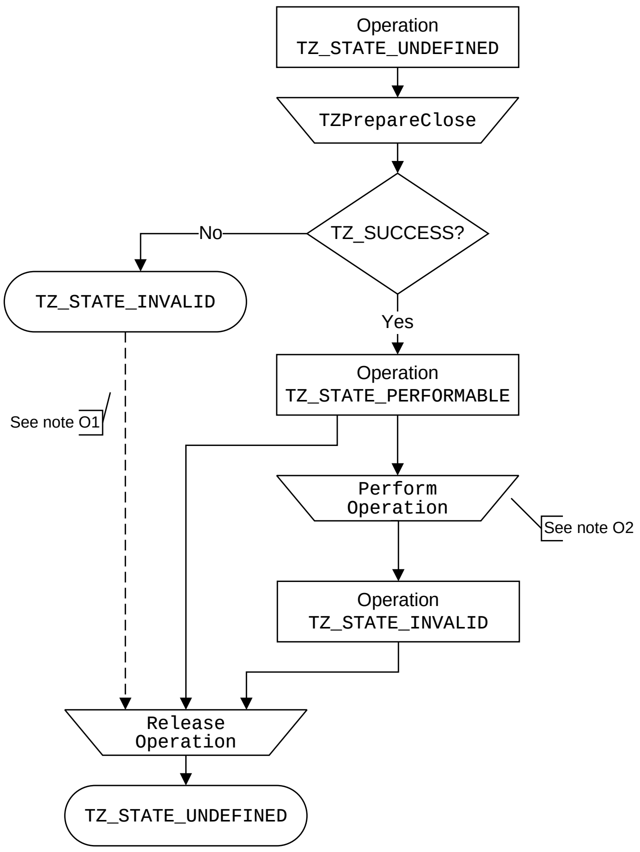

Figure 7: Close operation state machine

Note O1: As for Figure 6.

Note O2: As for Figure 6.

5.4.5 tz_shared_memory_t

The `tz_shared_memory_t` structure is used to contain control information related to a block of shared memory that is mapped between the client and the service.

```
typedef struct tz_shared_memory_t
{
    tz_state_t uiState;
    uint32_t   uiLength;
    uint32_t   uiFlags;
    void*      pBlock;
    struct
    {
        /* Implementation-defined. */
    } sImp;
} tz_shared_memory_t;
```

Field Description

`uiState`: The state of this structure. For shared memory only the following states are used:

- TZ_STATE_INVALID: Shared memory block is not valid, but in a known state which can be freed.
- TZ_STATE_OPEN: Shared memory block is valid and references to it can be encoded in structured messages.
- TZ_STATE_UNDEFINED: Pseudo state covering all other behavior. A structure in this state must not be used unless explicitly specified, otherwise *UNDEFINED* behavior may occur.

`uiLength`: The length of the shared memory block in bytes. Should not be zero.

`uiFlags`: The sharing flags of the shared memory block, indicating direction of data sharing.

Note that these access flags cannot usually be enforced by the hardware. If a client or a service ignores the flags specified for a shared memory block, or a corresponding memory reference, *UNDEFINED* behavior results.

Exactly one of the following flags must be specified:

- TZ_MEM_SERVICE_RO: The service can only read from the memory block.
- TZ_MEM_SERVICE_WO: The service can only write to the memory block.
- TZ_MEM_SERVICE_RW: The service can both read from and write to the memory block.

`pBlock`: The pointer to the block of shared memory.

5.5 Constants

In this specification the following constants are defined for use by the system and client applications using the API. For most constants this specification only defines the constant macro names; each implementation is free to choose the value of each constant. Client code must therefore not interpret the value of the constants, but refer to them only by the macro name.

5.5.1 Compiler build configuration macros

The following macros should be defined if the feature they represent is supported in the library implementation:

Constant	Description
TZ_BUILD_RUNTIME_SERVICE_CONTROL	The Service manger runtime service download and removal functions are support.
TZ_BUILD_ASYNCHRONOUS	The asynchronous operation invocation functions are supported.
TZ_BUILD_MULTITHREADS	The library supports multithreaded use of the API by a client application.

Table 3: TrustZone API compile time macros

5.5.2 Return codes

All return codes in this specification are of type `tz_return_t`. Return codes can be divided into two categories – those that are returned by the TZAPI or secure environment and those that are returned by the service that the client is communicating with. The system must only throw standard return codes defined in Table 4. A service can return any value; all service error returns and their values should be defined as part of the client-service protocol.

Constant	Description
TZ_SUCCESS	The operation succeeded.
TZ_PENDING	The asynchronous operation is still pending.
TZ_ERROR_ACCESS_DENIED	Access has been denied, or the item cannot be found.
TZ_ERROR_BUSY	The system is busy.
TZ_ERROR_CANCEL	The execution was cancelled.

Constant	Description
TZ_ERROR_COMMUNICATION	There is a system communication error.
TZ_ERROR_DECODE_NO_DATA	The decoder ran out of data.
TZ_ERROR_DECODE_TYPE	The decoder hit a type error.
TZ_ERROR_ENCODE_FORMAT	The encoded data is of a bad format. This generally applies to incorrectly specified memory references.
TZ_ERROR_ENCODE_MEMORY	The encoder ran out of memory.
TZ_ERROR_GENERIC	An unspecified error has occurred.
TZ_ERROR_ILLEGAL_ARGUMENT	A bad parameter has been specified.
TZ_ERROR_ILLEGAL_STATE	A state machine has been violated.
TZ_ERROR_MEMORY	There is not enough memory to perform the operation.
TZ_ERROR_NOT_IMPLEMENTED	The functionality is not implemented.
TZ_ERROR_SECURITY	There is a security error.
TZ_ERROR_SERVICE	The service has returned an error in the service return code variable.
TZ_ERROR_SHORT_BUFFER	The input buffer is not long enough.
TZ_ERROR_UNDEFINED	The implementation has reached an *UNDEFINED* condition.

Table 4: TrustZone API standard return codes

5.5.3 State machine state constants

The following states exist in the various state machines used in this specification. Each state is associated with the following constant:

Constant	Description
TZ_STATE_UNDEFINED	Structures in the *UNDEFINED* state may have any value for their state constant; it may not exist as an explicit value. *Clients should never make use of this constant, although an implementation may use it internally for debugging purposes.*
TZ_STATE_INVALID	The state is in a safe invalid state.
TZ_STATE_OPEN	The state is open.
TZ_STATE_CLOSING	The state is closing, but not yet closed.
TZ_STATE_ENCODE	The state an operation that is not running and can accept data to be encoded.
TZ_STATE_PERFORMABLE	The state of an operation that is not running, but which cannot accept data to be encoded. This state applies only to close operations.
TZ_STATE_RUNNING	The state of an operation that is executing synchronously.
TZ_STATE_RUNNING_ASYNC	The state of an operation that is executing asynchronously.
TZ_STATE_DECODE	The state of an operation that can have data read using the decoder functions.

Table 5: TrustZone API state constants

5.5.4 Service properties

The following mandatory properties are reserved for built-in service properties:

Constant	Description
TZ_PROPERTY_SVC_UUID	The UUID of the service, provided as an array `uint8_t` values of length 16. The array is not zero-terminated.
TZ_PROPERTY_SVC_NAME	The human-readable name of the service, provided as an array of `uint8_t` values of variable length. The array is zero-terminated.
TZ_PROPERTY_SVC_VENDOR	The human-readable name of the service vendor, provided as an array of `uint8_t` values of variable length. The array is zero-terminated.

Table 6: TrustZone API service property constants

A service may specify custom properties – these are not defined by this specification and the numeric values of these service-local names may overlap with the namespace of other services.

5.5.4.1 Property namespaces

The following standard namespaces are defined in this version of the specification. Note that these constants have a defined value, specified in Table 2:

Constant	Description
TZ_NAMESPACE_SYSTEM	This namespace is used for all system properties returned by the TZSystem functions.
TZ_NAMESPACE_SERVICE	This namespace is used for general service properties.
TZ_NAMESPACE_CLIENT	This namespace is reserved for client properties and should not need to be used by a client using this API.
TZ_NAMESPACE_CONFIG	This namespace is reserved for private configuration properties which are not exposed to the client.

Table 7: TrustZone API property namespaces

5.5.5 System properties

The following mandatory properties are reserved for system properties:

Constant	Description
TZ_PROPERTY_SYS_API_VERSION	The version number of the API. This property's value is a single `uint32_t`. The top 16 bits of the value are the major version, the bottom 16 bits are minor version. The value for this version of the specification will be `0x00030000`.
TZ_PROPERTY_SYS_MEMORY_ALIGN	The shared memory alignment constraints on memory references required for efficient memory coherency with hardware devices. This property's value is a single `uint32_t`. Value is implementation-defined.

Table 8: TrustZone API system property constants

5.5.6 Login flags

The following flags are used for login credentials:

Constant	Description
TZ_LOGIN_PUBLIC	No login is to be used.
TZ_LOGIN_CLIENT_DATA	A buffer of client data is to be provided.
TZ_LOGIN_USER	The user executing the application is provided.
TZ_LOGIN_GROUP	The user group executing the application is provided.
TZ_LOGIN_NAME	The name of the application is provided; may include path information.
TZ_LOGIN_DIGEST	The digest of the client application is provided.
TZ_LOGIN_ALL	A utility constant indicating all available login types should be used.

Table 9: TrustZone API login flags

5.5.7 Shared memory flags

The following flags are used for shared memory:

Constant	Description
TZ_MEM_SERVICE_RO	Service can only read from the memory block.
TZ_MEM_SERVICE_WO	Service can only write from the memory block.
TZ_MEM_SERVICE_RW	Service can read and write from the memory block.

Table 10: TrustZone API memory sharing flags

5.5.8 Decode data types

The following constants are returned by the function TZDecodeGetType:

Constant	Description
TZ_TYPE_NONE	There is no more data in the decode stream.
TZ_TYPE_UINT32	The next data type in the stream is a uint32_t.
TZ_TYPE_ARRAY	The next data type in the stream is an array.

Table 11: TrustZone API memory sharing flags

5.6 Conventions used by TZAPI functions

The following conventions are used by all of the functions in the TrustZone API; these conventions are generally only described here and are not repeated for each function in the main body text of the specification.

5.6.1 Use of structures in the UNDEFINED state

Most of the TrustZone API functions take one or more of the `tz_device_t`, `tz_session_t`, `tz_operation_t` or `tz_shared_memory_t` structures, each of which has an associated state machine as defined above. The implementation treats all instances of these structures as being in the state TZ_STATE_UNDEFINED until they have been initialized by an appropriate API function.

- TZDeviceOpen initializes the `tz_device_t` structure.
- TZOperationPrepareOpen initializes the `tz_session_t` structure.
- TZOperationPrepare* initializes the `tz_operation_t` structure.

TZSharedMemoryAllocate or TZSharedMemoryRegister initializes the `tz_shared_memory_t` structure.

Use of any input structure which is in the state TZ_STATE_UNDEFINED results in *UNDEFINED* implementation behavior unless the use of an input structure in this state is explicitly allowed by the function definition. The scope of the *UNDEFINED* behavior may include cases that corrupt client memory and consequently crash the client. The system design must ensure that the security of the secure environment is not affected by invalid API usage by a client.

In the functions which do accept a structure in the state TZ_STATE_UNDEFINED, the implementation must not interpret any of the fields of the structure without first overwriting them with a known valid value.

Some implementations may choose to safely handle some of the behavior that is classified as *UNDEFINED* in this specification by returning an error code. In these situations it is required that the error TZ_ERROR_UNDEFINED is returned – this indicates clearly to the client that it is doing something that is liable to break on other implementations of this API.

5.6.2 Simple resource free semantics using a defined INVALID state

It is often desirable for client code implementation to have a single exit block for both success and error conditions. This improves code reliability, clarity and code size. One example of this can be seen in standard C where the memory "free" function accepts a NULL pointer. It is quite common to see code structured thus:

```
void f(x)
{
    uint8_t *   pBuff1;
    uint8_t *   pBuff2 = NULL;

    pBuff1 = (uint8_t *)malloc(16);
    if(pBuff1 == NULL)
    {
        goto exit;
    }

    pBuff2 = (uint8_t *)malloc(16);
    if(pBuff2 == NULL)
    {
        goto exit;
    }

    /* Write useful code using temporary buffers here. */
    ...

    /* All exit paths start here. */
exit:

    free(pBuff1);
    free(pBuff2);
}
```

This code functions because the code can set `pBuff2` to a defined safe state at the start of the function; even if the allocation of `pBuff1` fails, the free of `pBuff2` is guaranteed to work.

To enable a similar coding style for the TrustZone API, the `tz_device_t`, `tz_session_t` and `tz_operation_t` structures all contain a defined state variable, `uiState`, which can be manually set by client code to the state TZ_STATE_INVALID at the start of the function execution. The functions which free the TZAPI structures are specified to accept a structure in the invalid state and must safely return TZ_SUCCESS.

6 FUNCTIONS

This section of the document is the specification of the functions that are provided by the TrustZone API.

6.1 Control functions

This section outlines the control functions that are used in the implementation. These form the main body of the API and deal with the creation of a session between a client and a service, the issuing of commands in that session, and the creation of shared memory mappings.

6.1.1 TZDeviceOpen

```
tz_return_t TZDeviceOpen(
    IN      void const *    pkDeviceName,
    IN      void const *    pkInit,
    OUT     tz_device_t *   psDevice)
```

Description

This function opens a connection with the device in the underlying operating environment that represents the secure environment. When the client no longer requires the device it must call TZDeviceClose to close the connection and free any associated resources.

This function accepts a pointer to a `tz_device_t` structure assumed to be in the TZ_STATE_UNDEFINED state.

On success this function must set the device structure `*psDevice` to the state TZ_STATE_OPEN with a session count of zero. On failure, the device is set to the state TZ_STATE_INVALID.

It is possible to create multiple concurrent device connections from a single client. The number of devices that can be supported globally within the entire system, or locally within a single client, is implementation-defined.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Parameters

`pkDeviceName`: An implementation-defined binary buffer, used to identify the underlying device to connect to. If this is NULL, the implementation will use an internally defined default device name.

`pkInit`: An implementation-defined binary block used to configure the implementation. If this is NULL, the implementation will use predefined default values for the library configuration.

`psDevice`: A pointer to the device structure.

Returned Value

TZ_SUCCESS: The device was successfully opened.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.2 TZDeviceClose

```
tz_return_t TZDeviceClose(
    INOUT   tz_device_t *   psDevice)
```

Description

This function closes a connection with a device, freeing any associated resources.

If the passed device is in the state TZ_STATE_INVALID this function must set to the state to TZ_STATE_UNDEFINED and return TZ_SUCCESS.

If the passed device is in the state TZ_STATE_OPEN with a session count of zero this function must delete the device. This operation cannot fail; the function must always set the state to TZ_STATE_UNDEFINED and return TZ_SUCCESS.

If the passed device is in the state TZ_STATE_OPEN with a non-zero session count, this function must return TZ_ERROR_ILLEGAL_STATE and leave the device unmodified.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Parameters

`psDevice`: A pointer to the device to delete.

Returned Value

TZ_SUCCESS: The device is successfully closed.

TZ_ERROR_ILLEGAL_STATE: The device is in the state TZ_STATE_OPEN and has a non-zero session count.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.3 TZDeviceGetTimeLimit

```
tz_return_t TZDeviceGetTimeLimit(
    INOUT   tz_device_t *    psDevice,
            uint32_t         uiTimeout,
    OUT     tz_timelimit_t * psTimeLimit)
```

Description

Calling this function generates a device-local absolute time limit in the structure pointed to by `psTimeLimit` from a timeout value `uiTimeout`. The absolute time limit is equal to the current time plus the specified timeout.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Calling with `psDevice` pointing to a device in the state TZ_STATE_INVALID.

Calling with `psTimeLimit` set to NULL.

Use of the time limit outside of the device in which it was created.

Parameters

`psDevice`: A pointer to the device.

`uiTimeout`: The required relative timeout, in milliseconds.

`pTimeLimit`: A pointer to the time limit structure to populate.

Returned Value

TZ_SUCCESS: the time limit was created successfully.

TZ_ERROR_*: an implementation-defined error code for any other error.

6.1.4 TZOperationPrepareOpen

```
tz_return_t TZOperationPrepareOpen(
    INOUT   tz_device_t*           psDevice,
    IN      tz_uuid_t const *      pksService,
    IN      tz_login_t const *     pksLogin,
    IN      tz_timelimit_t const * pksTimeLimit,
    OUT     tz_session_t*          psSession,
    OUT     tz_operation_t*        psOperation)
```

Description

This function is responsible for locally preparing an operation that can be used to connect with the service defined by the UUID pointed to by `pksService`, using the timeout pointed to by `pksTimeLimit` and the login credentials specified in `pksLogin`.

This function accepts a session and an operation structure assumed to be in the state TZ_STATE_UNDEFINED.

When this function returns TZ_SUCCESS it must increment the session count of the device. The count may subsequently need to be decremented by the TZOperationRelease function – releasing this operation if the corresponding perform operation failed or was never executed.

When this function returns TZ_SUCCESS the operation is set to the state TZ_STATE_ENCODE; this state allows the client to encode a message to be exchanged with the service using the encoder functions of the TZAPI. Once the message has been encoded the client must call the function TZOperationPerform, or the asynchronous equivalent, to issue the open session command to the security environment.

When this function returns TZ_SUCCESS the session must be in the state TZ_STATE_INVALID. The state transitions to TZ_STATE_OPEN after the perform function related to the open session operation has returned TZ_SUCCESS; only at this point does the session become usable. If the perform function returns any error code, the session is not opened and remains in the state TZ_STATE_INVALID.

When this function fails it can return any error code. In these conditions, the state of the device must be unchanged, including the session count. The state of the session is TZ_STATE_UNDEFINED and the operation must be set to TZ_STATE_INVALID.

Note that if the perform function fails, the client must still call TZOperationRelease to release resources associated with the operation.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Calling with `psDevice` pointing to a device in the state TZ_STATE_INVALID.

Calling with `pksService` set to NULL.

Calling with `psSession` set to NULL.

Calling with `psOperation` set to NULL.

Parameters

`psDevice`: A pointer to the device.

`pksService`: A pointer to the service UUID.

The UUID value is typically hard coded in the client, but can be loaded dynamically by querying the installed services using the Service Manger.

`pksLogin`: A pointer to the login control structure, or NULL.

The `psLogin` parameter provides a pointer to the structure detailing the login information which must be supplied to the service; refer to section 5.3.4 for more details. The response of the service to the login information is defined by the client-service protocol.

This parameter may be NULL, which implies the TZ_LOGIN_PUBLIC login type.

`pksTimeLimit`: A pointer to the time limit, or NULL.

The `pksTimeLimit` parameter defines the absolute time by which the operation must be complete, after which the implementation should attempt to cancel it; refer to section 5.4.1 for details.

This parameter may be `NULL` which implies no timeout it used.

`psSession`: A pointer to the session.

`psOperation`: A pointer to the operation.

Returned Value

TZ_SUCCESS: The operation has been prepared successfully.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.5 TZOperationPrepareInvoke

```
tz_return_t TZOperationPrepareInvoke(
    INOUT   tz_session_t *          psSession,
            uint32_t                uiCommand,
    IN      tz_timelimit_t const *  pksTimeLimit,
    OUT     tz_operation_t *        psOperation)
```

Description

This function is responsible for locally preparing an operation that can be used to issue a command to a service with which the client has already created a session.

This function accepts an operation assumed to be in the state TZ_STATE_UNDEFINED.

When this function returns TZ_SUCCESS the operation is set to the state TZ_STATE_ENCODE; this state allows the client to encode a message to be exchanged with the service using the encoder functions of the TZAPI. Once the message has been encoded the client must call the function TZOperationPerform, or the asynchronous equivalent, to issue the command to the service.

When this function fails it can return any error code. In these conditions the state of the session must be unchanged, including the operation count. The state the operation must be set to the state TZ_STATE_INVALID.

The `pksTimeLimit` parameter defines the absolute time by which the operation must be complete, after which the implementation should attempt to cancel it. This parameter may be NULL which implies no timeout it used.

Note that if the perform function fails the client must still call TZOperationRelease to release resources associated with the operation.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session in a state other than TZ_STATE_OPEN.

Calling with `psOperation` set to NULL.

Parameters

`psSession`: A pointer to the open session.

`uiCommand`: The identifier of the command to execute, defined by the client-service protocol.

`pksTimeLimit`: A pointer to the time limit, or NULL.

The `pksTimeLimit` parameter defines the absolute time by which the operation must be complete, after which the implementation will attempt to cancel it; refer to section 5.4.1 for details.

This parameter may be NULL which implies no timeout it used.

`psOperation`: A pointer to the operation.

Returned Value

TZ_SUCCESS: The operation has been prepared successfully.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.6 TZOperationPrepareClose

```
tz_return_t TZOperationPrepareClose(
    INOUT   tz_session_t *     psSession,
    OUT     tz_operation_t *   psOperation)
```

Description

This function is responsible for locally preparing an operation that can be used to close a session between the client and a service.

This function accepts an operation assumed to be in the state TZ_STATE_UNDEFINED.

When this function returns TZ_SUCCESS the state of the session is changed to TZ_STATE_CLOSING. In this state any operation still running or shared memory block still allocated can still be used, but it is not possible to create new shared memory blocks or prepare new operations within the session. If this operation is never issued, TZOperationRelease must transition the session state back to TZ_STATE_OPEN.

Note that performing a close operation while other operations exist with a session, or while shared memory blocks are still allocated within it, results in *UNDEFINED* behavior.

When this function returns TZ_SUCCESS the operation is set to the state TZ_STATE_PERFORMABLE; this state allows the client to perform the close operation, but not to encode a message to be exchanged with the service. The client must call the function TZOperationPerform, or the asynchronous equivalent, to issue the operation to the security environment.

When this function fails it can return any error code. In these conditions the state of the session must be unchanged, including the operation count. The state the operation must be set to TZ_STATE_INVALID.

Note that the perform operation for a session closure cannot be canceled or timed-out by the client. When TZOperationPerform completes, whether with success or failure, the session is considered closed.

On failure of the perform function the client must still call TZOperationRelease to release resources associated with the operation.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session in a state other than TZ_STATE_OPEN.

Calling with `psOperation` set to NULL.

Parameters

`psSession`: A pointer to session.

`psOperation`: A pointer to the operation.

Returned Value

TZ_SUCCESS: The operation has been prepared successfully.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.7 TZOperationPerform

```
tz_return_t TZOperationPerform(
    INOUT   tz_operation_t *  psOperation,
    OUT     tz_return_t *     puiServiceReturn)
```

Description

This function performs a previously prepared operation – issuing it to the secure environment.

There are three kinds of operations that can be issued: opening a client session, invoking a service command, and closing a client session. Each type of operation is prepared with its respective function, which returns the operation structure to be used:

- TZOperationPrepareOpen prepares an open session operation.
- TZOperationPrepareInvoke prepares an invoke service command operation.
- TZOperationPrepareClose prepares a close session operation.

When calling this function, the operation must be in the state TZ_STATE_ENCODE or TZ_STATE_PERFORMABLE. For operations that support a structured message it is not required that a message has actually been encoded by the client. Once this function has been called, the state transitions to TZ_STATE_RUNNING and it is no longer possible to use to the encoder functions on the operation.

If an error is detected by the system before the operation reaches the service, then an error code is returned by TZOperationPerform and the value TZ_ERROR_GENERIC is assigned to `*puiServiceReturn`. In this case the operation will be in the state TZ_STATE_INVALID when the function returns and the decoder functions cannot be used on the operation.

The most common causes for an error occurring before the command reaches the service are:

- The encoder ran out of space – error returned is TZ_ERROR_ENCODE_MEMORY.
- The service does not exist – error returned is TZ_ERROR_ACCESS_DENIED.
- The system rejects a new session due to bad login credentials – error return is TZ_ERROR_ACCESS_DENIED.
- The operation has timed out, or been cancelled; error return is TZ_ERROR_CANCEL.
- The secure environment is busy or low on resource and cannot handle the request.

For open and invoke operations, if the operation reaches the service, but the service returns an error, then TZOperationPerform returns TZ_ERROR_SERVICE. The error code from the service is assigned to `*puiServiceReturn`. Unlike the case where a system error occurs, the service can return a message: the operation transitions to the state TZ_STATE_DECODE and the decoder functions can be used.

For open and invoke operations, if the operation succeeds then TZOperationPerform returns TZ_SUCCESS and the value TZ_SUCCESS is also assigned to `*puiServiceReturn`. The operation transitions to the state TZ_STATE_DECODE and the client can use the decoder functions to retrieve the message, if present, from the service.

For close operations the service cannot return a structured message, and the operation will always transition to TZ_STATE_INVALID. The decoder functions cannot be used on a close operation. A close operation cannot be performed while the session has other operations open, or has allocated shared memory blocks – this results in *UNDEFINED* behavior.

Regardless of the success or failure of the function the client code must always call TZOperationRelease to release any resources used by the operation.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_ENCODE.

Calling with `psOperation` pointing to a close operation, where the session being closed still has other operations open, or has allocated shared memory blocks.

Calling with `puiServiceReturn` set to NULL.

Parameters

`psOperation`: A pointer to the operation.

`puiServiceReturn`: A pointer to the variable that will contain the service return code.

Returned Value

TZ_SUCCESS: The operation was executed successfully.

TZ_ERROR_ENCODE_MEMORY: The encoder is in an error condition – it ran out of memory space.

TZ_ERROR_ACCESS_DENIED: The service was not found or the client was not authorized to access it.

TZ_ERROR_SERVICE: The service itself threw an error, which can be found in `*puiServiceReturn`.

TZ_ERROR_CANCEL: The operation timed out, or was explicitly cancelled.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.8 TZOperationRelease

```
void TZOperationRelease(
    INOUT   tz_operation_t *   psOperation)
```

Description

This function releases an operation, freeing any associated resources.

The behavior of this function varies slightly depending on the state of the operation:

- TZ_STATE_ENCODE or TZ_STATE_PERFORMABLE: The operation has not been issued to the system, and is destroyed without being issued. In this case it may be required to unwind some of the state change made to related structures, for example if the operation is a session closure the session state must transition back to TZ_STATE_OPEN.
- TZ_STATE_DECODE: The operation has been issued to the system and a response has been returned. After destroying an operation in this state, any messages in its decoder are lost, including unread entries and arrays that have been decoded by reference. If the client needs to keep any data from the message it must copy it out of the decoder owned memory before calling this function.
- TZ_STATE_INVALID: This function does nothing.

After this function returns the operation must be considered to be in the state TZ_STATE_UNDEFINED.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation in a state other than TZ_STATE_ENCODE, TZ_STATE_PERFORMABLE, TZ_STATE_DECODE or TZ_STATE_INVALID.

Parameters

`psOperation`: A pointer to the operation to release.

Returned Value

None (void).

6.1.9 TZOperationCancel

```
void TZOperationCancel(
    INOUT   tz_operation_t *  psOperation)
```

Description

This function requests the cancellation of an operation in an asynchronous manner; it sends a cancellation request to the secure environment but does not wait for the cancellation to occur.

This function can be used to cancel an operation prepared by TZOperationPrepareOpen or TZOperationPrepareInvoke; a close operation cannot be canceled. If TZOperationCancel is called on a close operation, the function behavior is *UNDEFINED*.

The function TZOperationCancel can be called at any time that the operation is valid, i.e. when it is not in the state TZ_STATE_INVALID or TZ_STATE_UNDEFINED.

If the client calls this function on an operation that has been prepared, but not performed, then the perform function returns TZ_ERROR_CANCEL immediately, without sending the operation to the secure environment.

If the client calls TZOperationCancel after the operation has completed, but before it is released, then the call has no effect.

If the application calls TZOperationCancel while the operation is being performed, the behavior is implementation-dependent. There is no guarantee that the cancellation request is taken into account by either the implementation or the service. If the cancellation request is acted upon the perform function, TZOperationPerform or TZOperationAsyncGetResult, should return TZ_ERROR_CANCEL.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation in the state TZ_STATE_INVALID or the state TZ_STATE_UNDEFINED.

Parameters

`psOperation`: A pointer to the operation to cancel.

Returned Value

None (void).

6.1.10 TZSharedMemoryAllocate

```
tz_return_t TZSharedMemoryAllocate(
    INOUT   tz_session_t *        psSession,
    INOUT   tz_shared_memory_t *  psSharedMem)
```

Description

This function allocates a block of memory, defined by the structure pointed to by `psSharedMem`, which is shared between the client and the service it is connected to.

Once a block is allocated, the client and the service can exchange references to the allocated block in messages encoded using structured messages. Depending on the implementation of the secure environment, this may allow the service to directly access this block of memory without the need for a copy; this allows for high-bandwidth non-structured communications.

On entry to this function the session must be in the state TZ_STATE_OPEN.

On entry to this function the fields `uiFlags` and `uiLength` of the shared memory structure must have been filled in with the required values. Other fields of the shared memory structure have *UNDEFINED* state on entry to this function and are filled in by the time the function returns.

If this function returns TZ_SUCCESS, the value `psSharedMem->pBlock` will contain the address of the shared memory allocation and the shared memory structure will be in the state TZ_STATE_OPEN. The implementation must guarantee that the returned buffer allocation is aligned on an 8-byte address boundary.

If this function returns any other value, the state of the structure is TZ_STATE_INVALID and `psSharedMem->pBlock` will be NULL.

After successful allocation of a block the client may subsequently pass the shared memory structure to the function TZEncodeMemoryReference to create a reference to a portion of the block.

Blocks are flagged with the intended direction of data flow, as described by the `psSharedMem->uiFlags` parameter. If an attempt is later made to encode a memory reference with incompatible sharing attributes an encoder error is thrown when the operation is performed.

The structure must be passed to the function TZSharedMemoryRelease when the block is no longer needed.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` in a state other than TZ_STATE_OPEN.

Calling with `psSharedMem` set to NULL.

Calling with `psSharedMem->uiFlags` set to an invalid set of flags.

Calling with `psSharedMem->uiLength` set to 0.

Parameters

`psSession`: A pointer to the session.

`psSharedMem`: A pointer to the shared memory block structure.

Returned Value

TZ_SUCCESS: the memory was allocated successfully.

TZ_ERROR_MEMORY: there is not enough memory to meet the allocation request.

TZ_ERROR_*: an implementation-defined error code for any other error.

6.1.11 TZSharedMemoryRegister

```
tz_return_t TZSharedMemoryRegister(
    INOUT   tz_session_t *        psSession,
    INOUT   tz_shared_memory_t *  psSharedMem)
```

Description

This function registers a block of memory, defined by the structure pointed to by `psSharedMem`, which is shared between the client and the service it is connected to.

Once a block is allocated the client and the service can exchange references to the allocated block in messages encoded using structured messages. Depending on the implementation of the secure environment, this may allow the service to directly access this block of memory without the need for a copy; this allows for high-bandwidth non-structured communications.

On entry to this function the session must be in the state TZ_STATE_OPEN.

On entry to this function the fields `pBlock`, `uiFlags` and `uiLength` of the shared memory structure must have been filled in with the required values. Other fields of the shared memory structure have *UNDEFINED* state on entry to this function and are filled in by the time the function returns.

If this function returns TZ_SUCCESS, the shared memory structure will be in the state TZ_STATE_OPEN.

If this function returns any other value the state of the structure is TZ_STATE_INVALID.

After successful registration of a block the client may subsequently pass the shared memory structure to the function TZEncodeMemoryReference to create a reference to a portion of the block.

Blocks are flagged with the intended direction of data flow, as described by the `psSharedMem->uiFlags` parameter. If an attempt is later made to encode a memory reference with incompatible sharing attributes an encoder error is thrown when the operation is performed.

The structure must be passed to the function TZSharedMemoryRelease when the block is no longer needed.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` in a state other than TZ_STATE_OPEN.

Calling with `psSharedMem` set to NULL.

Calling with `psSharedMem->pBlock` set to NULL or a pointer to invalid memory.

Calling with `psSharedMem->uiFlags` set to an invalid set of flags.

Calling with `psSharedMem->uiLength` set to 0.

Parameters

`psSession`: A pointer to the session.

`psSharedMem`: A pointer to the shared memory block structure.

Returned Value

TZ_SUCCESS: The memory was registered successfully.

TZ_ERROR_MEMORY: There is not enough memory to meet the registration request.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.12 TZSharedMemoryRelease

```
void TZSharedMemoryRelease(
    INOUT   tz_shared_memory_t *  psSharedMem)
```

Description

This function marks a block of shared memory associated with a session as no longer shared.

If the input shared memory structure is in the state TZ_STATE_INVALID this function does nothing, otherwise it frees the memory block. The caller must not access the memory buffer after calling this function.

The shared memory structure returned by this function will be in the state TZ_STATE_UNDEFINED.

When this function is called, the shared memory block must not be referenced by any operation, otherwise *UNDEFINED* behavior will occur.

Note that shared memory blocks must always be freed by calling this function; all memory blocks must be freed before a session can closed.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSharedMem` set to NULL.

Calling with `psSharedMem` pointing to a shared memory block that is still referenced by an operation.

Parameters

`psSharedMem`: A pointer to the shared memory block to free.

Returned Value

None (void).

6.1.13 TZSystemGetPropertyList

```
tz_return_t TZSystemGetPropertyList(
    INOUT   tz_device_t *          psDevice,
    OUT     tz_property_name_t *   psList,
    OUT     uint32_t *             puiLength)
```

Description

Calling this function retrieves the list of all of the property names associated with the system identified by psDevice.

The client passes in a pointer to an allocated input buffer, psList, containing space for *puiLength property name entries; each name is a tz_property_name_t structure.

If there is enough space in this buffer for the complete list of names the system copies them into the low entries of the array, sets the value *puiLength to the actual number of entries used, and returns TZ_SUCCESS.

If the buffer is specified, i.e. it is not NULL, but is too short to contain all of the properties then no entries are copied and *puiLength is set to the required number of entries. In this case, the function returns TZ_ERROR_SHORT_BUFFER.

If the client wishes to query the number of properties it can call this function with psList set to NULL. In this case *puiLength is set to the required number of entries and the function returns TZ_SUCCESS.

If this function errors for any reason, for example there is not enough resource in the secure environment to process the command, then *puiLength is set to zero and this function returns an implementation-defined error code.

The system properties are defined in section 5.5.4.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with psDevice set to NULL.

Calling with psDevice pointing to a device not in the state TZ_STATE_OPEN.

Calling with puiLength set to NULL.

Parameters

psDevice: A pointer to an open device.

psList: A pointer to an allocated buffer of *puiLength name sized entries.

May be NULL if the function is only called to query the number of entries required in the list.

puiLength: A pointer to a variable containing the number of entries in the buffer.

Returned Value

TZ_SUCCESS: The property list was returned successfully.

TZ_ERROR_SHORT_BUFFER: The input buffer passed by the client is too short.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.1.14 TZSystemGetProperty

```
tz_return_t TZSystemGetProperty(
    INOUT   tz_device_t *    psDevice,
    INOUT   tz_property_t *  psProperty)
```

Description

Calling this function retrieves the value of a specific property associated with the system identified by `psDevice`.

The client passes in a pointer to a `tz_property_t` structure which has the following fields set on entry:

- `uiName` is set to the name of the property to retrieve.
- `uiNamespace` is set to the namespace of the property to retrieve.
- `pValue` is set to a pointer to a client allocated buffer, or NULL.
- `uiLength` is set to the length of the allocated buffer, in bytes.

The client must ensure that the input buffer is suitably aligned for the expected content of the property value; for example if the value is an array of `uint32_t` values this array must be 4 byte aligned. The worst case alignment is on an eight byte boundary, enabling direct sharing of any C data type in accordance with the ARM Application Binary Interface **[ARM IHI 0036A]**.

If there is enough space in the value buffer for the complete property then the system copies the data into the low addresses of the array, sets the value `psProperty->uiLength` to the actual number bytes used and returns TZ_SUCCESS.

If the buffer is specified, i.e. it is not NULL, but is too short to contain the entire property then no data is copied and `psProperty->uiLength` is set to the required number of bytes. In this case, the function returns TZ_ERROR_SHORT_BUFFER.

If the client wishes to query the length of a property value it can call this function with `psProperty->pValue` set to NULL. In this case `psProperty->uiLength` is set to the required number of bytes and the function returns TZ_SUCCESS.

If this function errors for any reason, for example there is not enough resource in the system to process the command, then `psProperty->uiLength` is set to zero and this function returns an implementation-defined error code.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Calling with `psDevice` pointing to a device in the state TZ_STATE_INVALID or TZ_STATE_UNDEFINED.

Calling with `psProperty` set to NULL.

Parameters

`psDevice`: A pointer to an open device.

`psProperty`: A pointer to the property structure to load.

Returned Value

TZ_SUCCESS: The property list was returned successfully.

TZ_ERROR_SHORT_BUFFER: The input buffer passed by the client is too short.

TZ_ERROR_ACCESS_DENIED: The requested property cannot be found, or access is denied.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.2 Encoder and decoder functions

The encoder and decoder functions of the API are used to encode and decode structured messages exchanged between the client and the service. The encoder functions can only be used on an operation in the state TZ_STATE_ENCODE, which occurs after the operation has been prepared, but before it has been performed. The decoder functions can only be used on an operation in the state TZ_STATE_DECODE, this occurs when an operation succeeds or returns with a service error.

6.2.1 TZEncodeUint32

```
void TZEncodeUint32(
    INOUT   tz_operation_t *  psOperation,
            uint32_t          uiData)
```

Description

Calling this function appends the value of the passed `uint32_t`, `uiData`, to the end of the encoded message.

This function can only be called when the operation is in the state TZ_STATE_ENCODE. This occurs after the operation has been prepared, but before it has been performed.

If an error occurs, for example if there is no more space in the encoder buffer, this function sets the error state of the encoder.

This function does nothing if the error state of the encoder is already set upon entry.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation in a state other than TZ_STATE_ENCODE.

Parameters

`psOperation`: A pointer to the operation for which we are encoding.

`uiData`: The value to encode in the buffer.

Output

This function may set the state of the encoder to TZ_ERROR_ENCODE_MEMORY if there is insufficient space to encode the `uint32_t` value.

Returned Value

None (void).

6.2.2 TZEncodeArray

```
void TZEncodeArray(
    INOUT   tz_operation_t *  psOperation,
    IN      void const *      pkArray,
            uint32_t          uiLength)
```

Description

Calling this function appends a binary array pointed to by pkArray and of length uiLength bytes to the end of the encoded message. The implementation must guarantee that when decoding the array in the service the base pointer is eight byte aligned to enable any basic C data structure to be exchanged using this method.

It is valid behavior to encode a zero length array, where pkArray is not NULL but uiLength is 0, and a NULL array, where pkArray is NULL and uiLength is zero, using this function.

This function can only be called when the operation is in the state TZ_STATE_ENCODE. This occurs after the operation has been prepared, but before it has been performed.

If an error occurs, for example if there is no more space in the encoder buffer, this function sets the error state of the encoder.

This function does nothing if the error state of the encoder is already set upon entry.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with psOperation set to NULL.

Calling with psOperation pointing to an operation in a state other than TZ_STATE_ENCODE.

Calling with pkArray set to NULL when uiLength is a length other than 0.

Output

This function may set the state of the encoder to TZ_ERROR_ENCODE_MEMORY if there is insufficient space to encode the array.

Parameters

psOperation: A pointer to the operation for which we are encoding.

pkArray: A pointer to the binary buffer to encode.

uiLength: The length of the binary buffer in bytes.

Returned Value

None (void).

6.2.3 TZEncodeArraySpace

```
void * TZEncodeArraySpace(
     INOUT   tz_operation_t *  psOperation,
             uint32_t          uiLength)
```

Description

Calling this function appends an empty binary array of length `uiLength` bytes to the end of the encoded message and returns the pointer to this array to the client. This allows an implementation with fewer copies, as the encoder buffer can be filled directly by the client without needing a copy from an intermediate buffer into the real encoder buffer.

The implementation must guarantee that the returned buffer allocation is aligned on an eight byte boundary, enabling direct sharing of any C data type in accordance with the ARM Application Binary Interface **[ARM IHI 0036A]**.

It is valid behavior to allocate space for a zero length array in the encoder stream. This will return a pointer that is not NULL, but this pointer must never be dereferenced by the client code or *UNDEFINED* behavior may result.

This function can only be called when the operation is in the state TZ_STATE_ENCODE. This occurs after the operation has been prepared, but before it has been performed. Once the operation transitions out of the state TZ_STATE_ENCODE, which occurs if the operation is performed or is released, then the client must no longer access this buffer or *UNDEFINED* behavior may result.

If an error occurs, for example if there is no more space in the encoder buffer, this function sets the error state of the encoder and returns NULL.

This function does nothing if the error state of the encoder is already set upon entry, and will return NULL.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation in a state other than TZ_STATE_ENCODE.

Output

This function may set the state of the encoder to TZ_ERROR_ENCODE_MEMORY if there is insufficient space to encode the array space.

Parameters

`psOperation`: A pointer to the operation for which we are encoding.

`uiLength`: The length of the desired binary buffer in bytes.

Returned Value

A pointer to the buffer, or NULL upon error.

6.2.4 TZEncodeMemoryReference

```
void TZEncodeMemoryReference(
    INOUT   tz_operation_t *      psOperation,
    INOUT   tz_shared_memory_t *  psSharedMem,
            uint32_t              uiOffset,
            uint32_t              uiLength,
            uint32_t              uiFlags)
```

Description

Calling this function appends a reference to a range of a previously created shared memory block.

Memory references are used to provide a synchronization token protocol which informs the service when it can read from or write to a portion of the shared memory block. A memory reference is associated with a specific operation and is valid only during the execution of that operation.

When the TZEncodeMemoryReference function completes successfully the shared memory block is said to be "referenced". This reference is destroyed when the operation perform function completes (with or without an error). If the operation is never performed for any reason, the reference is destroyed when the operation is released. A shared memory block cannot be released while is it still referenced in an operation. Once a memory reference has been created the client must not read from, or write to, the referenced range until the reference is destroyed.

Some implementations of the secure environment may not be able to implement genuine shared memory and/or may make use of device hardware outside of the core. In these cases the system may require data copies or cache maintenance operations to ensure visibility of the data in a coherent manner. For this reason the memory reference is marked with a number of flags which can be used to ensure the correct copies and cache maintenance operations occur. Primarily these indicate the memory operations that the service is allowed to perform. Exactly one of the following flags must be specified:

- TZ_MEM_SERVICE_RO: The service can only read from the memory block.
- TZ_MEM_SERVICE_WO: The service can only write to the memory block.
- TZ_MEM_SERVICE_RW: The service can both read from and write to the memory block.

These flags must be a sub-set of the service permissions specified when the block was created using TZSharedMemoryAllocate, otherwise the encoder error TZ_ERROR_ENCODE_FORMAT will be raised.

If an error occurs, for example if there no more space in the encoder buffer or the range lies outside of the memory block, this function sets the error state of the encoder. Additionally, there is a restriction whereby the client cannot have multiple concurrent references to any address in the memory block. Attempting to write a memory reference that overlaps an existing one will result in the encoder entering an error state. In any of these cases the shared memory block is not referenced by calling this function.

This function does nothing if the error state of the encoder is already set upon entry.

Efficiency

In many systems the secure environment may pass buffers of data to secure hardware, such as a cryptographic accelerator, which exists outside of the main processor's caches. In these cases cache lines must be cleaned and invalidated to the point of coherence with the memory view of the hardware accelerator. This can be made significantly more efficient if the memory alignment and of memory references suits the requirements of the hardware implementation. See section 4.4.3.1 for more details about how a client can make efficient use of shared memory.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation in a state other than TZ_STATE_ENCODE.

Calling with `psSharedMem` set to NULL.

Calling with `psSharedMem` pointing to a memory block in a state other than TZ_STATE_OPEN.

Parameters

`psOperation`: A pointer to the operation for which we are encoding.

`psSharedMem`: A pointer to the shared memory block structure.

`uiOffset`: The offset, in bytes, from the start of the shared memory block to the start of the memory range.

`uiLength`: The length, in bytes, of the memory range.

`uiFlags`: The access flags for this memory reference.

Output

This function may set the state of the encoder to TZ_ERROR_ENCODE_MEMORY if there is insufficient space to encode the memory reference.

This function may set the state of the encoder to TZ_ERROR_ENCODE_FORMAT if the memory reference structure is not correct.

Returned Value

None (void).

6.2.5 TZDecodeUint32

```
uint32_t TZDecodeUint32(
    INOUT   tz_operation_t *   psOperation)
```

Description

This function decodes a single item of type `uint32_t` from the current offset in the structured message returned by the service.

If on entry the decoder is in an error state this function does nothing and returns 0. The state of the decoder remains unchanged.

If the decoder error state is not set on entry, the system attempts to decode the data item at the current offset in the decoder and return the result. If an error occurs this function returns 0 and sets the error state of the decoder. Otherwise the data value is returned by the function and the decoder offset is incremented past the item that has been decoded.

The decoder may set its error state in the following situations:

- There are no more items in the decoder.
- The item in the decoder is not of the type requested.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_DECODE.

Parameters

`psOperation`: The operation from which we decoding the integer.

Output

This function will set the state of the decoder to TZ_ERROR_DECODE_NO_DATA if there is no further data to decode.

This function may optionally check the type of the data returned and may set the state of the decoder to TZ_ERROR_DECODE_TYPE if there is a type mismatch. The presence of type checking in the library is implementation-defined; to ensure safety, clients must check the value of returned data to ensure that it meets any critical criteria.

Returned Value

`uint32_t`: The value of the data item on success, 0 on any error.

6.2.6 TZDecodeArraySpace

```
void * TZDecodeArraySpace(
    INOUT   tz_operation_t *  psOperation,
    OUT     uint32_t *        puiLength)
```

Description

This function decodes a block of binary data from the current offset in the structured message returned by the service. The length of the block is returned in `*puiLength` and the base pointer is the function return value.

The implementation must guarantee that the returned buffer allocation is aligned on an eight byte boundary, enabling direct sharing of any C data type in accordance with the ARM Application Binary Interface **[ARM IHI 0036A]**.

It is expected that this function will return a pointer to the binary data in the encoder buffer. The client must make use of this memory before the operation is released. After the operation is released the client must not access the buffer again. If the data is needed locally after the operation has been released it must first be copied into a client allocated memory block.

If on entry the decoder is in an error state this function does nothing and returns NULL, with `puiLength` set to 0. The state of the decoder remains unchanged.

If the decoder error state is not set on entry, the system attempts to decode the data item at the current offset in the decoder and return the result. If an error occurs this function returns NULL as the base pointer, sets the length to 0, and sets the error state of the decoder. Otherwise the data value is returned by the function, and the decoder offset is incremented past the array item in the decoder.

Note that this function may decode a NULL array and a zero length array as valid types. In the former case the return value is NULL and the length is zero. In the second case the return value is non-NULL and the length is zero – the pointer must not be dereferenced by the client.

The decoder may return errors in the following situations:

- There are no more items in the decoder.
- The item in the decoder is not of the type requested.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_DECODE.

Calling with `puiLength` set to NULL.

Parameters

`psOperation`: The operation from which we are decoding the array.

`puiLength`: The pointer to the variable that will contain the array length on exit.

Output

This function will set the state of the decoder to TZ_ERROR_DECODE_NO_DATA if there is no further data to decode.

This function may optionally check the type of the data returned, and may set the state of the decoder to TZ_ERROR_DECODE_TYPE if there is a type mismatch. The presence of type checking in the library is implementation-defined; to ensure safety, clients must check the value of returned data to ensure that it meets any critical criteria.

Returned Value

`uint32_t`: The value of the data item on success, 0 on any error.

6.2.7 TZDecodeGetType

```
uint32_t TZDecodeGetType(
    INOUT   tz_operation_t *  psOperation)
```

Description

This function returns the type of the data at the current offset in the decoder stream.

This function does not affect the error state of the decoder.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_DECODE.

Parameters

`psOperation`: The operation from which we are retrieving the result.

Returned Value

TZ_TYPE_NONE: There is no more data.

TZ_TYPE_UINT32: The next item in the decode stream is a `uint32`.

TZ_TYPE_ARRAY: The next item in the decode string is an array.

6.2.8 TZDecodeGetError

```
tz_return_t TZDecodeGetError(
    INOUT   tz_operation_t *  psOperation)
```

Description

This function returns the error state of the decoder associated with the given operation.

This function does not affect the error state of the decoder.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_DECODE.

Parameters

`psOperation`: The operation from which we are retrieving the result.

Returned Value

TZ_SUCCESS: There have been no decoding errors.

TZ_ERROR_*: The first decoder error to occur.

6.3 Service manager functions

This section of the specification deals with the Service manager component of the TrustZone API. These functions are used for installing and removing new services and querying the already installed services dynamically at run time.

The functions TZManagerDownloadService, which can be used for dynamic download of new services at run-time, and TZManagerRemoveService, which can be used to remove services at run-time, are optional and may not be present in some implementations. These functions can be found in the optional functions section, sub-sections 7.2.1 and 7.2.2 respectively.

6.3.1 TZManagerOpen

```
tz_return_t TZManagerOpen(
    INOUT   tz_device_t *        psDevice,
    IN      tz_login_t const *   pksLogin,
    OUT     tz_session_t *       psSession)
```

Description

This function creates a session with the Service manager in the secure environment. When the client no longer requires access to the Service manager it must call TZManagerClose to release the session.

This function accepts a session and an operation structure assumed to be in the state TZ_STATE_UNDEFINED.

The parameter `psLogin` points to a structure that defines login information, or may be NULL if the login type TZ_LOGIN_PUBLIC is to be used. The login information and logic is identical to that used for TZOperationPrepareOpen.

The ability for a client to query installed services and their properties using TZManagerGetServiceList, TZManagerGetServicePropertyList and TZManagerGetServiceProperty must be available to a client logged in using TZ_LOGIN_PUBLIC. Access to the service download and removal functionality provided by TZManagerDownloadService and TZManagerRemoveService may require stronger login credentials – these requirements are implementation-defined. If the login fails the session is not opened, its state is set to TZ_STATE_INVALID, and this function returns TZ_ERROR_ACCESS_DENIED.

If this function succeeds the session transitions to the state TZ_STATE_OPEN; if it fails it is set to the state TZ_STATE_INVALID.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psDevice` set to NULL.

Calling with `psDevice` pointing to a device not in the state TZ_STATE_OPEN.

Calling with `psSession` set to NULL.

Parameters

`psDevice`: A pointer to the device.

`pksLogin`: A pointer to the login credentials, or NULL if TZ_LOGIN_PUBLIC is to be used.

`psSession`: A pointer to the structure to contain the session information.

Returned Value

TZ_SUCCESS: If the session has been opened successfully.

TZ_ERROR_ACCESS_DENIED: If the provided login credentials are incorrect or are rejected.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.3.2 TZManagerClose

```
tz_return_t TZManagerClose(
    INOUT   tz_session_t *  psSession)
```

Description

Calling this function closes a Service manager session.

This function accepts a structure in the state TZ_STATE_OPEN or in the state TZ_STATE_INVALID.

If the input is an open session then the function cancels any operation pending on the Service manager, waits for their completion and then closes the session. When this function returns it is guaranteed that the Service manager session is closed, even if the function returns an error.

If the input is a session in the state TZ_STATE_INVALID this function does nothing and returns TZ_SUCCESS.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session with something other than the Service manager.

Parameters

`psSession`: The session to close.

Returned Value

TZ_SUCCESS: If the session had been closed.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.3.3 TZManagerGetServiceList

```
tz_return_t TZManagerGetServiceList(
    INOUT   tz_session_t *  psSession,
    OUT     tz_uuid_t *     psList,
    INOUT   uint32_t *      puiLength)
```

Description

Calling this function retrieves the list of all of the UUIDs of the services installed on the device.

The client passes in a pointer to an allocated input buffer psList containing space for *puiLength UUID entries.

If there is enough space in this buffer for the complete list of services the system copies all of the service UUIDs into the low entries of the array, sets the value *puiLength to the actual number of entries used, and returns TZ_SUCCESS.

If the buffer is specified, i.e. it is not NULL, but is too short to contain the complete list then no entries are copied and *puiLength is set to the required number of entries. In this case, the function returns TZ_ERROR_SHORT_BUFFER.

If the client wishes to query the number of installed services it can call this function with psList set to NULL. In this case *puiLength is set to the required number of entries and the function returns TZ_SUCCESS.

If this function errors for any reason, for example there is not enough resource in the secure environment to process the command, then *puiLength is set to zero and this function returns an implementation-defined error code.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with psSession set to NULL.

Calling with psSession pointing to a session not in the state TZ_STATE_OPEN.

Calling with psSession pointing to a session with something other than the Service manager.

Calling with puiLength set to NULL.

Parameters

psSession: A pointer to an open Service manager session.

psList: A pointer to an allocated buffer of *puiLength tz_uuid_t sized entries. May be NULL if the function is only called to obtain the number of entries required in the list.

puiLength: A pointer to a variable containing the number of entries in the buffer.

Returned Value

TZ_SUCCESS: If the operation has been prepared successfully.

TZ_ERROR_SHORT_BUFFER: The input buffer passed by the client is too short.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.3.4 TZManagerGetServicePropertyList

```
tz_return_t TZManagerGetServicePropertyList(
    INOUT   tz_session_t *          psSession,
    IN      tz_uuid_t const *       pksService,
    OUT     tz_property_name_t *    pasList,
    INOUT   uint32_t *              puiLength)
```

Description

Calling this function retrieves the entire list of the property name values associated with the service `pksService`.

The client passes in a pointer to an allocated input buffer `pasList` containing space for `*puiLength` names entries; each name is a `tz_property_name_t` structure.

If there is enough space in this buffer for the complete list of names the system copies them into the low entries of the array, sets the value `*puiLength` to the actual number of entries used, and returns TZ_SUCCESS.

If the buffer is specified, i.e. it is not NULL, but is too short to contain the entire list then no entries are copied and `*puiLength` is set to the required number of entries. In this case, the function returns TZ_ERROR_SHORT_BUFFER.

If the client wishes to query the number of properties it can call this function with `pasList` set to NULL. In this case `*puiLength` is set to the required number of entries and the function returns TZ_SUCCESS.

If this function errors for any reason, for example there is not enough resource in the secure environment to process the command, then `*puiLength` is set to zero and this function returns an implementation-defined error code.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session not in the state TZ_STATE_OPEN.

Calling with `psSession` pointing to a session with something other than the Service manager.

Calling with `pksService` set to NULL.

Calling with `puiLength` set to NULL.

Parameters

`psSession`: A pointer to an open Service manager session.

`pksService`: A pointer to the service UUID.

`pasList`: A pointer to an allocated buffer of `*puiLength` name sized entries.

May be NULL if the function is only to return the number of entries required in the list.

`puiLength`: A pointer to a variable containing the number of entries in the buffer.

Returned Value

TZ_SUCCESS: The property list was returned successfully.

TZ_ERROR_SHORT_BUFFER: The input buffer passed by the client is too short.

TZ_ERROR_ACCESS_DENIED: The service does not exist or does not allow access to this client.

TZ_ERROR_*: An implementation-defined error code for any other error.

6.3.5 TZManagerGetServiceProperty

```
tz_return_t TZManagerGetServiceProperty(
    INOUT   tz_session_t *     psSession,
    IN      tz_uuid_t const *  pksService,
    INOUT   tz_property_t *    psProperty)
```

Description

Calling this function retrieves the value of a specific property associated with the service identified by `*pksService`.

The client passes in a pointer to a `tz_property_t` structure which has the following fields set on entry:

- `uiName` is set to the name of the property to retrieve.
- `uiNamespace` is set to the namespace of the property to retrieve.
- `pValue` is set to a pointer to a client allocated buffer, or NULL.
- `uiLength` is set to the length of the allocated buffer, in bytes.

The client must ensure that the input buffer is suitably aligned for the expected content of the property value – for example if the value is an array of `uint32_t` values this array must be four byte aligned. The worst case alignment is a buffer aligned on an eight byte boundary, enabling direct sharing of any C data type in accordance with the ARM Application Binary Interface **[ARM IHI 0036A]**.

If there is enough space in the value buffer for the complete property then the system copies the data into the low addresses of the array, sets the value `psProperty->uiLength` to the actual number bytes used, and returns TZ_SUCCESS.

If the buffer is specified, i.e. it is not NULL, but is too short for the entire property value then no data is copied and `psProperty->uiLength` is set to the required number of bytes. In this case, the function returns TZ_ERROR_SHORT_BUFFER.

If the client wishes to query the length of a property value it can call this function with `psProperty->pValue` set to NULL. In this case `psProperty->uiLength` is set to the required number of bytes and the function returns TZ_SUCCESS.

If this function errors for any reason, for example there is not enough resource in the system to process the command, then `psProperty->uiLength` is set to zero and this function returns an implementation-defined error code.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session in the state TZ_STATE_OPEN.

Calling with `psSession` pointing to a session with something other than the Service manager.

Calling with `pksService` set to NULL.

Calling with `psProperty` set to NULL.

Parameters

`psSession`: A pointer to an open Service manager session.

`pksService`: A pointer to the service to query.

`psProperty`: A pointer to the property structure to load.

Returned Value

TZ_SUCCESS: The property list was returned successfully.

TZ_ERROR_SHORT_BUFFER: The input buffer passed by the client is too short.

TZ_ERROR_ACCESS_DENIED: The service or the property name does not exist, or the service does not allow access to this client.

TZ_ERROR_*: An implementation-defined error code for any other error.

7 OPTIONAL FUNCTIONS

This section of the API details optional functions which may only be implemented on some systems. In particular this section contains functions that deal with asynchronous execution of operations and the download and removal of services at run-time.

7.1 Asynchronous operations

Some operating systems, such as SymbianOS, use an event-driven execution model – in such cases, most of the interfaces with the system are asynchronous. The client application requests some task to be performed and then continues executing; in these designs the client thread only has to block when it reaches the point that it cannot continue executing without the response. To enable efficient use of TZAPI by clients in these systems the following functions can be used in place of the TZOperationPerform.

In operating systems such as Linux and Windows Mobile there is not an established precedent for asynchronous system call execution. Most calls to the system are blocking and a client thread must wait for the response to that command invocation before continuing. In these system designs it is expected that the blocking implementation of the TZAPI will be used and that the asynchronous functions need not be implemented.

7.1.1 TZOperationAsyncStart

```
void TZOperationAsyncStart(
    INOUT   tz_operation_t *  psOperation,
    INOUT   void *            pSyncObj)
```

Description

This function performs part of the TZOperationPerform task, but in an asynchronous way. This function initiates the operation, sending it to the service, and updates the operation state to TZ_STATE_RUNNING_ASYNC. It then returns immediately without waiting for the response from the service.

When the asynchronous operation completes, the implementation specific synchronization object `*pSyncObj` is signaled. At this point the operation result can be retrieved by calling TZOperationAsyncGetResult.

Note that the implementation must ensure that the synchronization object remains valid until TZOperationAsyncGetResult is called and returns a value other than TZ_PENDING.

If the asynchronous operation cannot be performed for any reason, the synchronization object will still be signaled and a subsequent call to TZOperationAsyncGetResult will return the appropriate error code.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_ENCODE.

Calling with `pSyncObj` set to NULL.

Parameters

`psOperation`: The operation to execute.

`pSyncObj`: The implementation-defined synchronization object.

Returned Value

None (void).

7.1.2 TZOperationAsyncGetResult

```
tz_return_t TZOperationAsyncGetResult(
    INOUT   tz_operation_t *  psOperation,
    OUT     uint32_t *        puiServiceReturn)
```

Description

This function performs the second part of an asynchronous operation, fetching the result.

The operation result is available for retrieval when the synchronization object specified in the asynchronous start function is signaled. This function also allows a polled mode of implementation – returning TZ_PENDING until complete or cancelled. Each call to TZOperationPerform that returns TZ_PENDING will assign the value TZ_ERROR_GENERIC to `*puiServiceReturn`.

If the error return is TZ_PENDING the operation must remain unchanged by the implementation. For any other error the behavior is identical to TZOperationPerform, which is repeated in summary below:

For open and invoke operations, if the error return is not TZ_PENDING or TZ_ERROR_SERVICE, then the operation transitions to the state TZ_STATE_INVALID and the decoder functions cannot be used to retrieve an answer message.

For open and invoke operations, if the return is TZ_SUCCESS or TZ_ERROR_SERVICE then the operation transitions to the state TZ_STATE_DECODE and the response from the service can be read using the decoder functions.

For close operations the state always transitions to TZ_STATE_INVALID and the decoder functions cannot be used.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psOperation` set to NULL.

Calling with `psOperation` pointing to an operation not in the state TZ_STATE_RUNNING_ASYNC.

Calling with `puiServiceReturn` set to NULL.

Parameters

`psOperation`: The operation whos status to query.

`puiServiceReturn`: The pointer to the variable to contain the service error code.

Returned Value

TZ_SUCCESS: The operation has completed successfully.

TZ_PENDING: The operation is still executing.

TZ_ERROR_ENCODE: The operation encoder was in error when the operation was performed.

TZ_ERROR_ACCESS_DENIED: The service was not found or the client was not authorized to access it.

TZ_ERROR_SERVICE: The service itself threw an error, which can be found in `*puiServiceReturn`.

TZ_ERROR_CANCEL: The operation timed out, or has been explicitly cancelled.

TZ_ERROR_*: An implementation-defined error code for any other error.

7.2 Run-time download and removal of services

Feature rich systems may allow run-time download, upgrade or removal of security services, often in the form of byte-code services running on a virtual machine. However, many platforms are fixed function and are not extensible in the field. The TZManager service download and removal functions are optional to cater for both of these use-cases.

7.2.1 TZManagerDownloadService

```
tz_return_t TZManagerDownloadService(
    INOUT   tz_session_t *   psSession,
    IN      uint8_t const *  kauiData,
            uint32_t         uiLength,
    OUT     tz_uuid_t *      psService)
```

Description

This function downloads a new service to the secure environment.

The service code to download is located in the buffer `kauiData` and is `uiLength` bytes long.

The implementation may refuse the client access to this function, based on the login credentials and the implementation of the secure environment. In cases where this occurs this function will return TZ_ERROR_ACCESS_DENIED.

The format of the download service data structure is implementation-dependent, but will typically includes protection mechanisms such as a cryptographic signature. The implementation may refuse to download a service if it is incorrectly formatted – in such cases this function returns TZ_ERROR_ACCESS_DENIED.

If a service with the same UUID is already installed on the system the behavior is implementation-defined. In some cases the existing service may be replaced by the new version, allowing an upgrade to be deployed, although many designs will prohibit such a procedure. If the download is prevented then this function returns TZ_ERROR_ACCESS_DENIED.

If this function succeeds, the newly downloaded service's UUID is returned in `*psService`. This identifier may be used by the client to connect to the service.

In the cases where this function returns an error, the contents of `*psService` are *UNDEFINED*.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session in the state TZ_STATE_INVALID.

Calling with `psSession` pointing to a session with something other than the Service manager.

Calling with `kauiData` set to NULL.

Calling with `psService` set to NULL.

Parameters

`psSession`: A pointer to the Service manager session.

`kauiData`: A pointer to the Service download data.

`uiLength`: The length in bytes of the data buffer.

`psService`: A pointer to a UUID structure to contain the downloaded service's UUID on success.

Returned Value

TZ_SUCCESS: The service download was successful.

TZ_ERROR_ACCESS_DENIED: The service download was rejected.

TZ_ERROR_*: An implementation-defined error code for any other error.

7.2.2 TZManagerRemoveService

```
tz_return_t TZManagerRemoveService(
    INOUT   tz_session_t *     psSession,
    IN      tz_uuid_t const *  pksService)
```

Description

This function attempts to remove a service, identified by `pksService`, from the secure environment.

The implementation may refuse the client access to this function, based on the login credentials and the implementation of the secure environment. In cases where this occurs this function will return TZ_ERROR_ACCESS_DENIED.

In some cases the specified service may not be removable, or the service may not exist. In these cases this function returns TZ_ERROR_ACCESS_DENIED.

Undefined Behavior

The following situations result in *UNDEFINED* behavior:

Calling with `psSession` set to NULL.

Calling with `psSession` pointing to a session in the state TZ_STATE_INVALID.

Calling with `psSession` pointing to a session with something other than the Service manager.

Calling with `pksService` set to NULL.

Parameters

`psSession`: A pointer to the Service manager session.

`pksService`: A pointer to a UUID structure containing the ID of the service to remove.

Returned Value

TZ_SUCCESS: The service download was successful.

TZ_ERROR_ACCESS_DENIED: The service removal was rejected.

TZ_ERROR_*: An implementation-defined error code for any other error.

8 APPENDIX: OVERVIEW TRUSTZONE API 3.0 CHANGES

In the time since TZAPI 2.0 was published ARM has received feedback from developers implementing the library beneath the API and from those writing client applications built on top of it. This API revision acts in accordance with much of this feedback to:

- reduce the complexity of the API,
- achieve zero-copy behavior in more circumstances,
- and make existing features such as properties more general.

To make it possible to achieve the desired improvements, this version of the specification is not an incremental change on TZAPI 2.0, but is a significant redesign of the interface. It must however be noted that the underlying principles of the API, a communications model based on connection-oriented sessions, has not been changed. This section outlines some of the changes made in more detail.

8.1 Source-level compatibility only

TZAPI 3.0 does not attempt to provide binary compatibility between different implementations, this enables significant compile-time optimization and selection of implementation features.

All client applications using TZAPI 3.0 must be recompiled against implementation-specific TZAPI headers before they can be used on top of that implementation.

8.2 Simplifying functions

TZAPI 3.0 has taken two approaches to reduce the number of parameters required by some functions to reduce code size and improve performance:

- Introduced a well-defined state machine for each of the critical objects in the API enabling the smaller independent objects in TZAPI 2.0, such as encoders and decoders, to be turned in to states of the larger objects.
- Encapsulated similar data items, such as login credentials and properties, in structures which can be passed by reference reducing the number of copies passed around the system.

8.3 Session open operation login parameters

The login mechanism has been redesigned to allow a service to access the independent login credentials that identify the calling client; this enables a service to make more intelligent decisions about who or what it grants access to. The login credentials include information such as the identity of the user executing the client application, and the file-system path of the client application.

8.4 Simplifying the structured message encoders and decoders

The biggest area of functional change is the simplification of the structured message encoder and decoder functions. TZAPI 3.0 reduces the number of types which can be encoded from thirteen to only three fundamental data types:

- a single 32-bit unsigned integer,
- a binary data array,
- and a shared memory reference.

The new array functions are constructed to allow zero-copy behavior, reducing the impact of unnecessary copies of arrays on the system's memory performance.

A new function has been added to allow a client to dynamically query the type of data at the current offset in a decode stream, enabling flexible dynamic protocol construction.

8.5 Session close operation data payload

Session close operations can no longer carry a structured message payload.

TZAPI 2.0 implementations had to handle a corner case where the service close operation needed to be generated by the system after the client had crashed or the system was low on resources. The secure service therefore had to handle two close conditions: one case where the close operation was constructed by the client and contained a payload, and one where the close operation was sent by the system and contained no payload. To enable services to implement a unified close mechanism TZAPI 3.0 only provides the common case; no payload for close operations is allowed.

8.6 System and service properties

The property architecture used by the system and by services has been redesigned to enable a smaller footprint and more diverse usage. The property name is now specified as a pair of unsigned 32-bit integers, representing a namespace and a local name, and the property value is stored as a binary buffer. This replaces the wide-character strings which were used for both the name and the value in TZAPI 2.0.