GlobalPlatform Device Technology

TEE TA Debug Specification

Version 1.0.1

Public Release

June 2016

Document Reference: GPD_SPE_025

Contents

1 Introduction 6
1.1 Audience 6
1.2 IPR Disclaimer 6
1.3 References 6
1.3.1 Normative References 6
1.3.2 Informative References 7
1.4 Terminology and Definitions 7
1.4.1 Key Words 7
1.4.2 Parameter Annotations 7
1.4.3 Other Terminology 7
1.5 Abbreviations and Notations 9
1.6 Revision History 9

2 General Information 10
2.1 Security and Certification Considerations 10
2.2 General Debug Architecture 11
2.2.1 Pseudo-TAs Implementing the Debug APIs 11
2.2.2 Monitoring Applications 12
2.3 Mandatory and Optional Parts of this Specification 14
2.3.1 Status of Examples Found in this Specification 14
2.4 Data Types 15
2.4.1 Basic Types 15
2.4.2 Additional Types 15

3 TA Post Mortem Reporting (PMR) Protocol 16
3.1 PMR Service pTA Availability 18
3.1.1 Releasing the Panicked TA State Data inside the Trusted OS 18
3.2 PMR Service Command Protocol 19
3.2.1 Header File 21
3.2.2 Constants 21
3.2.2.1 Return Codes 21
3.2.2.2 PMR Service UUID 22
3.2.3 PMR Session Commands 22
3.2.3.1 PMR Command Values 22
3.2.3.2 `CMD_PMR_INIT_SESSION` 23
3.2.3.3 `CMD_PMR_WAIT` 24
3.2.3.4 `CMD_PMR_FETCHPMR` 26
3.2.3.5 `CMD_PMR_CLOSE_SESSION` 27
3.2.4 Structures 28
3.2.4.1 State, Stack, and Heap 28
3.2.4.2 Client Side `PMR_State` 29
3.2.4.3 Client Side `PMR_MessageBuffer` 31
3.2.5 PMR Code Example: Using the TEE PMR Client Protocol 34

4 TEE Debug General Extensions 35
4.0 Header File 35
4.1 `TEE_DebugSetMarker` 35
4.2 `gpd.ta.session.ID` Property 37
4.3 Specification Number 38
4.4 Function Numbers 39

5 Debug Log Message (DLM) API and Protocol 40
5.1 Overview 40
5.2 DLM Service pTA Availability 42
5.3 Client Side DLM Protocol 43
5.3.1 Header Files 44
5.3.1.1 Client Interface Header 44
5.3.1.2 TA Interface Header 44
5.3.2 Constants 45
5.3.2.1 Return Codes 45
5.3.2.2 DLM Service UUID 45
5.3.2.3 DLM Command Values 46
5.3.3 DLM Session Commands 47
5.3.3.1 `CMD_DLM_INIT_SESSION` 47
5.3.3.2 `CMD_DLM_FLUSH` 48
5.3.3.3 `CMD_DLM_HALT` 48
5.3.3.4 `CMD_DLM_RUN` 49
5.3.3.5 `CMD_DLM_BLOCKINGMODE` 49
5.3.3.6 `CMD_DLM_WAITANDFETCHBUFFER` 50
5.3.4 Client Side `DLM_State` Structure 51
5.3.5 DLM Buffer Structure and Handling 52
5.3.5.1 DLM Message Filter Type 52
5.3.5.2 DLM Message Buffer Structure 52
5.3.5.3 Circular Buffer Usage 53
5.3.6 DLM Code Example: Using the TEE DLM Client Protocol 53
5.4 Internal DLM API 54
5.4.1 `TEE_DLMServiceOpen` 54
5.4.2 `TEE_DLMServiceClose` 55
5.4.3 `TEE_DLMServiceStatus` 55
5.4.4 TEE Internal API Extension – Debug Log Functions 57
5.4.4.1 Property Indicating DLM Installed on the System 57
5.4.4.2 `TEE_DLMPrintf` 57
5.4.5 DLM Code Example: Using the TEE DLM API in a TA 62

6 Debug Authorization 63
6.1 Authorization Principles 63
6.1.1 Debug Authorization Methodology 63
6.2 Authorization Debug Rules Properties 64
6.2.1 Debug Rules Properties 64
6.2.2 Available PMR Data – `DATA_AVAILABLE` 65
6.2.2.1 Required Support of `DATA_AVAILABLE` PMR States 66
6.2.3 Available DLM Data – `DATA_AVAILABLE` 67
6.2.4 Required Support of `DATA_AVAILABLE` DLM States 67
6.2.5 Properties of Underlying Data Structure Format and Security Strength 67
6.3 Debug Unlock Property 68

7 TEE Management Framework Control of Debug Rules Properties 69
7.1 Changing TEE Debug Rules Properties 69
7.2 Changing TA Debug Rules Properties 69

Annex A Panicked Function Identification 70

Annex B DLM Client Example 71

Annex C PMR Client Example 75

Annex D DLM TA Usage Example 81

Figures

Figure 2-1: General Case PMR and DLM Architecture Diagram 12
Figure 2-2: Additional DLM Architecture Diagram Including Optional Proprietary Link 13
Figure 2-3: Minimal Implementation 14
Figure 3-1: PMR Monitoring Introduction....................16
Figure 3-2: Monitoring Application View of PMR with Session ID Available 19
Figure 3-3: Monitoring from the Point of View of the PMR Service pTA 20
Figure 5-1: DLM Debugging with REE Client 41
Figure 5-2: DLM Debugging with Off-device Monitoring Client 41
Figure 5-3: Simple Flow Diagram of PMR Monitoring 43

Tables

Table 1-1: Normative References....................6
Table 1-2: Informative References 7
Table 1-3: Terminology and Definitions....................7
Table 1-4: Abbreviations and Notations 9
Table 1-5: Revision History 9
Table 2-1: Mandatory and Optional Parts of the Specification....................14
Table 2-2: Basic Types....................15
Table 2-3: Additional Types....................15
Table 3-1: API Return Codes – PMR Interface 21
Table 3-2: PMR Command Values....................22
Table 4-1: Function Numbering 39
Table 5-1: API Return Codes – DLM and Related APIs....................45
Table 5-2: DLM Command Values 46
Table 5-3: TEE DLM Availability Property 57
Table 5-4: Valid Conversion Specifiers 59
Table 6-1: Debug Rules Properties 64
Table 6-2: DATA_AVAILABLE Valid PMR States 65
Table 6-3: `DATA_AVAILABLE` Valid DLM States....................67
Table 6-4: TEE Debug Unlock Property 68
Table A-1: Function Identification Values 70

1 Introduction

This document specifies the GlobalPlatform Trusted Execution Environment (TEE) Debug interfaces and protocols. This version specifies:

- The syntax and semantics of the TA Post Mortem Reporting (PMR) client interface, i.e. local commands, and data structures transported through the TEE Client API to report the Panic state of a TA to a Client Application.
- The syntax and semantics of the Debug Log Message (DLM) client interface, i.e. local commands, and data structures transported through the TEE Client API to transfer debug `printf()`-type messages to a display or logging tool.
- The syntax and semantics of the DLM internal interface, i.e. local commands, and data structures available to a TEE TA to make use of the DLM service.

1.1 Audience

This document is intended primarily for the use of TEE OS Component implementers, Debug Tools developers, and Trusted Application developers.

1.2 IPR Disclaimer

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

1.3 References

1.3.1 Normative References

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPD_SPE_007	GlobalPlatform Device Technology TEE Client API Specification	[TEE Client API]
GPD_SPE_009	GlobalPlatform Device Technology TEE System Architecture	[TEE Sys Arch]
GPD_SPE_010	GlobalPlatform Device Technology TEE Internal API Specification	[TEE Internal API]
GPD_SPE_120	TEE Management Framework v1.0 [to be published; prior to publication the proprietary options for related requirements should be used]	[TEE Mgmt Fwk]

Standard / Specification	Description	Ref
C99	ISO/IEC 9899:TC3 – Programming languages – C – Technical Corrigenda 3 http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf	[C99]
RFC 2119	Key words for use in RFCs to Indicate Requirement Levels http://www.ietf.org/rfc/rfc2119.txt	[RFC 2119]
RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace http://www.ietf.org/rfc/rfc4122.txt	[RFC 4122]

1.3.2 Informative References

Table 1-2: Informative References

Standard / Specification	Description	Ref
RFC 3629	UTF-8, a transformation format of ISO 10646 http://www.ietf.org/rfc/rfc3629	[RFC 3629]

1.4 Terminology and Definitions

1.4.1 Key Words

The key words “MUST”, “MUST NOT”, “SHALL”, “SHALL NOT”, “REQUIRED”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document indicate normative statements and are to be interpreted as described in [RFC 2119].

1.4.2 Parameter Annotations

As this specification forms an extension of the TEE Internal API Spec, it inherits the same terminology regarding parameters. Please see [TEE Internal API] section 3.4 for a discussion of `[in]`, `[out]`, `[inout]`, `[instring]`, etc.

1.4.3 Other Terminology

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

Table 1-3: Terminology and Definitions

Term	Definition
Debug Rules	A set of rules attached to the TEE and TAs that define (in combination) what debug is allowed by the client requesting it. How these rules are attached to the TEE or TAs, and what mechanisms may change these are out of scope for this document.

Term	Definition
Debug Rules Properties	A set of properties associated with either the TEE or TA. When these are combined they control the ability to perform certain debug operations on the TAs resident in the TEE. These properties may be read by a TA and may be changed by TA management or proprietary mechanisms (see Chapter 7).
Debug Service	A service providing an interface for a client to receive DLM or PMR output from a TEE; may be a pTA (see below) or a proprietary solution.
Debug Log Message (DLM)	A text message stream providing status information from the TA.
Execution Environment (EE)	A set of hardware and software components providing facilities necessary to support running of applications. Typically consists of the following elements: • A hardware processing unit • A set of connections between the processing unit and other hardware resources • Physical volatile memory • Physical non-volatile memory
Function Number	A number identifying a function in the context of a specification number. See sections 4.3 and 4.4.
Modifiable Debug Rules Properties	A special sub class of properties that may only be modified when `gpd.tee.debug.debug_unlock_properties` is `true`. (See section 6.3).
Monitored TA	The TA that is being debugged or watched for Panics.
Monitoring Application	The client of the Debug service pseudo-TA. This may be resident in the REE, or in the TEE, or (if using proprietary mechanisms) in a separate device such as debugging PC. Note that the Monitoring Application may be the first part of a chain of applications such that the actual tool that displays information to the developer may be a debugger that links to the Monitoring Application over many possible mechanisms.
Post Mortem Reporting (PMR)	A summary of the TA state when a Panic occurred.
Pseudo-Trusted Application (Pseudo-TA or pTA)	A TA that is tightly integrated into the Trusted Framework. It may be connected to using the client API but may also present [TEE Internal API] extensions, or act on other information available in the Trusted Framework but not available through the TEE Internal API. It may not have the usual TA management controls (e.g. if fully integrated to the Trusted Framework, it may be impossible to apply some management states or to upgrade separately). See section 2.2.1.
Session Client Application	The Client Application that is communicating with the Monitored TA.
Specification Number	A number indicating in which GlobalPlatform specification a function was defined. See section 4.3.

Term	Definition
TA Download Package	When a TA is in its pre-installation state it is typically bundled with data relevant to installation procedures and this is cryptographically protected. While this bundling is proprietary, some of the state of the associated data is described in this document.

1.5 Abbreviations and Notations

This section is supplementary to the Abbreviations and Notations found in [TEE Internal API].

Table 1-4: Abbreviations and Notations

Abbreviation / Notation	Meaning
DLM	Debug Log Message
EE	Execution Environment
PMR	Post Mortem Reporting
pTA	Pseudo-Trusted Application

1.6 Revision History

Table 1-5: Revision History

Date	Version	Description
February 2014	1.0	Initial release
June 2016	1.0.1	Added `#define GPD_TEE_TA_DEBUG_1_0_1` in line with new policy (sections 3.2.1, 5.3.1.1, 5.3.1.2) Added detail to `#define` statements (section 3.2.4.3) Specified header file for general extensions (section 4.0). Clarified `printf` parameter data types (Table 5-4) DLM buffer shall include time in milliseconds (section 5.4.4.2) Corrected example of access to stack (Annex C)

2 General Information

This document describes services that are designed to support TA development and/or compliance testing of the GlobalPlatform defined internal APIs.

The Post Mortem Reporting (PMR) service supports compliance testing and TA debug. This service provides a method for a TEE to report to clients the termination status of TAs which enter the Panic state. Without this capability it is not possible to certify correct functionality of the internal APIs, as the Panic state is used to report various error conditions that need to be tested. All compliant TEEs shall implement the PMR interface.

The Debug Log Message (DLM) service is useful in a TA debug scenario. This service provides a method for a TA to report simple debug information on authorized systems. It may report to Client Applications or off-device hardware or both. (The reporting target is a device design choice and not specified in this document.) Compliant implementations should implement the DLM interface but it is not mandatory to do so.

Neither of these services (DLM or PMR) precludes the use of proprietary methods of debug or status logging that fit within the security requirements of the GlobalPlatform TEE.

Both services can identify Monitored TAs (and filter the returned results) based on Monitored TA UUID and (optionally) client session identification.

- How a Monitoring Application receives the Monitored TA UUID or client session identification is out of scope of this specification.

Both PMR and DLM interfaces require some form of authorization mechanism. This mechanism may be implemented by proprietary mechanisms or through GlobalPlatform TEE remote administration mechanisms.

Debug authorization is exposed through TAs and the TEE having associated PMR and/or DLM *Debug Rules Properties* stating what kind of debug is allowed on a particular TA or device.

- This document specifies the relationship between TEE Debug Rules Properties restrictions and TA Debug Rules Properties restrictions.
- This document specifies what information shall be present in those properties and how the debug service should interpret them to control debug output.

Proprietary debug handling may extend what is defined in this specification. As such this specification may be a subset of what the device manufacturer implements, if they choose to add a proprietary extension.

2.1 Security and Certification Considerations

Clearly the presence of debug interfaces is anathema to providing a secure environment, but conversely such interfaces are often required to prove correct functionality of some aspects of a device, as well as to enable development.

Three principles are defined to provide security assurance.

- The combination of access rules provided by TAs and TEEs shall permit only the least level of authorized debug.
- The absence of authorization shall be equivalent to an authorization stating that no debug is allowed.
- A certified device may have its debug capability disabled by removing the appropriate Debug Rules Properties. This level of change shall not change the certified status of a device.

See section 6.1, for the definitive set of security rules relating to debug output.

2.2 General Debug Architecture

The architecture presented here focuses around the concept of a pseudo-TA providing the connection for applications wishing to control and receive the debug output; however, in some cases outlined below, these pseudo-TAs may be replaced with proprietary output and control mechanisms.

2.2.1 Pseudo-TAs Implementing the Debug APIs

Pseudo-TAs (pTAs) provide protocol based services for clients, and may be connected to from either inside or outside of the TEE.

As an implementation choice, either of the following is acceptable for a pTA implementing the APIs described in this document:

- The pTA may be a real TA in every sense.
- The pTA is addressable via the TEE Client API as though a TA, but may not have the same management characteristics as a TA. For example, it may be tightly integrated with the Trusted Framework and so not lockable, removable, or independently upgradable.

The functionality required here cannot be strictly implemented using the current GlobalPlatform TEE API specifications, as it extends the Trusted Core Framework and Trusted Functions (see TEE System Architecture [TEE Sys Arch] section 3.1.2). The methods for making such extensions in a TA are outside the scope of current GlobalPlatform specifications. The required functionality offers a service over the TEE Client API and so counts as a pTA rather than a simple framework extension.

2.2.2 Monitoring Applications

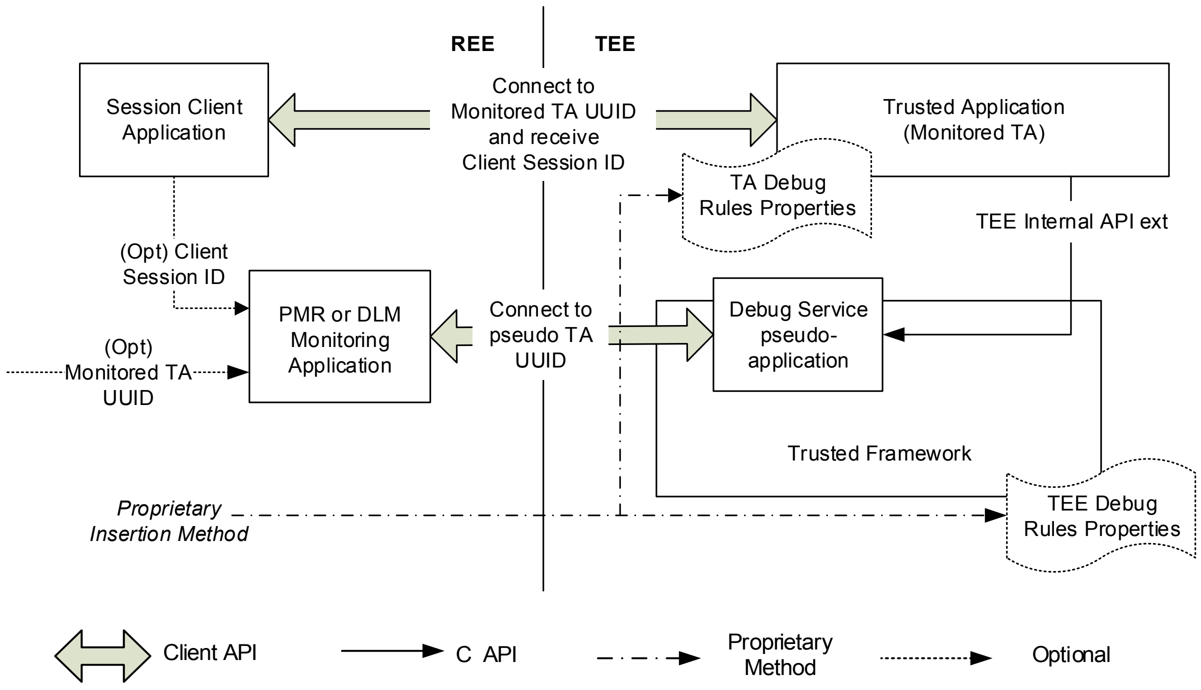
The Monitoring Application is shown in Figure 2-1 as resident in the REE. Support for a Monitoring Application is a requirement for the PMR interface, as it shall be available for functional compliance testing, but support for a Monitoring Application in this location is **optional** for the DLM interface, where proprietary methods may be used to expose the stream to the developer.

A Monitoring Application may choose to monitor PMR events on the following granularities (authorization allowing):

- Specific session (identified by TA session identifier and TA UUID)
- Specific TA (identified by TA UUID)
- Any TA

With regard to DLM streams, the debug service shall connect to a specific session of a specific TA.

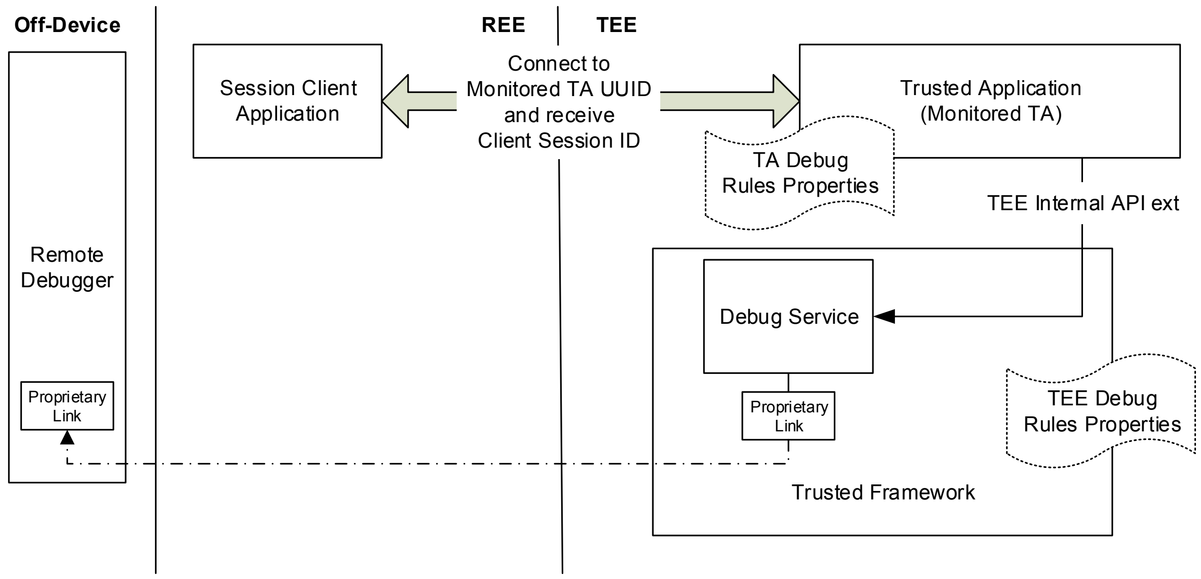
Figure 2-1: General Case PMR and DLM Architecture Diagram


REE
TEE
Session Client Application
Connect to Monitored TA UUID and receive Client Session ID
Trusted Application (Monitored TA)
TA Debug Rules Properties
TEE Internal API ext
(Opt) Client Session ID
PMR or DLM Monitoring Application
Connect to pseudo TA UUID
Debug Service pseudo-application
(Opt) Monitored TA UUID
Trusted Framework
Proprietary Insertion Method
TEE Debug Rules Properties
Client API
C API
Proprietary Method
Optional

The DLM facility uses an almost identical architecture to the PMR, but for the DLM implementation it is acceptable to use any of the following:

- A DLM Monitoring Application similar to the PMR Monitoring Application shown in Figure 2-1
- A remote monitor as shown in Figure 2-2
- Both a Monitoring Application and a remote monitor

Figure 2-2: Additional DLM Architecture Diagram Including Optional Proprietary Link


Off-Device
REE
TEE
Session Client Application
Connect to Monitored TA UUID and receive Client Session ID
Trusted Application (Monitored TA)
TA Debug Rules Properties
TEE Internal API ext
Remote Debugger
Debug Service
Proprietary Link
Proprietary Link
TEE Debug Rules Properties
Trusted Framework

For PMR or DLM, where a Monitoring Application is supported, both REE and TEE based Monitoring Applications shall be supported. This is not shown in the diagrams for simplicity.

Note also that the PMR facility is mandatory if this specification is part of the target TEE environment, implementation of the DLM facility is always optional.

2.3 Mandatory and Optional Parts of this Specification

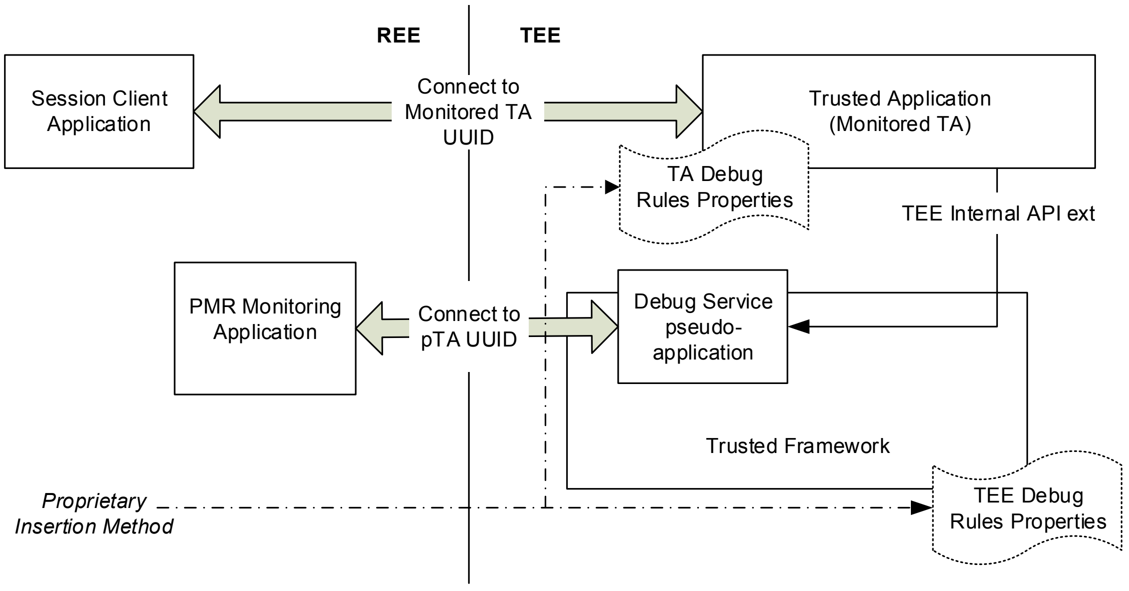
Table 2-1 describes the mandatory and optional parts of this specification.

Table 2-1: Mandatory and Optional Parts of the Specification

Part	Title	Requirement
Chapter 3	TA Post Mortem Reporting (PMR) Protocol	Mandatory
Chapter 4	TEE Debug General Extension	Mandatory
Section 5.3	Client Side DLM Protocol	Optional
Section 5.4	Internal DLM API	Conditional: Mandatory if section 5.3 is implemented.
Section 5.4.4.1	Property Indicating DLM Installed on the System	Conditional: Mandatory if section 5.4 is implemented.

Implementation of only the mandatory functionality results in a minimal system that supports only PMR, as illustrated in Figure 2-3.

Figure 2-3: Minimal Implementation


REE
TEE
Session Client Application
Connect to Monitored TA UUID
Trusted Application (Monitored TA)
TA Debug Rules Properties
TEE Internal API ext
PMR Monitoring Application
Connect to pTA UUID
Debug Service pseudo-application
Trusted Framework
Proprietary Insertion Method
TEE Debug Rules Properties

2.3.1 Status of Examples Found in this Specification

All sections marked as examples in this specification are provided as informative rather than normative material. This includes all code in the appendices, along with examples in various sections of the document.

2.4 Data Types

2.4.1 Basic Types

This specification makes use of the integer and Boolean C types as defined in the C99 standard (ISO/IEC 9899:1999) [C99]. Table 2-2 lists the basic types.

Table 2-2: Basic Types

Type	Definition
`uint64_t`	Unsigned 64-bit integer
`int64_t`	Signed 64-bit integer
`uint32_t`	Unsigned 32-bit integer
`int32_t`	Signed 32-bit integer
`uint16_t`	Unsigned 16-bit integer
`int16_t`	Signed 16-bit integer
`uint8_t`	Unsigned 8-bit integer
`int8_t`	Signed 8-bit integer
`bool`	Boolean type with the values `true` and `false`
`char`	Character; used to denote a byte in a null-terminated string encoded in UTF-8

In this specification, bits in integers are numbered from `0` (least-significant bit) to 7, 15, 31, or 63 (most-significant bit), depending on the size of the integer.

2.4.2 Additional Types

Table 2-3 lists additional types used by this specification.

Table 2-3: Additional Types

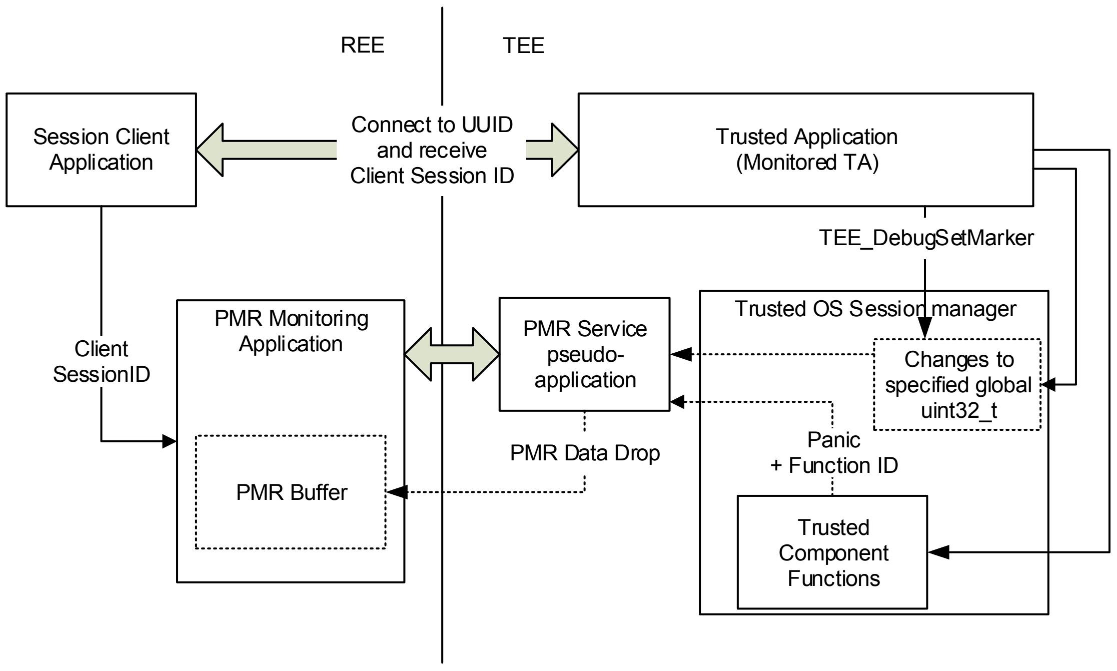
Type	Definition
`TEE_UUID`, `TEEC_UUID`	`TEE_UUID` is a Universally Unique Resource Identifier type as defined in [RFC 4122]. This type is used to identify Trusted Applications and clients. For TEE specific usage see [TEE Internal API] section 3.2.3.
`TEE_Result`, `TEEC_Result`	`TEE_Result` is the type used for return codes from the APIs. For details, see [TEE Internal API] section 3.2.2.
`atomic_t`	An integral type whose value can be read from and written to memory in a manner such that partially updated values are never observed. May require the use of special macros to achieve this. It shall store a value range at least equivalent to the address space of the TEE.

3 TA Post Mortem Reporting (PMR) Protocol

This service enables a PMR Monitoring Application to receive Post Mortem Reporting information. This information may be from a particular TA (defined by UUID and (optionally) client session ID), or from any TA in the device. It receives PMR information only from TAs which it is authorized to monitor.

Any compliant implementation of this specification shall provide the PMR functionality.

Figure 3-1: PMR Monitoring Introduction


REE
TEE
Session Client Application
Connect to UUID and receive Client Session ID
Trusted Application (Monitored TA)
TEE_DebugSetMarker
Client SessionID
PMR Monitoring Application
PMR Buffer
PMR Service pseudo-application
PMR Data Drop
Trusted OS Session manager
Changes to specified global uint32_t
Panic + Function ID
Trusted Component Functions

Informative examples of process involved

Example 1: Monitoring a particular client session ID

1. Once the Session Client Application has successfully opened a Client API Session with the Monitored TA and received a client session ID, it passes this to a PMR Monitoring Application.
    - It may or may not be desirable to host both the Session Client Application and PMR Monitoring Application in a single application and there is no restriction preventing this. They are separated here for clarity.
2. The PMR Monitoring Application expresses an interest in Panic events by opening a monitoring session to the PMR service pTA. It associates this with a particular client session ID by providing that ID.
3. When a Panic occurs associated with that client session, the PMR Monitoring Application is informed of this through the relevant Session Client Application.
4. The PMR Monitoring Application then receives any appropriate PMR data that Debug Rules Properties have authorized.
5. The PMR Monitoring Application then processes the returned PMR data as it sees fit.
6. Once the PMR Monitoring Application has received data about a Panic:
    - The PMR Monitoring Application will not receive any other PMR data from that client session ID.
    - It is advisable to close this monitoring session between the PMR Monitoring Application and the PMR service pTA.

Example 2: Monitoring for any Panics in any TAs resident in the TEE

1. The PMR Monitoring Application expresses an interest in Panic events by opening a session to the PMR service pTA.
2. The PMR Monitoring Application waits for any Panic to occur.
3. The PMR Monitoring Application receives any appropriate PMR data authorized by the Debug Rules Properties of the TEE and executing TAs when Panics occur.
4. The PMR Monitoring Application then processes the returned PMR data as it sees fit.
5. The PMR Monitoring Application does one of the following:
    - Returns to state 2 to continue waiting for more Panics.
    - Indicates that it is no longer interested in further Panics by closing its session with the PMR service pTA.

3.1 PMR Service pTA Availability

A PMR service shall only respond to a client TA with debug information on a device with suitable debug authorization. Debug authorization shall be determined by a combination of the TA specific and TEE specific Debug Rules Properties.

Such properties may be changed through one or more of:

- Proprietary means
- TA management updating a TA Download Package with different properties (See [TEE Mgmt Fwk])
- TEE management updating the TEE with different properties (See [TEE Mgmt Fwk])

All three of these methods shall be as cryptographically robust as the cryptographic protection applied to other TEE resource protection such as the TEE Internal API Trusted Storage (see Chapter 5 of [TEE Internal API]).

Any routine defined by a GlobalPlatform Device Technology TEE specification may generate a Panic if it detects a relevant hardware failure OR is passed invalid arguments that could have been detected by the programmer, even if no Panics are listed for that routine.

The connection to the PMR Service pTA shall always be cancellable (see section 2.1.3 of [TEE Internal API]).

3.1.1 Releasing the Panicked TA State Data inside the Trusted OS

Once all PMR Monitoring Applications that are entitled to receive the PMR data have either received that data or closed their session with the PMR service pTA, then the Trusted OS may release the state data associated with the panicked session.

3.2 PMR Service Command Protocol

This client command interface connects a Monitoring Application (in either the REE or TEE) to a PMR message stream being provided by the PMR Service pTA.

Figure 3-2 shows the flow for a PMR session where a particular session ID is being monitored. If the session was monitoring a multi instance TA or all available TAs in the TEE, then rather than closing it could loop back after each PMR event to the `CMD_PMR_WAIT` state.

Figure 3-2: Monitoring Application View of PMR with Session ID Available

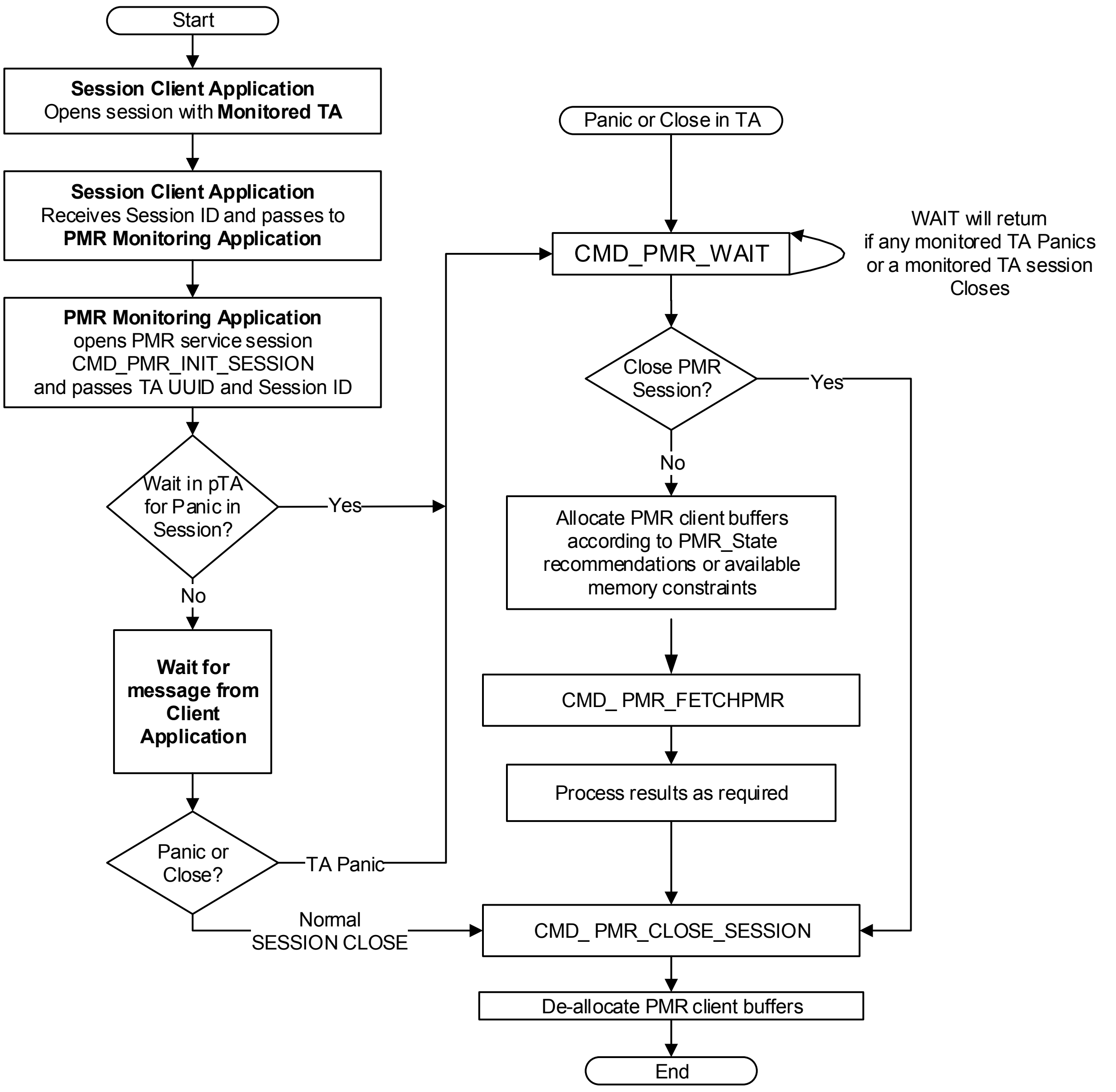
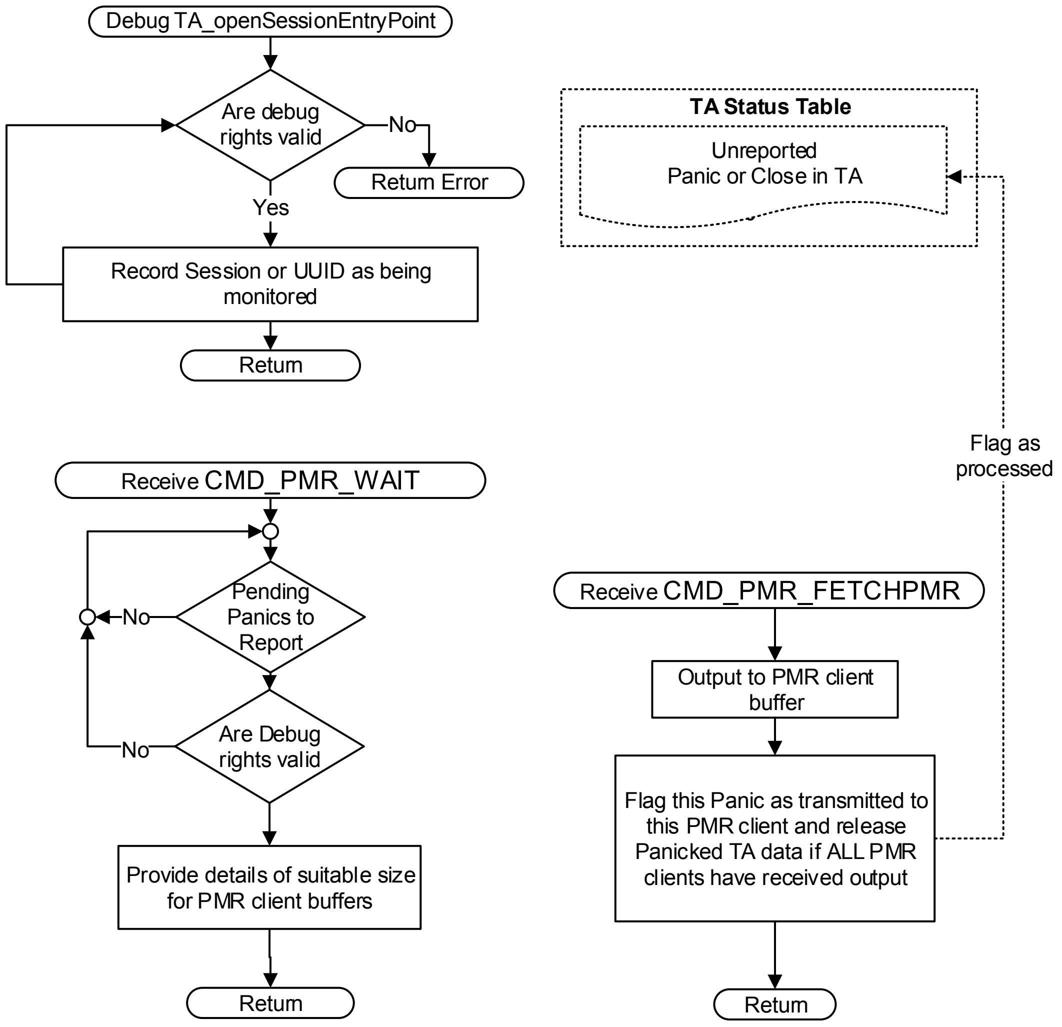

Start
Session Client Application Opens session with Monitored TA
Session Client Application Receives Session ID and passes to PMR Monitoring Application
PMR Monitoring Application opens PMR service session CMD_PMR_INIT_SESSION and passes TA UUID and Session ID
Wait in pTA for Panic in Session?
Yes
No
Wait for message from Client Application
Panic or Close?
TA Panic
Normal SESSION CLOSE
Panic or Close in TA
CMD_PMR_WAIT
WAIT will return if any monitored TA Panics or a monitored TA session Closes
Close PMR Session?
Yes
No
Allocate PMR client buffers according to PMR_State recommendations or available memory constraints
CMD_ PMR_FETCHPMR
Process results as required
CMD_ PMR_CLOSE_SESSION
De-allocate PMR client buffers
End

Figure 3-3: Monitoring from the Point of View of the PMR Service pTA


Debug TA_openSessionEntryPoint
Are debug rights valid
No
Return Error
Yes
Record Session or UUID as being monitored
Return
TA Status Table
Unreported Panic or Close in TA
Flag as processed
Receive CMD_PMR_WAIT
Pending Panics to Report
No
Are Debug rights valid
No
Provide details of suitable size for PMR client buffers
Return
Receive CMD_PMR_FETCHPMR
Output to PMR client buffer
Flag this Panic as transmitted to this PMR client and release Panicked TA data if ALL PMR clients have received output
Return

3.2.1 Header File

The header file for the TEE Debug Specification PMR API must have the name "`tee_client_PMR_api.h`".

```
#include "tee_client_PMR_api.h"
```

This header will include the following definition:

```
#ifndef GPD_TEE_TA_DEBUG_1_0_1
#define GPD_TEE_TA_DEBUG_1_0_1
#endif
```

3.2.2 Constants

3.2.2.1 Return Codes

Table 3-1 lists the return codes used by the PMR interface.

Table 3-1: API Return Codes – PMR Interface

Constant Names and Aliases		Value
TEE_SUCCESS	TEEC_SUCCESS	0x00000000
PMR_MONITORED_TA_PANIC		0x00251001
PMR_MONITORED_TA_CLOSED		0x00251002
PMR_MONITORED_SESSION_CLOSED		0x00251003
PMR_SESSION_CLOSE_RULE_CHANGE		0x00251004
PMR_ERROR_NO_PANIC		0x00251005
ERR_PMR_CLIENT_SESSIONID_REQD		0xF0251001
ERR_PMR_UUID_REQD		0xF0251002
ERR_PMR_ACCESS_DENIED		0xF0251003
ERR_PMR_INVALID_PMR_STATE		0xF0251004
TEE_ERROR_CANCEL	TEEC_ERROR_CANCEL	0xFFFF0002

3.2.2.2 PMR Service UUID

A debug service shall be connected in the same manner as any other service offered by a TA.

For each application wishing to access debug output, a session open request (`TEE_OpenTASession` or `TEEC_OpenSession` as appropriate) shall be made to the specific UUID assigned to the PMR service.

The PMR service is identified by the following UUID.

```
static const TEEC_UUID PMRservice =
{
0x09A193B3, 0x688A, 0x476F,
{0x88, 0xB3, 0x6A, 0xD7, 0xd9, 0xDA, 0x93, 0x22}
};
```

See section 5.2.2 of the TEE Client API Specification [TEE Client API] or section 4.9 of [TEE Internal API] for more details on connecting to a service provided by a TA.

3.2.3 PMR Session Commands

3.2.3.1 PMR Command Values

Table 3-2 lists the command values defined by the PMR interface.

Table 3-2: PMR Command Values

Constant Names and Aliases	Value
`CMD_PMR_INIT_SESSION`	`0x00000004`
`CMD_PMR_WAIT`	`0x00000005`
`CMD_PMR_FETCHPMR`	`0x00000006`
`CMD_PMR_CLOSE_SESSION`	`0x00000007`

3.2.3.2 `CMD_PMR_INIT_SESSION`

Description

This command initializes a PMR service.

It uses the `PMR_State` structure to inform the PMR service of the PMR sources that are requested in this session.

Command Details

- `commandID`: 4
- `params[1]`
  - `[in] PMR_State`

    See section 3.2.4.2 for a description of `PMR_State`.

    Here `PMR_State` is used to present to the PMR session the TA UUID and client session ID if required.

Return Code

- `ERR_PMR_INVALID_PMR_STATE`

  The device has no TA with the specified UUID or there is no session with the specified client session ID.

  If the device has a TA with the specified UUID that is not running, then the PMR session is valid and the PMR service shall report events from the moment the TA starts to execute.

  The session is not created.

- `ERR_PMR_CLIENT_SESSIONID_REQD`

  The combined Debug Rules Properties mean that client session ID is required, and it was not provided by `PMR_State`.

  The session is not created.

- `ERR_PMR_UUID_REQD`

  The combined Debug Rules Properties mean that Monitored TA UUID is required, and it was not provided by `PMR_State`.

  The session is not created.

- `ERR_PMR_ACCESS_DENIED`

  The combined Debug Rules Properties do not authorize this debug session.

  The session is not created.

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session. See [TEE Client API].

- `TEEC_SUCCESS`

  The command was completed successfully.

3.2.3.3 `CMD_PMR_WAIT`

Description

This command shall return panicked TA identifying information along with the size of the buffer required to contain the available debug information.

The buffer sizes shall be `0` when the Monitored TA has closed, as there is no Panic to report.

If multiple Panics have occurred, each of which is valid to be reported by this PMR service pTA session, then they shall be processed in a First In, First Out manner.

In a specific PMR debug session:

- To process a Panic requires a call to `CMD_PMR_WAIT` to discover the optimal buffer sizes, followed by a call to `CMD_PMR_FETCHPMR` to retrieve the available Panic details.
- Multiple consecutive calls to `CMD_PMR_WAIT`, with no intervening calls to `CMD_PMR_FETCHPMR`, and which all return `PMR_MONITORED_TA_PANIC`, will populate structure `PMR_State` with identical data.

This command may return Monitored TA session close indication (`PMR_MONITORED_TA_CLOSED`) only when linked to a designated client session ID.

This command shall block until one of the following:

- A Panic that it is allowed to report occurs or has occurred.
- The specified TA session is closed or has closed.
- The PMR session is closed.
- The command is cancelled.

Command Details

- `commandID`: 5
- `params[1]`
  - `[out] PMR_State`

    See section 3.2.4.2 for a description of `PMR_State`.

    Here the "Set By Either" parameters are used by the PMR service to better identify the Panic reason and the size parameters are set with the current allocation requirements for the output memory buffers.

Return Code

- `PMR_MONITORED_TA_PANIC`

  Post mortem data is now available and the `PMR_State` buffer describes the PMR that is available.
- `PMR_MONITORED_TA_CLOSED`

  A Monitored TA has closed.
- `PMR_MONITORED_SESSION_CLOSED`

  The PMR service session has been closed.

  This shall only occur when monitoring a particular client session ID.

- `PMR_SESSION_CLOSE_RULE_CHANGE`

  The PMR service session shall be closed because a remote entity has changed the TA or TEE Debug Rules Properties and the PMR session is no longer valid.

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session. See [TEE Client API].

3.2.3.4 CMD_PMR_FETCHPMR

Description

This command populates the PMR_MessageBuffer based on the same Panic event reported by CMD_PMR_WAIT.

The contents of the PMR_MessageBuffer shall be associated with the PMR_State returned from CMD_PMR_WAIT.

If multiple Panics have occurred, each of which is valid to be reported by this PMR service session, then they shall be processed in a First In, First Out manner.

In a specific PMR debug session:

- To process a Panic shall require a call to CMD_PMR_WAIT to discover the buffer sizes, followed by a call to CMD_PMR_FETCHPMR to retrieve the details.
- After the initial call, consecutive calls to CMD_PMR_FETCHPMR, with no call to CMD_PMR_WAIT, will return PMR_ERROR_NO_PANIC and have no effect on PMR_MessageBuffer.

Command Details

- commandID: 6
- params[1]
  - [out] PMR_MessageBuffer

    Memory buffer, optimally based on the sizes returned by CMD_PMR_WAIT.

Return Code

- PMR_MONITORED_TA_PANIC

  Post mortem data shall be present in the supplied PMR_MessageBuffer.
- PMR_ERROR_NO_PANIC

  This error indicates that the CMD_PMR_FETCHPMR has been called when there is no relevant Panic to report.
- PMR_SESSION_CLOSE_RULE_CHANGE

  The PMR service session shall be closed because a remote entity has changed the debug rules.
- TEEC_ERROR_CANCEL

  The command was cancelled by a TEEC_RequestCancellation on this session.

3.2.3.5 `CMD_PMR_CLOSE_SESSION`

Description

This command closes the session and informs the PMR service pTA that any outstanding Panic events may not need to be retained. The PMR service pTA should clear any outstanding Panic state that is not associated with other PMR sessions.

Command Details

- `commandID`: 7
- `params[0]`
  - No parameters are used for this command.

Return Value

None

3.2.4 Structures

3.2.4.1 State, Stack, and Heap

In the following structures we pass three distinct chunks of proprietary data from the TEE to the Monitoring Application for interpretation.

State

This is assumed to be a blob of data that the TEE uses to register some global facts about the TA execution state. It is expected to be small in size when compared to something such as the TA heap.

Such state will be particular to a Trusted OS and so suitable Trusted OS specific tools may be needed to interpret the contents. In some implementations the state of a particular TA may be so interleaved with the state of other TAs that the PMR service will expose it only when the lowest of PMR Debug Rules Properties restrictions are in effect.

Stack

This is assumed to be a blob of data that the TEE uses to contain the TA stack information. It is expected to be dynamic in size and potentially quite large in comparison to the State structure.

It is expected that useful information can be gained by a Monitoring Application examining only the most recently added segments (as they will relate to the functions that most immediately caused the Panic to occur). No assumption is made by the API as to whether the stack grows upwards, downwards, or uses block addition.

The stack structure may be particular to a Trusted OS and so suitable tools may be needed to interpret the contents. In some implementations the stack of a particular TA may be so interleaved with the stack of other TAs that the PMR service will expose it only when the lowest of PMR Debug Rules Properties restrictions are in effect.

Heap

This is assumed to be a blob of data that the TEE uses to contain the TA heap information. It is expected to be relatively static but still potentially dynamic in size and relatively large.

No assumption is made by the API as to how the heap is arranged.

The heap structure may be particular to a Trusted OS and so suitable tools may be needed to interpret the contents. In some implementations the heap of a particular TA may be so interleaved with the heap of other TAs that the PMR service will expose it only when the lowest of PMR Debug Rules Properties restrictions are in effect.

No restriction is made on the contents of these buffer areas, so a PMR service may pass additional data through these areas, such as a control structure to better support proprietary tools the TEE vendor may provide to enhance the Monitoring Application. Actual arrangement of state, stack, and heap data in these buffer areas does not need to directly correspond to the internal arrangement in the TEE, though to be useful any rearrangements need to be interpretable by the Monitoring Application.

3.2.4.2 Client Side `PMR_State`

The PMR session parameters are passed using the following structure:

```
typedef struct
{
    /* fixed parameters */
    [inout] TEEC_UUID     monitoredTA;             /* set by either */
    [inout] TEEC_UUID     monitoredSession;        /* set by either */

    /* device dependent parameters */
    [out] uint32_t        stateSize;               /* set by PMR service */
    [out] uint32_t        stackSize;               /* set by PMR service */
    [out] uint32_t        heapSize;                /* set by PMR service */
} PMR_State;
```

Description of `PMR_State` Structure Fields

- `monitoredTA`
  - UUID of the Monitored TA.
  - This value is optional and may be the nil UUID[1] to indicate an empty value.
    - If the nil UUID, then all debuggable sessions on the monitored TEE shall report post mortems to this debug session and the `monitoredSession` value is ignored.
- `monitoredSession`
  - The Monitored TA to Session Client Application, session identifier.
  - This value shall be ignored if this value or `monitoredTA` is the nil UUID.
  - If not the nil UUID:
    - This value contains a session unique identifier for the connection between the Session Client Application and the Monitored TA.
    - This value shall be the same as returned by the function `TEE_GetPropertyAsUUID` (see [TEE Internal API]) when the property name `gpd.ta.session.ID` is presented by the Monitored TA.
    - For Multi Instance TA, this value shall identify a unique instance and only Panics in that instance of the TA shall be reported to this PMR session.
    - For Single Instance TA, this value shall identify the active session to the TA instance, but a Panic of ANY session of the TA shall be reported to this PMR session.
- `stateSize`
  - The required memory in bytes to hold the TEE specific TA state information.
  - Shall be `0` if this information is not available to the client.
- `stackSize`
  - The required memory in bytes to hold the TEE specific TA **maximum** stack information.
  - Shall be `0` if this information is not available to the client.

[1] As defined in [RFC 4122], a UUID with all 128 bits set to zero.

- `heapSize`
    - The required memory in bytes to hold the TEE specific heap information.
    - Shall be `0` if this information is not available to the client.

The `monitoredTA` UUID shall be a constant for a specific Monitored TA and may be provided to the PMR Monitoring Client Application in a number of ways. Injection of UUID into the PMR Monitoring Client application is out of scope for this document.

To restrict debug to a specific session, the Client Application shall communicate the `monitoredSession` UUID to the PMR Monitoring Client Application. Such communication is out of scope of this document.

In both cases see Figure 2-1 on page 12.

3.2.4.3 Client Side `PMR_MessageBuffer`

The contents of some sections of this message buffer are implementation dependent.

The following aspects are defined.

The buffer shall be filled with the following structures.

- A set of PMR communication parameters
- An EE specific set of program status information
- An execution process specific set of program status information
- A stack dump from the top to bottom. If there is insufficient space for the whole stack, then this is flagged and as much as possible placed in the buffer.

The PMR communication parameters shall be passed using the following structures:

`TEE_RefAddress` structure

```
typedef union
{
    struct
   {
        uint32_t Hi; /* Top 32 bits of address – SBZ in 32 bit memory systems */
        uint32_t Lo; /* Lower 32 bits of address – */
    } Addr32bit  ;
    uint64_t Addr64Bit; /* 64 bit Address (optional on 32 bit systems)*/
} TEE_RefAddress;
```

Description of `TEE_RefAddress` Structure Fields

In 32-bit systems it is expected that `Addr32Bit.Hi` shall be left as `0` and the address access shall make use of `Addr32Bit.Lo`. `Addr64Bit` is provided for systems where greater than 32-bit addressing is needed.

`PMR_MessageBuffer` structure

```
typedef struct
{
    /* fixed parameters */
    TEEC_UUID    sourceUUID;       /* set by PMR: UUID of Panicked TA */
    TEEC_UUID    sessionID;        /* set by PMR: UUID of Session */

    uint16_t specNumber;           /* Panic specification number, set by PMR */
    uint16_t functionNumber;       /* Panic function number, set by PMR */

    uint32_t markValue;                /* set by last call to
                                              TEE_DebugSetMarker */

    TEE_Result   panicReasonCode;      /* Panic reason code - set by PMR */

    /* device dependent parameters */
    uint32_t stateSize;                /* set by client indicating available
                                              buffer size */

    uint32_t stackSize;                /* set by client indicating available
                                              buffer size */
    TEE_RefAddress   stackRefAddress;  /* set by PMR service, address in TEE */

    bool     completeStack;            /* True if there was room for the entire
                                              TA stack, otherwise only the latest
                                              entries shall be placed in the
                                               buffer */

    uint32_t heapSize;                 /* set by client indicating available
                                              buffer size */
    TEE_RefAddress   heapRefAddress;       /* set by PMR service, address in TEE */

} PMR_MessageBuffer;
```

Description of `PMR_MessageBuffer` Structure Fields

- `sourceUUID`

  UUID of panicked TA

- `sessionID`

  UUID of panicked Session as would be returned by `gpd.ta.session.ID` Property (see section 4.2)

- `specNumber`

  A number that identifies the GlobalPlatform specification in which the function that has panicked shall be found.

  See section 4.3, for an explanation of specification numbers.

  If the value has been set by a return other than that due to `PMR_MONITORED_TA_PANIC`, then it shall be `0`.

- `functionNumber`

  A number that identifies the particular function that panicked. It is valid in the context of the given `specNumber` value, and hence in the context of a particular specification.

  See section 4.4, for an explanation of function numbers.

  If the value has been set by a return other than that due to `PMR_MONITORED_TA_PANIC`, then it shall be `0`.

- `markValue`

  The current value of the Monitored TA's global variable associated with debug by the use of the `TEE_DebugSetMarker` function (see section 4.1).

  This variable should be used to provide an indication of the TA program state.

  If no variable has been associated with debug, then this shall be set to `0` by the PMR Service pTA.

- `panicReasonCode`

  The Panic Reason Code is provided to reflect the `panicCode` provided to `TEE_Panic` (see [TEE Internal API] section 4.8.1).

  In the case of a Panic for reasons other than a call to `TEE_Panic`, if no other code in [TEE Internal API] Table 3-1 is applicable, then `TEE_ERROR_GENERIC` shall be used.

- `stateSize`

  Indicates the available buffer size that the client is providing to the PMR service. The buffer is used for the PMR service to store a copy of the TA state information. Optimally it should be the same size as that returned by `CMD_PMR_WAIT`. If it is smaller than that size, then it shall be implementation specific as to what data is provided. If it is larger, then excess buffer should not be used.

- `stackSize`

  Indicates the available buffer size that the client is providing to the PMR service. The buffer is used for the PMR service to store a copy of some or all of the TA stack. Optimally it should be the same size as that returned by `CMD_PMR_WAIT`. If it is smaller than that size, then it should contain copies of the most recently pushed sections of the stack.

- `stackRefAddress`

  This holds the equivalent of the address `PMR_STACK_POINTER(MessageBuffer)` but in terms of the TA memory map. As the TA memory map may be different from the memory map used to hold `MessageBuffer`, this value may be used in simple architectures to interpret actual address locations of the stack data in the buffer.

  See below for definition of `PMR_STACK_POINTER(MessageBuffer)`.

- `completeStack`

  This Boolean flag indicates whether the stack returned by a Panic is the entire stack.

  Because of the format of data held on a stack, an incomplete stack (which contains just the latest segments pushed on the stack) is still of use and so may be provided if the stack buffer is insufficient for a full stack dump.

- `heapSize`

  Indicates the available buffer size that the client is providing to the PMR service. The buffer is used for the PMR service to store a copy of the TA heap information. Optimally it should be the same size as that returned by `CMD_PMR_WAIT`. If it is smaller than that size, then it shall be implementation specific as to what data is provided.

- `heapRefAddress`

  This holds the equivalent of the address `PMR_HEAP_POINTER(MessageBuffer)` but in terms of the TA memory map. As the TA memory map may be different from the memory map used to hold `MessageBuffer`, this value may be used to interpret address locations.

  See below for definition of `PMR_HEAP_POINTER(MessageBuffer)`.

The total memory to be allocated should be (in bytes):

```
#define SIZEOF_PMR_MessageBuffer  (sizeof(PMR_MessageBuffer) \
                          + ((PMR_MessageBuffer) PMR_MessageBuffer).stateSize \
                          + ((PMR_MessageBuffer) PMR_MessageBuffer).stackSize \
                          + ((PMR_MessageBuffer) PMR_MessageBuffer).heapSize )
```

The start address of the three areas shall then be defined using the following defines.

```
#define PMR_STATE_POINTER(MessageBuffer) \
                          (((uint8_t*) &MessageBuffer) \
                          + sizeof(PMR_MessageBuffer))

#define PMR_STACK_POINTER(MessageBuffer) \
                           (((uint8_t*) &MessageBuffer) \
                          + sizeof(PMR_MessageBuffer)\
                          + ((PMR_MessageBuffer) MessageBuffer).stateSize )

#define PMR_HEAP_POINTER(MessageBuffer) \
                           (((uint8_t*) &MessageBuffer) \
                          + sizeof(PMR_MessageBuffer) \
                          + ((PMR_MessageBuffer) MessageBuffer).stateSize \
                          + ((PMR_MessageBuffer) MessageBuffer).stackSize )
```

3.2.5 PMR Code Example: Using the TEE PMR Client Protocol

For an example of a PMR client, please see Annex C.

4 TEE Debug General Extensions

The extensions described in this chapter are requirements for both PMR and DLM implementations. They provide general support to enable the debug session to communicate in a device agnostic manner.

Any compliant implementation of this specification shall provide this functionality.

4.0 Header File

The header file for using the TEE Debug Specification general extensions must have the name "tee_internal_DSGE_api.h".

```
#include "tee_internal_DSGE_api.h"
```

This header will include the following definition:

```
#ifndef GPD_TEE_TA_DEBUG_1_0_1
#define GPD_TEE_TA_DEBUG_1_0_1
#endif
```

4.1 TEE_DebugSetMarker

```
void TEE_DebugSetMarker(
    [in]     uint32_t*     markValue     /* markValue shall be a uint32_t* */
                                         /*           shall be a global variable */
);
```

Description

This function enables both PMR and DLM services to be aware that execution has passed a particular location in the code base by enabling the setting of a reference value.

Changes to this reference value may be used to indicate the current program state without the need to interpret a TEE specific program counter context.

This value is set by the TA and it is up to the TA to use this capability, along with other reporting, to enable understanding of the current program location without the need to translate the device specific Program Counter.

This function is included to provide a device and processor architecture agnostic method of recording the location in the TA code space. It would obviously be better to record and interpret the actual program counter and stack state. This may be done for the PMR event through analysis of the device proprietary TA state buffer. For DLM events no current method is suggested for communicating the state of the actual Program Counter.

When a global variable is assigned by use of this function, any PMR or DLM event shall report the current value of that variable.

If no global variable is assigned, then the PMR or DLM events shall report a `0` value. If a null pointer value is assigned, then the `markValue` is returned to this unassigned state.

No guarantee is made that `markValue` may not be corrupted by events causing a Panic or other programming flaws in the TA.

It is recommended that the data pointed to by `markValue` is held in a `volatile` variable to avoid compiler optimizations removing value changes.

Parameters

- `markValue`
  - Pointer to variable that shall be reported.
  - `markValue` shall point to a global variable.
  - The value shall be transported as a `uint32_t`. The monitoring client may interpret this value to extract other values packed into this variable.
  - Only one global variable shall be so assigned at any one time for a particular TA instance.
    - Further calls to `TEE_DebugSetMarker` shall change which global variable is assigned.
  - When a PMR or DLM action occurs, the current value of this global variable shall be included in that report or message.
  - If a TEE Trusted OS Component function, that caused a Panic, may modify the relevant global variable, then the value is not guaranteed.

Return Value

None

Panic Reasons

- `markValue` points to a location that is not 32-bit aligned.
- `markValue` points to a location that is not a global non-const variable.

Example Usage

```
/*
 * The following test is to ascertain that ONLY data set 7 causes a Panic and
 * that no other condition before data set 7 is processed causes a Panic.
 * This is investigating if sequences of calls to ProcessData() cause incorrect
 * Panics (a situation that often arises from poor allocating cleanup)
 */

volatile uint32_t global_PMR_var;

#define TEST1 0x00010000
#define DONE1 0x10010000

TEE_DebugSetMarker(&global_PMR_var);

for (int v = 1; v < 20; v++)
{
    global_PMR_var = v | TEST1;    /* track state of test */
    ProcessData(dataset[v]);  /* where it is known that */
                               /* dataset[7] should panic */
}

global_PMR_var = DONE1;
```

4.2 `gpd.ta.session.ID` Property

A TA may discover a session specific ID for the current session context that it is executing in.

This is available through enquiring of the property interface of the value `gpd.ta.session.ID`.

Typically, the [TEE Internal API] command `TEE_GetPropertyAsUUID` will be used to retrieve this value.

The TA can pass this information by an implementation defined mechanism to a Monitoring Application which makes use of the PMR or DLM services.

Property name:

> `gpd.ta.session.ID`

Property type:

> `TEEC_UUID`
>
> As a TEE context specific session specific ID, it is acceptable that this ID creation may not conform to [RFC 4122], but it shall be based on a unique 128-bit value.
>
> It shall be retrievable as a `TEEC_UUID` type from the property interface.

Property meaning:

> This UUID uniquely identifies the session in the context of the current monitored TA.
>
> In the context of a particular session on a particular TA the property shall be unique.
>
> - No other session for a particular TA shall be identified with this value.
> - The session.ID value shall be at least statistically unique between TEE boot cycles.

This requirement does not specifically exclude a TA with a different `gpd.ta.appID` (from that of the monitored TA) from having the same `gpd.ta.session.ID`, and so for a client to uniquely identify a session the client shall use both `gpd.ta.appID` and `gpd.ta.session.ID`. A Trusted OS Component implementation may use the `gpd.ta.session.ID` property to provide a TEE global unique session identity.

The `gpd.ta.session.ID` property may be communicated from the monitored TA, via its client, to the Monitoring Application. The Monitoring Application can then use this ID, in association with the TA UUID, to raise a session specific debug context for either PMR or DLM debug or both.

4.3 Specification Number

Specification number (`specNumber`) is a number indicating the GlobalPlatform specification from which the panicked function originates.

Each GlobalPlatform specification is identified by a document reference of the form XXX_YYY_NNN, where:

XXX indicates the originating GlobalPlatform Committee.

e.g. GPD indicates a document from the GlobalPlatform Device Committee.

YYY indicates the type of document.

e.g. SPE indicates that the document is a specification.

NNN distinguishes between documents from the committee and in that category.

Each GlobalPlatform TEE specification document that exposes an API to the TA shall assign this value the numerical part of that specification's document reference.

Example:

[TEE Internal API] has the document reference GPD_SPE_010.

Therefore, a Panic in a function defined in [TEE Internal API] shall have a specification number (`specNumber`) of value 10.

4.4 Function Numbers

Function number (`functionNumber`) is a number indicating the function within which a Panic occurred in the context of a given GlobalPlatform specification.

The function number is a 16-bit value, with the high 8 bits used to group values according to functional groupings (categories) found in the particular specification.

A table of designated function numbers accompanies any GlobalPlatform TEE specification that declares functions exposed to the TA. For this debug specification and for values not related to function layer Panics, the assigned number shall be found in Table A-1.

In addition, values need to be defined for proprietary functions. Ranges for such values are defined in Table 4-1.

Table 4-1: Function Numbering

Function	Function Number in hexadecimal	Function Number in decimal

Function numbers in the range `0x0001` to `0x00FF` are reserved for Panics due to TA execution errors, rather than associated with a particular function.
While the PMR interface is designed to detect Panics caused by misuse of TEE API functions, it is also possible for a TA to Panic due to coding errors. An example of such an error is performing a divide by zero. These form a special class of error that is reported without an associated function and grouped under the heading of "TA Panics".
	`0x0001`
	to
	`0x00FF`
		1 to 255

Function numbers in the range `0xNN00` to `0xNN7F` (where `0x01` ≤ NN ≤ `0xEF` and NN designates a Category) are reserved for use by GlobalPlatform TEE specifications that declare functions exposed to the TA.	`0xNN00`
	to
	`0xNN7F`
	where
	`0x01` ≤ NN ≤ `0xEF`
		NN * 256 + 00
		to
		NN * 256 + 127
		where
		1 ≤ NN ≤ 239

Function numbers in the range `0xNN80` to `0xNNFF` (where `0x01` ≤ NN ≤ `0xEF` and NN designates a Category) have been assigned by the TEE to proprietary functions. Please see the TEE supplier's documentation to interpret these values.	`0xNN80`
	to
	`0xNNFF`
	where
	`0x01` ≤ NN ≤ `0xEF`
		NN * 256 + 128
		to
		NN * 256 + 255
		where
		1 ≤ NN ≤ 239

Function numbers `0xF000` and higher have been assigned by the TEE to functions within proprietary categories. Please see the TEE supplier's documentation to interpret these values.	≥ `0xF000`	≥ 61440

5 Debug Log Message (DLM) API and Protocol

5.1 Overview

The proposed system provides a simple service based protocol to deliver Debug Log Messages (DLMs) from the TA to a Client Application.

When implementing this specification document, provision of the DLM protocol is optional and provision of the DLM API is conditional. If the DLM protocol is implemented, then provision of the DLM API is mandatory.

DLMs are generated by the TA using a `printf()`-like logging function. This function is limited in its capability set but provides sufficient capability to inform the developer of program flow and variable state.

This function also allows the Monitoring Application to class the debug message into one of `MAX_FILTERS` groups for context. Typical contexts might be:

- TA session init
- TA session shutdown
- TA Client received commands
- TA crypto activity

This context classification may then be used by the debug client to filter the message context. The classification may be defined as required by a particular TA designer. The only limitation is the overall number of contexts, `MAX_FILTERS`.

`MAX_FILTERS` is implementation dependent but is typically dependent on the processor data size, using one bit of the 32-bit word to act as one filter context. Typically, on a 32-bit processor `MAX_FILTERS` shall be 32 and hence allow up to 32 different filter contexts to be identified and chosen between.

There are a number of components to the DLM capability.

- The API to allow the TA to transmit DLMs
- The protocol to allow a client to receive such DLMs
    - The monitoring client may be in the REE, TEE, or potentially remotely connected through a UART or the like.
    - TEE and REE implementations of the monitoring client shall use the proposed interface via the TEE Client API.
    - Remote connections such as those through a UART shall use a proprietary interface that is not specified here.
- Properties that allow:
    - The TA manager to specify under what circumstances debug messages are to be exposed
    - The TEE manager to specify under what circumstances debug messages are to be exposed

Figure 5-1 shows a DLM session where the monitoring client is implemented in an (optional) REE software component.

Figure 5-1: DLM Debugging with REE Client

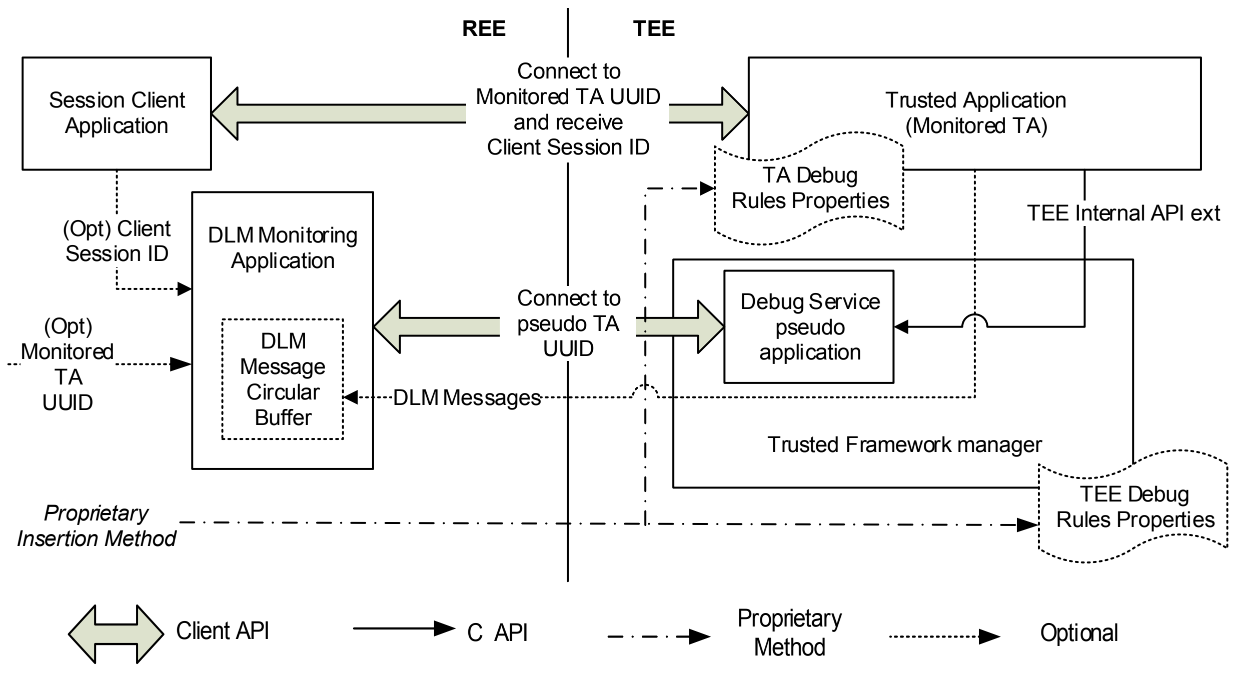
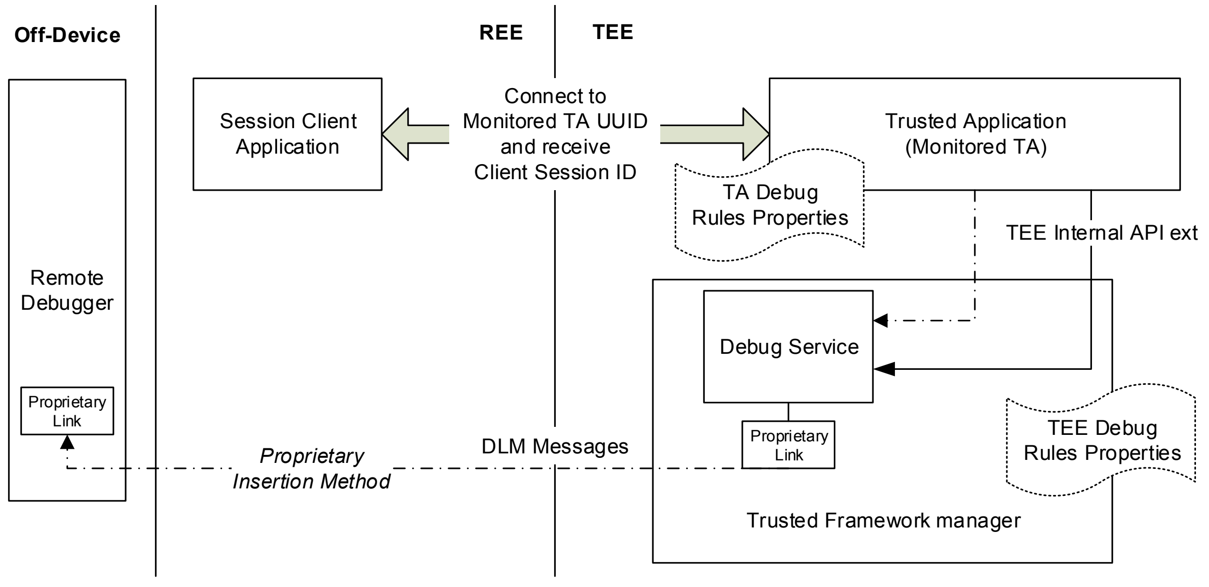

REE
TEE
Session Client Application
Connect to Monitored TA UUID and receive Client Session ID
Trusted Application (Monitored TA)
TA Debug Rules Properties
TEE Internal API ext
(Opt) Client Session ID
DLM Monitoring Application
DLM Message Circular Buffer
(Opt) Monitored TA UUID
Connect to pseudo TA UUID
Debug Service pseudo application
DLM Messages
Trusted Framework manager
TEE Debug Rules Properties
Proprietary Insertion Method
Client API
C API
Proprietary Method
Optional

Figure 5-2 shows a DLM session where the REE is not involved in monitoring. This may be implemented through the TEE responding to DLM commands from the TA, and providing DLM output through a serial port.

Figure 5-2: DLM Debugging with Off-device Monitoring Client


Off-Device
REE
TEE
Remote Debugger
Proprietary Link
Session Client Application
Connect to Monitored TA UUID and receive Client Session ID
Trusted Application (Monitored TA)
TA Debug Rules Properties
TEE Internal API ext
Debug Service
Proprietary Link
DLM Messages
Proprietary Insertion Method
TEE Debug Rules Properties
Trusted Framework manager

5.2 DLM Service pTA Availability

A DLM service shall only respond to a client TA with debug information on a device with suitable debug authorization. Debug authorization shall be determined by a combination of the TA specific and TEE specific Debug Rules Properties.

Such properties may be changed through one or more of:

- Proprietary means
- TA management updating a TA Download Package with different properties
- TEE management updating the TEE with different properties

All three of these methods shall be as cryptographically robust as the cryptographic protection applied to other TEE resource protection such as the TEE Internal API Trusted Storage (see Chapter 5 of [TEE Internal API]).

The connection to the DLM Service pTA shall always be cancellable (see section 2.1.4 of [TEE Internal API]).

5.3 Client Side DLM Protocol

This TEE Client API command protocol is designed to connect a Client Application (in either REE or TEE) to a debug message stream being provided by a TA.

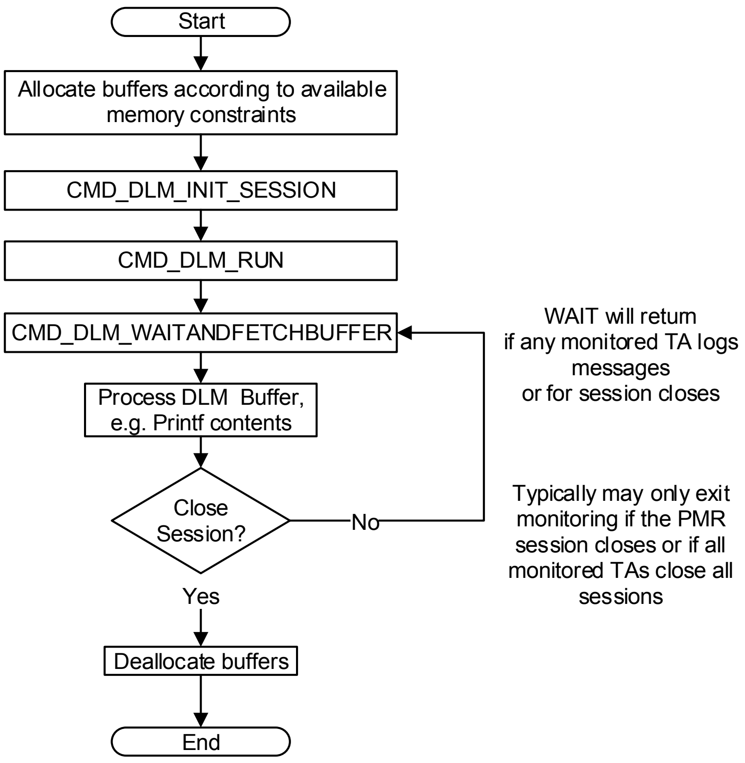
If there is no equivalent to the Client Application executing on the device, then a proprietary protocol shall be used to provide similar functionality for an off device client.

The DLM service for a Monitored TA is created asynchronously with regard to the connection to the Monitored TA.

A session is initially opened with a predefined pTA that provides DLM services inside the TEE (see section 2.2.1, Pseudo-TAs Implementing the Debug APIs). This DLM service session routes the DLM calls from the TA being debugged to the debug client.

The connection to the DLM Service pTA shall always be cancellable (see section 2.1.3 of [TEE Internal API]).

Figure 5-3: Simple Flow Diagram of PMR Monitoring


Start
Allocate buffers according to available memory constraints
CMD_DLM_INIT_SESSION
CMD_DLM_RUN
CMD_DLM_WAITANDFETCHBUFFER
WAIT will return if any monitored TA logs messages or for session closes
Process DLM Buffer, e.g. Printf contents
Close Session?
No
Typically may only exit monitoring if the PMR session closes or if all monitored TAs close all sessions
Yes
Deallocate buffers
End

5.3.1 Header Files

There are two header files for the TEE Debug Specification DLM API.

One applies to the client interface and one applies to the Trusted Application interface.

5.3.1.1 Client Interface Header

The client interface header file for the TEE Debug Specification DLM API must have the name "`tee_client_DLM_api.h`".

```
#include "tee_client_DLM_api.h"
```

This header will include the following definition:

```
#ifndef GPD_TEE_TA_DEBUG_1_0_1
#define GPD_TEE_TA_DEBUG_1_0_1
#endif
```

5.3.1.2 TA Interface Header

The TA interface header file for the TEE Debug Specification DLM API must have the name "`tee_internal_DLM_api.h`".

```
#include "tee_internal_DLM_api.h"
```

This header will include the following definition:

```
#ifndef GPD_TEE_TA_DEBUG_1_0_1
#define GPD_TEE_TA_DEBUG_1_0_1
#endif
```

5.3.2 Constants

This section describes constants used by the DLM session commands and related APIs.

5.3.2.1 Return Codes

Table 5-1 lists return codes used by the DLM session commands and related APIs.

Table 5-1: API Return Codes – DLM and Related APIs

Constant Names and Aliases		Value
`TEE_SUCCESS`	`TEEC_SUCCESS`	`0x00000000`
`DLM_NO_SESSION`	(used only by `TEE_DLMPrintf`, for which the lack of an available debug session does not indicate an error)	`0x00252001`
`DLM_EARLY_CLOSE`		`0x00252002`
`DLM_DATA_LOST`		`0x00252003`
`ERR_DLM_MESSAGE_BUFFER_SMALL`		`0xF0252001`
`ERR_DLM_BAD_UUID`		`0xF0252002`
`ERR_DLM_NO_SESSION`		`0xF0252003`

5.3.2.2 DLM Service UUID

A debug service shall be connected in the same manner as any other service offered by a TA.

For each application wishing to access debug output, a session open request (`TEE_OpenTASession` or `TEEC_OpenSession` as appropriate) is made to the specific UUID assigned to the DLM service.

The DLM service is identified by the following UUID.

```
static const TEEC_UUID DLMservice =
{
0x6D4006CA, 0xBC1B, 0x429C,
{0x92, 0x20, 0x9D, 0x54, 0x2C, 0xFD, 0x76, 0x79 }
};
```

See section 5.2.2 of [TEE Client API] or section 4.9 of [TEE Internal API] for more details on connecting to a service provided by a TA.

5.3.2.3 DLM Command Values

Table 5-2 lists the command values defined by the DLM interface.

Table 5-2: DLM Command Values

Constant Names and Aliases	Value
`CMD_DLM_INIT_SESSION`	`0x00000004`
`CMD_DLM_FLUSH`	`0x00000005`
`CMD_DLM_HALT`	`0x00000006`
`CMD_DLM_RUN`	`0x00000007`
`CMD_DLM_BLOCKINGMODE`	`0x00000008`
`CMD_DLM_WAITANDFETCHBUFFER`	`0x00000009`

5.3.3 DLM Session Commands

This section describes the handling of the DLM session commands.

5.3.3.1 `CMD_DLM_INIT_SESSION`

Description

This command initializes a DLM service.

It uses the `DLM_State` structure to inform the DLM service of the DLM source that should be connected to this session.

An initialized service shall be halted (see section 5.3.3.3) and have no filters set.

Subsequent calls to this command in a particular DLM service session shall replace the parameters provided by earlier calls, become halted, and have no filters set. The latest provided `DLM_State` structure shall always define the DLM source to be connected to this session. The latest `DLM_MessageBuffer` shall always define the current message buffer location and properties.

Command Details

- `commandID`: 4
- `params[2]`
  - `[in] DLM_State`

    `DLM_State` is described in section 5.3.4.

    The `DLM_State` provides the DLM session with the TA UUID and the client session ID.
  - `[inout] DLM_MessageBuffer`

    `DLM_MessageBuffer` is described in section 5.3.5.2.

    This structure shall be used to share buffer status information during the lifetime of the DLM session.

Return Code

- `ERR_DLM_MESSAGE_BUFFER_SMALL`

  Message buffer is too small.

  For minimum sizes, see section 5.3.5.3.
- `ERR_DLM_BAD_UUID`

  TA UUID is not recognized.
- `ERR_DLM_NO_SESSION`

  TA Session ID is not recognized.
- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.
- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.3.2 `CMD_DLM_FLUSH`

Description

This command clears any pending DLMs in the system. It is not a requirement for the system to internally buffer unread DLMs but if a particular design chooses to do so then this command shall flush pending messages.

If there is no active session or data to flush, then the command shall return `TEEC_SUCCESS.`

Command Details

- `commandID`: 5
- `params[0]`

  No parameters are used for this command.

Return Code

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.

- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.3.3 `CMD_DLM_HALT`

Description

This command informs the DLM service interface in the TEE to halt the Monitored TA(s) thread(s) at the next call to a TEE DLM function. When a running TA being logged calls a subsequent DLM function, that function shall block until a `CMD_DLM_RUN` is received from the DLM service client(s).

This command is provided to allow limited execution control such that a Monitoring Application may act on received DLMs or record them through a slow interface.

Command Details

- `commandID`: 6
- `params[0]`

  No parameters are used for this command.

Return Code

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.

- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.3.4 `CMD_DLM_RUN`

Description

This command informs the DLM service interface in the TEE to resume the Monitored TA thread at the next normal opportunity inside the Trusted OS.

This command is provided to allow limited execution control such that a Monitoring Application may act on received DLMs or record them through a slow interface.

Note that this command does not override the effects of `CMD_DLM_BLOCKINGMODE`.

Command Details

- `commandID`: 7
- `params[0]`

  No parameters are used for this command.

Return Code

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.

- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.3.5 `CMD_DLM_BLOCKINGMODE`

Description

This command informs the DLM service interface in the TEE to block if one of the receiving message buffers is full.

This command is provided to allow limited DLM flow control such that a Monitoring Application may act on received DLMs or record them through a slow interface.

Command Details

- `commandID`: 8
- `params[1]`
  - `[in]` Boolean
    - `true` – Block if the buffer is full.
    - `false` – Data may be lost if the buffer is full.

Return Code

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.

- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.3.6 `CMD_DLM_WAITANDFETCHBUFFER`

Description

This command requests a buffer update from the DLM service interface in the TEE.

This command shall wait until the next message is placed into the buffer before returning and thereby provides an alternative to polling the buffer state for updates.

Command Details

- `commandID`: 9
- `params[0]`

  None

Return Code

- `TEEC_ERROR_CANCEL`

  The command was cancelled by a `TEEC_RequestCancellation` on this session.
- `TEEC_SUCCESS`

  The command was completed successfully.

5.3.4 Client Side `DLM_State` Structure

```
typedef struct
{
    /* fixed parameters */
    [in] TEEC_UUID                  monitoredTA;
    [in] TEEC_UUID                  monitoredSession;

} DLM_State;
```

Description of `DLM_State` Structure Fields

- `monitoredTA`
  - UUID of the Monitored TA
  - This value is optional and may be the nil UUID to indicate an empty value.
    - If the nil UUID is specified, then all debuggable sessions on the monitored TEE shall report DLM output to this debug session and the `monitoredSession` value is ignored.
- `monitoredSession`
  - The Monitored TA to Session Client Application, session identifier
  - This value is ignored if this value or `monitoredTA` is the nil UUID.
  - If not the nil UUID:
    - This value contains a session unique identifier for the connection between the Session Client Application and the Monitored TA.
    - This value shall be the same as returned by the function `TEE_GetPropertyAsUUID` (see [TEE Internal API]) when the property name `gpd.ta.session.ID` is presented by the Monitored TA.
    - For Multi Instance TA, this value shall identify a unique instance and only Panics in that instance of the TA shall be reported to this DLM session.
    - For Single Instance TA, this value shall identify the active requesting session to the TA instance, but a Panic of ANY session of the TA shall be reported to this DLM session.

The `monitoredTA` UUID shall be a constant for a specific Monitored TA and may be provided to the DLM monitoring Client Application in a number of ways. Injection of UUID into the DLM monitoring Client Application is out of scope for this document.

To restrict debug to a specific session, the Client Application shall communicate `monitoredSession` with the DLM Monitoring Client Application. Such communication is out of scope of this document.

In both cases see Figure 2-1 on page 12.

5.3.5 DLM Buffer Structure and Handling

Because the threads driving the activity of the TA are potentially asynchronous relative to the DLM client, the DLM message buffer should be managed in an asynchronous manner.

The actual size of the data part of the buffer is provided by the client.

5.3.5.1 DLM Message Filter Type

`DLMfilter` is a 32-bit field used to indicate and filter one or more of 32 channels of logging data.

```
typedef uint32_t DLMfilter;
```

5.3.5.2 DLM Message Buffer Structure

```
typedef struct
{
    [in]     uint32_t storeSize;              /* set during initialization */
    [in]     DLMfilter    logMask;            /* set by client */
    [in]     atomic_t readIndex;           /* set by client */
    [out]    atomic_t writeIndex;          /* set by DLM service */
    [out]    char     ringBuffer[storeSize]; /* buffer contents set by DLM service */
} DLM_MessageBuffer
```

There is no particular enforcement of packing requirements and as such this may be implementation specific.

Description of `DLM_MessageBuffer` Structure Fields

- `storeSize`

  Storage buffer size in bytes

- `logMask`

  This value is used to determine whether a given authorized message shall be logged to the `ringBuffer`.

  Given a message with a filter value of `MsgFilter`, then:

  If `((logMask & MsgFilter) != 0)`, then the message shall be stored in the `ringBuffer`.

- `readIndex`

  Next unread character

  This index is managed by the client to indicate progress in consuming the message buffer.

  Read index `0` references the first element of the `ringBuffer`.

- `writeIndex`

  Last added character

  This index is managed by the DLM service to indicate progress in filling the message buffer.

  Write index `0` references the first element of the `ringBuffer`.

- `ringBuffer[]`

  Circular buffer containing DLM service transferred text

5.3.5.3 Circular Buffer Usage

The `DLM_MessageBuffer.ringBuffer` is managed as a round robin ring character buffer with a “keep one slot open” concept.

TEE characters are UTF-8 and the DLM service shall only place whole characters into the buffer.

The minimum circular buffer size value is 8 bytes. Note that if the minimum buffer is used, this only allows one full (4 byte) UTF-8 character.

Note: Read and write operations to the buffer and from buffer control fields in `DLM_MessageBuffer` should be assumed to be asynchronous.

5.3.6 DLM Code Example: Using the TEE DLM Client Protocol

The example in Annex B is a function in a Monitoring Application reading from the circular buffer.

5.4 Internal DLM API

This API is designed to provide simple printf()-like style debugging for standard TA development.

If the DLM protocol in section 5.3 is implemented, then the DLM API in this section shall be implemented.

The debug log message returned to the REE or other debugger shall contain the following information:

- A TA instance source identifier
- A time stamp
- The current `markValue` (see section 4.1)
- A string containing the data logged

5.4.1 `TEE_DLMServiceOpen`

```
TEE_Result TEE_DLMServiceOpen();
```

Description

This function opens a session from this Monitored TA to the DLM service.

Only one outgoing service exists per Monitored TA instance.

If the session is already open, then the service remains open in the same session and no Panic occurs.

Once open, if the Monitored TA emits any DLM and the TEE and TA Debug Rules Properties are correct, such DLM data shall be received by any listening Client Applications.

Note that if no DLM listeners are attached to the DLM service then the DLM service pTA may choose whether to buffer or disregard any DLM.

Parameters

None

Return Code

- `ERR_DLM_NO_SESSION`

  No DLM service available.
- `TEE_SUCCESS`

  Successful open

5.4.2 TEE_DLMServiceClose

```
void TEE_DLMServiceClose();
```

Description

This function requests closure of the DLM service session from this Monitored TA.

To actually close the DLM service session for this Monitored TA, for each call to `TEE_DLMServiceOpen` there must be a corresponding call to `TEE_DLMServiceClose`. Additional calls to `TEE_DLMServiceClose` shall be ignored.

Closing the DLM service session for the Monitored TA shall NOT result in a monitoring client connection to the DLM service becoming closed, however it shall receive no further DLMs from this TA until the session is re-opened.

If the Monitored TA emits any DLM messages when no DLM service session is open from the Monitored TA, then they shall NOT be received by any listening Client Applications.

Parameters

None

Return Value

None

5.4.3 TEE_DLMServiceStatus

```
void TEE_DLMServiceStatus(
    [out]    bool* dataDropped,
    [out]    bool* fullClient,
    [out]    bool* blockOnNext,
    [out]    bool* blockOnFull,
    [out]    bool* dropOnFull,
    [out]    bool* dedicatedToTA,
    [out]    bool* dedicatedToSession,
    [out]    uint32_t* buffFree
);
```

Description

This function allows the TA to enquire as to the state of the communication channel through the DLM service.

Note that as many clients may listen to one DLM service, the state is returned in terms of worst case situations.

This function is useful where the channel is being fed into an output device and that output device may block or drop data.

Parameters

The following parameters shall return `false` unless the stated condition is met:

- `dataDropped`

  `true` if at least one client has disregarding data since last `TEE_DLMServiceStatus` or `TEE_DLMServiceOpen`, whichever is later.

- `fullClient`

  `true` if at least one client DLM buffer is full.
- `blockOnNext`

  `true` if at least one client shall cause `DLM printf()` functions to block on next debug call.
- `blockOnFull`

  `true` if at least one client shall cause `DLM printf()` functions to block if buffer full.
- `dropOnFull`

  `true` if at least one client shall cause `DLM printf()` functions to drop data if buffer full.
- `dedicatedToTA`

  `true` if client is listening to just this TA.
- `dedicatedToSession`

  `true` if client is listening to just this session.

Other parameters:

- `buffFree`

  Number of bytes left in fullest buffer (Note that due to use of UTF-8 this does not relate directly to number of character spaces free.)

Return Value

None

5.4.4 TEE Internal API Extension – Debug Log Functions

5.4.4.1 Property Indicating DLM Installed on the System

A TA may enquire of the device as to whether the DLM capability is available.

Table 5-3: TEE DLM Availability Property

Property Name	Type	Description
`gpd.tee.debug.DLM_available`	`bool`	If `true`, then the DLM extension has been installed on this device. If `false` or not present, then the DLM extension has not been installed on this device. Note that just because the DLM extension is installed does not mean that debug log messages shall be transferred through the system, as they may be ignored due to other Debug Rules Properties of TAs and the TEE. This property is a *Modifiable Debug Rules Property* (see section 6.3*)*.

5.4.4.2 `TEE_DLMPrintf`

```
TEE_Result TEE_DLMPrintf(
    [in]         DLMfilter        filter,
    [instring]   const char*      format,
    [in]         ...
);
```

This function uses `printf()`-like formatting to pass data to any attached debug client.

If no debug clients meet the DLM filter requirement, then the format and the variable arguments shall not be interpreted. This exclusion is to specifically preclude errors of variable arguments interpretation occurring in production software containing `TEE_DLMPrintf()`.

Description

This API is designed to provide simple `printf()`-like style debugging for standard TA development.

The capabilities of this function are based on the `fprintf` as defined in [C99] (see section 7.19.6.1) with the limitations and extensions described in the following Output Format section.

Parameters

- `DLMfilter filter`

  A `uint32_t` value that shall log to any attached DLM session that meets the required authorization and where `((LogMask & filter) != 0)` is `true`. `LogMask` is a DLM session buffer specific value (see section 5.3.5).

  - This parameter gives the TA designer a limited amount of control over which debug messages shall be transferred through the interface. The designer may, for example, assign different `filter` values to messages originating during the initialization, processing, or closedown phases of the TA code.

• `char* format`:	The output format string containing `printf()` format structure
• `var parameters`:	Specified by the `printf()` format structure

Output Format

Flag

The function shall provide all the *flag* capabilities found in the [C99] version of `fprintf()`.

Field Width

The function shall provide all the *field width* capabilities found in the [C99] version of `fprintf()`.

Precision

The function shall provide all the *precision* capabilities found in the [C99] version of `fprintf()`.

Length Modifier

~~As the TEE only supports UTF-8 character encoding, it is assumed that all strings and characters shall be in such formatting. As such, ~~The presence ~~or absence ~~of the "l" sub-specifier in relation to character (%c) or string (%s) shall have no effect.

Conversion Specifiers

Table 5-4 lists the minimum recognized `printf()` format conversion specifier capabilities. If there is a conflict between Table 5-4 and those in [C99], then [C99] has priority except with reference to support for wide character formats.

Table 5-4: Valid Conversion Specifiers

Conversion Specifier	Description	Comments
%s	Null-terminated UTF-8 String	• The argument shall be a pointer to a start of a character in a Null-terminated string encoded in UTF-8. ~~• If an 'l' length modifier is not present, characters from the array are written up to (but not including) the terminating `NULL` character.~~ • A partial multibyte character shall be ignored.
%B	BigInt as OctetString	• The argument provided shall be a pointer to a **TEE_BigInt**. • The output from this shall be equivalent to the contents of the output buffer of a call to `TEE_BigIntConvertToOctetString()`. (See section 8.6.2. of [TEE Internal API].) • It is unaffected by the 'l' length modifier. • If the converter inside the `printf()` function is unable to malloc sufficient memory, then the `TEE_DLMPrintf()` call shall return `DLM_DATA_LOST`.
%d	int32_t as Decimal	• The ~~**int**~~ **int32_t** argument is converted to signed decimal in the style *[−]dddd*. • While no precision may be specified, an equivalent default precision is 1. The precision specifies the minimum number of digits to appear; if the value being converted may be represented in fewer digits, it is expanded with leading zeros.
%x	~~int32_t~~ uint32_t as hexadecimal	• The ~~**unsigned int**~~ **uint32_t** argument is converted to unsigned hexadecimal notation in the style *dddd*; the letters **abcdef** are used for **x** conversion. • While no precision may be specified, an equivalent default precision is 16. The precision specifies the minimum number of digits to appear; if the value being converted may be represented in fewer digits, it is expanded with leading zeros.
%c	~~utf8~~ uint32_t UNICODE character	• The **uint32_t** argument is a printable 32-bit UNICODE character in the Unicode range 00-7F, and any other values will be ignored.~~The uint32_t argument is converted to a UTF-8 character, and the resulting character, if complete, is written.~~ ~~• A partial multibyte character shall be ignored and no output occurs.~~ • ~~UTF-8 characters longer than 4 bytes are ignored and no output occurs.~~
%%	Converter escape	• A % character is written. No argument is converted. The complete conversion specification shall be %%.

Output

The value output to the DLM buffer shall be the result of the above `printf()` function(s) pre-pended with:

- The system time in ~~microseconds~~ milliseconds

  As this is for debug functionality, not security, this time stamp may originate from the REE.

- The TA UUID and the Session ID that identifies the current session of the Monitored TA
- The current `markValue` (see section 4.1)

Each output shall be terminated by a `NULL` character.

Pseudocode example of output generation

```
/*
    Example of TEE_DLMPrintf implementation in TEE Trusted Framework
    (c) 2013-2016 GlobalPlatform

    This example code does not claim to be the most efficient method of
    Implementing this function, but does cover the basic actions required.

    */

/* UUID_STR_SIZE
 Assumes format of "nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn"
 **/
#define UUID_STR_SIZE ((size_t)(32+4+1))

TEE_Result TEE_DLMPrintf(DLMfilter     filter,
                           const char*  format,
                           va_list        vargs)
{
    char          *RawOutput; // formatted output string based on format and vargs
    size_t        RawSize;

    char          *PrependedOutput;     // formatted output string
                                   // with additional header data
    size_t        PrepSize;

    TEE_Time SystemTime;

    TEE_Result    DLMPrintfRet;
    int      ErrRet;

    uint32_t NumberOfListeners;

    /* NoOfListeners() function checks to see if any monitoring application has
     * its filters set such that it shall receive this DLM
     */
    NumberOfListeners = NoOfListeners(filter);

    if (NumberOfListeners == 0)
        return TEE_SUCCESS;                // message sent to all listeners

    TEE_GetREETime(&SystemTime);

    /* MallocBuffers() example function
```

```
     * calculates the buffers required from the format and va_list arg
     * Points RawOutput and PrependedOutput to appropriate allocated buffers
     * Sets RawZise and PrepSize to the number of bytes in each allocated buffer
     */
    DLMPrintfRet = MallocBuffers(&RawOutput,
                 &RawSize,
                 &PrependedOutput,
                 &PrepSize,
                 format ,vargs// this information is used
                          // to determine buffer sizes
                );

    if (DLMPrintfRet != TEE_SUCCESS)
        return DLMPrintfRet;

    /* build the output requested
     * this function differs from standard vsnprintf in
     * the format parameter handling
     */
    ErrRet = TEE_vsnprintf(RawOutput, format, vargs);
    if (ErrRet < 0 || ErrRet > RawSize)
    {
        FreeBuffers( &RawOutput, &PrependedOutput);
        return DLM_DATA_LOST;
    }

    TEE_PropSetHandle enumHandle;
    TEE_Result res;

    res = TEE_AllocatePropertyEnumerator( &enumHandle );
    if (res != TEE_SUCCESS)
        ErrRet = snprintf(PrependedOutput,
                   PrepSize,
                   "Out Of Memory");
    else
    {
    // this is bad practice as really the STR size allocation should
// be based on a null request

        char myAppID[UUID_STR_SIZE]="";
        size_t RefSize = UUID_STR_SIZE;
        TEE_GetPropertyAsString(enumHandle,
                   "gpd.ta.appID",
                   myAppID,
                   &RefSize);

        char mySessionID[UUID_STR_SIZE]="";
        TEE_GetPropertyAsString(enumHandle,
                   "gpd.ta.session.ID",
                   mySessionID,
                   &RefSize);

        /* build the output requested */
        ErrRet = snprintf(PrependedOutput,
                   PrepSize,
                   "%d.%03d, %36s, %36s, %s",
                   SystemTime.seconds,
                   SystemTime.millis,
                   myAppID,
                        mySessionID,
                        RawOutput);
    }

    if (ErrRet < 0 || ErrRet > PrepSize)
    {
        FreeBuffers( &RawOutput, &PrependedOutput);
        return DLM_DATA_LOST;
    }

    /* The DLM implementers DLMOutput() function transfers the DLM to waiting
    * Monitoring Applications or external debuggers as required by the
    * "filter" settings **/
    DLMPrintfRet = DLMOutput(filter, PrependedOutput);

    // de-allocate the memory pointed to by the two buffers
// and set the pointers to NULL
    FreeBuffers( &RawOutput, &PrependedOutput);
    return DLMPrintfRet;

}
```

Return Code

- `DLM_NO_SESSION`

  No debug session available.

- `DLM_EARLY_CLOSE`

  Debug session closed during transmission.

- `DLM_DATA_LOST`

  Debug message data lost. Typically, this may be due to buffer overrun. It is not guaranteed that this return value shall occur for lost data as it depends on the buffering regime implemented by the DLM service.

- `TEE_SUCCESS`

  Successful completion.

Panic

If the parameters are incorrect, then the TA shall Panic.

5.4.5 DLM Code Example: Using the TEE DLM API in a TA

For an example of DLM usage in a TA, see Annex D.

6 Debug Authorization

6.1 Authorization Principles

A debug authorization system needs to consider various requirements for the actors involved and the information flow.

A number of different debug methods are provided:

- DLM is useful to the developer.
- PMR is useful to both developer and conformance evaluator.

  Others may be added in the future or via proprietary extensions and so the method of authorization must provide sufficient flexibility to cope with such future debug methods.

There is granularity of authorization target:

- A specific TA, or all TAs on a TEE
- Beyond this, the ability to create a TA that is debugged only on a particular device, a class of devices, or all devices, depends on the TA Management system capabilities.

The removal, or nullification, of Debug Rules Properties shall indicate that debugging is disabled.

- Removal of the TEE Debug Rules Property shall block ALL TEE debug on the device.
- Removal of a TA Debug Rules Property from a particular TA shall block ALL TEE debug when that TA is active or prevent that TA from becoming active if a debug session is in progress.

Further, there is a layered approach to the actual authorization.

- While a TA may authorize debug, a TEE may impose additional system wide restrictions that may further reduce debug output on that TA.
- When one TA allows debug, other TAs may wish to not function, or to block that authorization while they are active.

6.1.1 Debug Authorization Methodology

Debug is authorized by attaching appropriate Debug Rules Properties to both the TA and TEE.

The combination of these two sets of debug rules determines whether the debug service shall provide debug output, and what that output shall contain, under a specific set of conditions.

TA Debug Rules Properties shall be attached to the TA as part of the TA's build process, or as part of the TA management process.

TEE Debug Rules Properties shall be injected through proprietary means, or as part of the TEE TA management process (see TEE Management Framework [TEE Mgmt Fwk]).

6.2 Authorization Debug Rules Properties

A TA may read its associated debug rules by using the property access functions to examine the TA and TEE Debug Rules Properties discussed in this section. (See section 4.4 of [TEE Internal API] for the definition of Property Access Functions.)

It is expected that the Debug Rules Properties information shall only be interrogated by the DLM or PMR service.

6.2.1 Debug Rules Properties

Each TA may have either, or both, of the first two properties in Table 6-1.

A TEE may have either, or both, of the last two properties in Table 6-1.

Table 6-1: Debug Rules Properties

Property	Definition
`GPD.TA.DBG_PMR.DATA_AVAILABLE`	See section 6.2.2.
`GPD.TA.DBG_DLM.DATA_AVAILABLE`	See section 6.2.3.
`GPD.TEE.DBG_PMR.DATA_AVAILABLE`	See section 6.2.2.
`GPD.TEE.DBG_DLM.DATA_AVAILABLE`	See section 6.2.3.

All the properties in Table 6-1 are modifiable Debug Rules Properties (see section 6.3) and are of type `int8_t`.

6.2.2 Available PMR Data – `DATA_AVAILABLE`

Debug Rules Property `GPD.TA.DBG_PMR.DATA_AVAILABLE` states what sort of PMR data shall be retrievable, according to the TA.

Debug Rules Property `GPD.TEE.DBG_PMR.DATA_AVAILABLE` states what sort of PMR data shall be retrievable, according to each TEE.

Table 6-2: DATA_AVAILABLE Valid PMR States

Name	Value	Description	Support Required
`TEE_BLOCKED`	-64	As a TA rule value, indicates that while the TA with this value has active sessions, PMR events of any TA in the TEE shall be ignored and not reported through the PMR service. As a TEE rule value, shall be considered the same as `BLOCKED`.	Mandatory
`BLOCKED` (default)	`0`	As a TA rule value, indicates that PMR events of this TA shall be ignored and not reported through the PMR service. As a TEE rule value, indicates that PMR events of any TA shall be ignored and not reported through the PMR service.	Mandatory

`CODE_ONLY`	32	Only the following data about the panicked TA shall be available:
		`specNumber`
		`functionNumber`
		`markValue`
		`panicReasonCode`
		(As defined in `PMR_MessageBuffer` (see section 3.2.4.3))
			Mandatory

`CODE_STATE`	64	This adds the panicked TA's state to the `CODE_ONLY` set of data available. If sufficient resources are available, the state data shall be placed at the location defined by `PMR_STATE_POINTER` in section 3.2.4.3.	Optional
`HEAP_STACK`	96	This adds the panicked TA's heap and stack to the `CODE_STATE` set of data available. If sufficient resources are available, the heap and stack data shall be placed at the locations defined by `PMR_HEAP_POINTER` and `PMR_STACK_POINTER` in section 3.2.4.3.	Optional
`ALL`	112	All data in the TEE is available to be reported. In systems where TA specific state could not be presented due to the method of sharing TA stacks and heap between trusted applications, in this state such data may be presented.	Optional

`ALL` is the weakest security value, proceeding as ordered in Table 6-2 to `TEE_BLOCKED`, the strongest.

Default reflects the state and value when the specified property is not present in a TA or TEE.

Notes on TA heap and stack reporting:

1. Enabling heap and stack reporting is dangerous in that potentially keys might be reported in the clear. It should only be done with development systems and it is advised that steps are taken to prevent this being accidentally enabled in other systems.
2. When option `ALL` is selected, it is acceptable (though undesirable) for the PMR to provide non TA specific heap, stack, or state data; i.e. it shall provide the whole TEE heap rather than a specific TA heap. When enabling option `ALL` access to the PMR data, information leakage of other TA heaps should be taken into account.

If any Debug Rules Property is changed, then any pending output shall not be sent to Monitoring Applications.

6.2.2.1 Required Support of `DATA_AVAILABLE` PMR States

A conforming implementation shall support the `TEE_BLOCKED`, `BLOCKED`, and `CODE_ONLY` values for `DATA_AVAILABLE`.

6.2.3 Available DLM Data – `DATA_AVAILABLE`

Debug Rules Property `GPD.TA.DBG_DLM.DATA_AVAILABLE` states what sort of DLM data can be retrieved, according to the TA.

Debug Rules Property `GPD.TEE.DBG_DLM.DATA_AVAILABLE` states what sort of DLM data can be retrieved, according to each TEE.

Table 6-3: `DATA_AVAILABLE` Valid DLM States

Name	Value	Description
`TEE_BLOCKED`	-64	As a TA rule value, indicates that while the TA with this value has active sessions, DLM events of any TA in the TEE shall be ignored and not reported through the DLM service. As a TEE rule value, shall be considered the same as `BLOCKED`.
`BLOCKED` (default)	`0`	No DLM is available. As a TA rule value, indicates that while this TA may open a DLM session and use `TEE_DLMPrintf()`, there shall be no output. As a TEE rule value, indicates that while any TA may open a DLM session and use `TEE_DLMPrintf()`, there shall be no output.
`ALL`	64	All DLM data is available.

`ALL` is the weakest security value, proceeding as ordered in Table 6-3 to `TEE_BLOCKED`, the strongest.

Default reflects the state and value when no specified parameter is present.

If any Debug Rules Property is changed, then any pending output shall not be sent to Monitoring Applications.

6.2.4 Required Support of `DATA_AVAILABLE` DLM States

A conforming implementation which supports the DLM feature shall support the `TEE_BLOCKED`, `BLOCKED`, and `ALL` values for `DATA_AVAILABLE`.

6.2.5 Properties of Underlying Data Structure Format and Security Strength

The structure details of the underlying data structure of the Monitored TA bound DLM and PMR Debug Rules Properties, and the associated key management, shall be implemented and managed as a bound part of a particular TA or TEE instance on a device. It is bound to the TA or TEE by the same cryptographic strength as that used to install the TA and protect the TEE and its assets.

Further details are outside of the scope of this document.

6.3 Debug Unlock Property

An additional property is defined to enable a device to be locked such that no further changes shall be made to TEE debug rules.

Table 6-4: TEE Debug Unlock Property

Property Name	Type	Description
`gpd.tee.debug.debug_unlock_properties`	`bool`	If `true`, then modifiable Debug Rules Properties may be changed. If `false` or not present, then modifiable Debug Rules Properties shall not be changeable. This property is itself a *modifiable Debug Rules Property* and so setting this `false` prevents resetting it to `true`.

7 TEE Management Framework Control of Debug Rules Properties

It is implementation specific whether the device uses proprietary methods to change Debug Rules Properties or uses extensions of the TEE Management Framework [TEE Mgmt Fwk].

This section describes how the TEE Debug system is managed using the TEE Management Framework.

7.1 Changing TEE Debug Rules Properties

TEE Debug Rules Properties shall be modified using the TEE Management Framework command STORE TEE PROPERTY mechanism. This command shall only allow changes to modifiable Debug Rules Properties.

7.2 Changing TA Debug Rules Properties

TA properties are metadata associated with the TA.

To change TA Debug Rules Properties, the implementer may use the TEE Management Framework to remove and then install the TA again with revised properties or to update an existing TA.

Annex A Panicked Function Identification

Every TEE internal function shall be assigned a specification number (see section 4.3) and function number (see section 4.4), even if it is not expected to Panic.

In Table A-1:

- Function numbers `0x00nn` are assigned to Panics due to TA execution errors, rather than associated with a particular function.
- Function Numbers `0x01nn` are assigned to functions defined in this document.

For more information about assignment of function numbers, see section 4.4.

Table A-1: Function Identification Values

Category	Function	Function Number in hexadecimal	Function Number in decimal
TA Panics	TA divide by zero	`0x0001`	1
	TA stack overflow	`0x0002`	2
	TA memory access error	`0x0003`	3
	Undefined	`0x0004`	4
Debug Support	`TEE_DebugSetMarker`	`0x0101`	257
	`TEE_DLMPrintf`	`0x0102`	258
	`TEE_DLMServiceOpen`	`0x0103`	259
	`TEE_DLMServiceClose`	`0x0104`	260
	`TEE_DLMServiceStatus`	`0x0105`	261
Other Specification-defined Panics	See individual TEE specifications.	`0xNN00` to `0xNN7F` where `0x01 ≤ NN ≤ 0xEF`	NN * 256 + 00 to NN * 256 + 127 where 1 ≤ NN ≤ 239
Proprietary Panics	Assigned by the TEE to proprietary functions.	`0xNN80` to `0xNNFF` where `0x01 ≤ NN ≤ 0xEF`	NN * 256 + 128 to NN * 256 + 255 where 1 ≤ NN ≤ 239
	Assigned by the TEE to functions within proprietary categories.	≥ `0xF000`	≥ 61440

Annex B DLM Client Example

The following example code is informative.

No guarantee is made as to its quality or correctness.

```
/*
    Example usage of the DLM protocol over Client API
    (c) 2013-2016 GlobalPlatform

    This is an example of the code executed by the
    Monitoring Application to setup and retrieve output
    from the DLM service.

    */

#include <stdio.h>
#include <string.h>

#include "TEE_Client_API.h"
#include "tee_client_DLM_api.h"

// returns true if transfer buffer is empty
#define isDLM_Empty(buffer)       \
    ((atomic_get((buffer)->readIndex)) == (atomic_get((buffer)->writeIndex)))

// The size in bytes of the largest possible UTF 8 Character (RFC 3629)
// This may vary depending upon implementation
#define MAX_UTF8 4

// Code provided by the Monitoring application
// that outputs one UTF8 character over STDOUT
void YourCode_cputsUTF8(uint8_t  *utf8char);

// Code provided by the Monitoring application
// that copies one UTF8 character and increments the read pointer
void YourCode_CopyUTF8char(DLM_MessageBuffer * DLM_Buffer, uint8_t  *utf8char);


/* This is an example of DLM Client code
*/
TEEC_Result DLM_Client_Example(
    TEEC_UUID        monitoredTA,
    TEEC_UUID        monitoredSession
    )
{
    /* Allocate TEE Client structures on the stack. */
    TEEC_Context      context;
    TEEC_Session      session;
    TEEC_Operation    operation;

    TEEC_Result       result;

    // *************************

    /* ========================================================================
    [1] Connect to TEE
    ======================================================================= */

    result = TEEC_InitializeContext( NULL, &context);

    if (result != TEEC_SUCCESS)
    {
        goto cleanup1; // failed to initialise context
    }

    /* ======================================================================
    [2] Open session with TEE application
    ======================================================================= */
    /* Open a Session with the TEE application. */
    static const TEEC_UUID DLMservice =
    {
        0x6D4006CA, 0xBC1B, 0x429C,
        {0x92, 0x20, 0x9D, 0x54, 0x2C, 0xFD, 0x76, 0x79 }
    };


    result = TEEC_OpenSession(
        &context,
        &session,
        &DLMservice,
        TEEC_LOGIN_USER,
        NULL,     /* No connection data needed for TEEC_LOGIN_USER. */
        NULL,     /* No payload, and do not want cancellation. */
        NULL);    /* No origin of potential errors required. */
    if (result != TEEC_SUCCESS)
    {
        goto cleanup2; //failed to open session
    }

    // ***********************

    /* ======================================================================
    [3] Initialize the Shared Memory buffer
    ======================================================================= */

    // allocate state
    DLM_State DLM_InstanceState;
    TEEC_SharedMemory DLM_StateRef;

    DLM_StateRef.size = sizeof(DLM_State);
    DLM_StateRef.buffer = (void *) &DLM_InstanceState;
    DLM_StateRef.flags = TEEC_MEM_INPUT| TEEC_MEM_OUTPUT;
    result = TEEC_RegisterSharedMemory(&context , &DLM_StateRef);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup3; //failed to register shared memory
    }

    // configure the state to define Monitored Application
    // copy the two UUIDs
    memcpy(  (void*) &DLM_InstanceState.monitoredTA, \
             (void*) &monitoredTA, sizeof(TEEC_UUID));

    memcpy(  (void*) &DLM_InstanceState.monitoredSession, \
             (void*) &monitoredSession, sizeof(TEEC_UUID));
```

```
// allocate a Circular buffer space
TEEC_SharedMemory DLM_BufferRef;

/* reserve a 10Kbyte +Header buffer for this service */
DLM_BufferRef.size = sizeof (DLM_MessageBuffer) + 10 * 1024;

DLM_BufferRef.flags = TEEC_MEM_INPUT | TEEC_MEM_OUTPUT;

result = TEEC_AllocateSharedMemory(&context , &DLM_BufferRef);
if (result != TEEC_SUCCESS)
{
    goto cleanup4; // Failed to allocate shared memory
}

// configure the buffer
DLM_MessageBuffer * DLM_Buffer;

DLM_Buffer = (DLM_MessageBuffer * ) DLM_BufferRef.buffer;
DLM_Buffer->storeSize = DLM_BufferRef.size - sizeof(DLM_MessageBuffer);

// allow all messages
DLM_Buffer->logMask       = 0xFFFFFFFF;

// point read and write at the start of the buffer
atomic_set(DLM_Buffer->readIndex, 0);
atomic_set(DLM_Buffer->writeIndex, 0);

// ***********************

/* ==========================================================================
[4] Query DLM service
========================================================================== */

/* Define parameter usage */
operation.paramTypes = TEEC_PARAM_TYPES(
    TEEC_MEMREF_WHOLE,
    TEEC_MEMREF_WHOLE,
    TEEC_MEMREF_WHOLE,
    TEEC_NONE);

// State
operation.params[1].value.a = 0;

// Message Buffer
operation.params[2].memref.parent = &DLM_BufferRef;
operation.params[2].memref.offset = 0;
operation.params[2].memref.size   = DLM_BufferRef.size ;

/* Provide the circular buffer space and state*/
result = TEEC_InvokeCommand(
    &session,
    CMD_DLM_INIT_SESSION,
    &operation,
    NULL);
if (result != TEEC_SUCCESS)
{
    goto cleanup5; // Provide context command failed
}
```

```
    /* Process logging received from the monitored TA(s) */
    do
        {
            result = TEEC_InvokeCommand(
                &session,
                CMD_DLM_WAITANDFETCHBUFFER,
                NULL,
                NULL);
            if (result != TEEC_SUCCESS)
            {   /* our session just died so need to tidy up */
                goto cleanup6;
            }
            else
            {
            // Process the current buffer until it is empty
            do  // fetch ALL the unread characters and output one at a time
                {
                uint8_t utf8char[MAX_UTF8]; // one UTF8 character
                YourCode_CopyUTF8char(
                    (DLM_MessageBuffer *) DLM_BufferRef.buffer,
                    utf8char );
                if (utf8char != 0)
                    YourCode_cputsUTF8(utf8char);
                }
                while(!isDLM_Empty(DLM_Buffer));

        }
        }
        while (result == TEEC_SUCCESS) ;

    // ***********************


    /* ========================================================================
    [7] Tidyup resources
    ======================================================================= */
cleanup6:  // Success OR CMD_DLM_WAITANDFETCHBUFFER failed
        /* No additional cleanup operations are required here*/
cleanup5: //  Failed CMD_DLM_INIT_SESSION command
    TEEC_ReleaseSharedMemory(&DLM_BufferRef);
cleanup4: // Failed to allocate shared memory
    TEEC_ReleaseSharedMemory(&DLM_StateRef);
cleanup3: //failed to register shared memory
    TEEC_CloseSession(&session);
cleanup2: // failed to open session
    TEEC_FinalizeContext(&context);
cleanup1:; // failed to initialise context

}       // end of DLM_Client_Example()
```

Annex C PMR Client Example

The following example code is informative.

The provided code example will work only if the endianness and processor architecture are the same in the Client Application and the TEE.

No guarantee is made as to its quality or correctness.

```
/*
    Example usage of the PMR protocol over Client API
    (c) 2013-2016 GlobalPlatform

    This is an example of the code executed by the
    Monitoring Application to setup and retrieve
    output from the PMR service.

    */

#include <stdio.h>
#include <string.h>

#include "TEE_Client_API.h"
#include "TEE_internal_api.h"    // included because TEE internal values may be exposed
#include "tee_client_PMR_api.h"

/* HexDisplay
 A function that output "bytes" of data pointed to by "StartPtr"
 Provided by Monitoring Application implementer
 baseAddr   address of startPtr in TA memory map
 startPtr   address of buffer in local EE memory map
 bytes      length of buffer in bytes
 **/
void HexDisplay(TEE_RefAddress *baseAddr, uint8_t * startPtr, int bytes);

/* UUIDtoString
 A function that converts a TEEC_UUID to a string
 Provided by Monitoring Application implementer
 string     destination for output string
 UUID       UUID to be converted
 **/
void UUIDtoString(char * string, TEEC_UUID * UUID);

/* UUID_STR_SIZE
 This assumes format of "nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn"
 being generated by UUIDtoString() above.
 **/
#define UUID_STR_SIZE (36+1)

static const TEEC_UUID NilUUID =
{
0x00000000, 0x0000, 0x0000,
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
};


/* This function contains an example of retrieving data associated with a Panic.
```

```
 In this case we shall limit processing to retrieving the PMR event and
 reporting available information through printf.
 **/
int FetchAndProcessPMR (
    TEEC_Context * PMRContext,
    TEEC_Session * PMRSession,
    PMR_State *instanceState)
{
    TEEC_Result      result;
    TEEC_Operation       operation;

    // report the information available from the PMR_State data
    printf("Panic Report : requested parameters~~/n~~\n");
    printf("UUID :%s\n",instanceState->monitoredTA);

    if (memcmp(&instanceState->monitoredSession, &NilUUID, sizeof(TEEC_UUID) )!=0)
        printf("TEE Internal Session Handle  :%8x\n",instanceState
                 ->monitoredSession);
    else
        printf("No Session ID available\n");


    // allocate the buffer space
    int totalBufferSize = sizeof (PMR_MessageBuffer)
        + instanceState->heapSize
        + instanceState->stackSize
        + instanceState->stateSize;

    TEEC_SharedMemory PMR_BufferRef;

    PMR_BufferRef.size = totalBufferSize;    // how much to allocate
    PMR_BufferRef.flags = TEEC_MEM_OUTPUT;  // usage

    result = TEEC_AllocateSharedMemory(PMRContext , &PMR_BufferRef);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup1;        // Allocate failed
    }

    // Message Buffer
    operation.params[0].memref.parent = &PMR_BufferRef;
    operation.params[0].memref.offset = 0;
    operation.params[0].memref.size   = PMR_BufferRef.size ;

    // Examine the state and process the buffers

    result = TEEC_InvokeCommand(
        PMRSession,
        CMD_PMR_FETCHPMR,
        &operation,
        NULL);

    if (result != TEEC_SUCCESS)
    {
        goto cleanup2;    // invoke failed
    }

    // resolve pointer to base of PMR_MessageBuffer
    PMR_MessageBuffer  * Buffer = (PMR_MessageBuffer  *) PMR_BufferRef.buffer;
    char sourceUUIDStr[UUID_STR_SIZE];
    char sessionIDStr[UUID_STR_SIZE];

    UUIDtoString(sourceUUIDStr, &Buffer->sourceUUID);
    printf("Panic UUID :%s\n",sourceUUIDStr);

    UUIDtoString(sessionIDStr, &Buffer->sessionID);
    printf("Panic Session :%s\n",sessionIDStr, Buffer->sessionID);

    printf("Panic Function : Source specification : 0x%4x %d\n", \
        Buffer->specNumber,Buffer->specNumber);
    printf("Panic Function : Function in Spec: 0x%4x %d\n",\
        Buffer->functionNumber,Buffer->functionNumber);
    printf("Panic Reason   : Reason Code  :%d\n",Buffer->panicReasonCode);
    printf("MARK VALUE state: %d (0x%8X)\n",Buffer->markValue,Buffer->markValue);

    printf("State data ~~/n~~\n");

    // Create a dummy address for use with state data
    TEE_RefAddress ZeroAddress;
    ZeroAddress.Addr32bit.Hi =0;
    ZeroAddress.Addr32bit.Lo =0;

    if (Buffer->stateSize)
        HexDisplay(
        &ZeroAddress // it is not considered useful
            // to provide a base address for the state data
        ,PMR_STATE_POINTER(Buffer)
        , Buffer->stateSize);
    else
        printf("No state data available~~/n~~\n");


    printf("Stack data ~~/n~~\n");
    if (Buffer->stackSize)
    {
        HexDisplay(
            &Buffer->stackRefAddress,
            PMR_STACK_POINTER(Buffer),
            Buffer->stackSize);

        if (Buffer->completeStack)
            printf("\n This is a complete stack.");
        else
            printf("\n This is only a partial stack.");
    }
    else
        printf("No state data available~~/n~~\n");

    printf("Heap data ~~/n~~\n");
    if (Buffer->heapSize)
        HexDisplay(
            &Buffer->heapRefAddress,
            PMR_HEAP_POINTER(Buffer) ,
            Buffer->heapSize);
    else
        printf("No state data available~~/n~~\n");

cleanup2:            // Command invoke failed
```

```
    TEEC_ReleaseSharedMemory(&PMR_BufferRef);

cleanup1: ;      // Allocate failed

}


/* This is an example of PMR Client code
*/
TEEC_Result PMR_Client_Example(
    TEEC_UUID    *    monitoredTA,
    TEEC_UUID    *   monitoredSession
    )
{
    /* Allocate TEE Client structures on the stack. */
    TEEC_Context      context;
    TEEC_Session      session;
    TEEC_Operation    operation;

    TEEC_Result       result;

    /* =====================================================================
    [1] Connect to TEE
    ===================================================================== */

    result = TEEC_InitializeContext( NULL, &context);

    if (result != TEEC_SUCCESS)
    {
        goto cleanup1; // Init Failed
    }

    /* =====================================================================
    [2] Open session with TEE application
    ===================================================================== */
    /* Open a Session with the TEE application. */
    static const TEEC_UUID PMRservice =
    {
        0x09A193B3, 0x688A, 0x476F,
        {0x88, 0xB3, 0x6A, 0xD7, 0xd9, 0xDA, 0x93, 0x22}
    };

    result = TEEC_OpenSession(
        &context,
        &session,
        &PMRservice,
        TEEC_LOGIN_USER,
        NULL,     /* No connection data needed for TEEC_LOGIN_USER. */
        NULL,     /* No payload, and do not want cancellation. */
        NULL);    /* No origin of potential errors required. */
    if (result != TEEC_SUCCESS)
    {
        goto cleanup2;    // open session failed
    }

    /* =====================================================================
    [3] Initialize the Shared Memory buffer
    ===================================================================== */
    // allocate state
    PMR_State PMR_InstanceState;
    TEEC_SharedMemory PMR_StateRef;

    PMR_StateRef.size = sizeof(PMR_State);
    PMR_StateRef.buffer = (void *) &PMR_InstanceState;
    PMR_StateRef.flags = TEEC_MEM_INPUT| TEEC_MEM_OUTPUT;
    result = TEEC_RegisterSharedMemory(&context , &PMR_StateRef);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup3;        // failed allocate shared memory PMR_StateRef
    }

    // configure the state to define Monitored TA
    memcpy(&PMR_InstanceState.monitoredTA, monitoredTA, sizeof(TEEC_UUID));
    memcpy(&PMR_InstanceState.monitoredSession, monitoredSession, sizeof(TEEC_UUID));

    // allocate a Target buffer space
    TEEC_SharedMemory PMR_BufferRef;

    // reserve a 10Kbyte +Header buffer for this service
    PMR_BufferRef.size = sizeof (PMR_MessageBuffer) + 10 * 1024;

    PMR_BufferRef.flags = TEEC_MEM_OUTPUT;

    result = TEEC_AllocateSharedMemory(&context , &PMR_BufferRef);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup4;        // Failed allocate shared memory PMR_Buffer
    }

    /* unlike DLM, no buffer configuration is required as the buffer is used
    purely to retrieve data from the PMR service in the TEE */


    /* ========================================================================
    [4] Query PMR service
    ======================================================================== */

    /* Define parameter usage */
    operation.paramTypes = TEEC_PARAM_TYPES(
        TEEC_MEMREF_WHOLE,
        TEEC_MEMREF_WHOLE,
        TEEC_MEMREF_WHOLE,
        TEEC_NONE);

    // State
    operation.params[1].value.a = 0;

    // Message Buffer
    operation.params[2].memref.parent = &PMR_BufferRef;
    operation.params[2].memref.offset = 0;
    operation.params[2].memref.size   = PMR_BufferRef.size ;

    /* Provide the circular buffer space and state*/
    result = TEEC_InvokeCommand(
        &session,
        CMD_PMR_INIT_SESSION,
        &operation,
        NULL);
    if (result != TEEC_SUCCESS)
    {
        goto cleanup5;    // failed CMD_PMR_INIT_SESSION
    }

    /* Process logging received from the monitored TA(s) */
    do
    {
        result = TEEC_InvokeCommand(
            &session,
            CMD_PMR_WAIT,
            NULL,
            NULL);

        if (result == PMR_MONITORED_TA_PANIC)
            result = FetchAndProcessPMR(&context,&session, &PMR_InstanceState);

    }
    while (result == PMR_MONITORED_TA_PANIC) ;

    /* ========================================================================
    [5] Tidyup resources
    ======================================================================== */

cleanup5:        // failed CMD_PMR_INIT_SESSION
    TEEC_ReleaseSharedMemory(&PMR_BufferRef);
cleanup4:        // failed allocate shared memory PMR_Buffer
    TEEC_ReleaseSharedMemory(&PMR_StateRef);
cleanup3:        // failed allocate shared memory PMR_StateRef
    TEEC_CloseSession(&session);
cleanup2:        // open session failed
    TEEC_FinalizeContext(&context);
cleanup1:        // Init Failed
    ;

}       // end of PMR_Client_Example()
```

Annex D DLM TA Usage Example

The following example code is informative.

No guarantee is made as to its quality or correctness.

```
/*
    Example usage of the DLM interface use in a TA
    (c) 2013-2016 GlobalPlatform

    This is an example of the code executed by the
    Trusted Application wishing to emit
    DLM debug.

    */

#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "tee_internal_api.h"
#include "tee_internal_DLM_api.h"

#define DLM_FILTER_INIT       0x0001
#define DLM_FILTER_CMD    0x0002
#define DLM_FILTER_CORE   0x0003

//commands
#define EXAMPLE_Do_1FOO       0x01
#define EXAMPLE_Do_2FOO       0x02

// Global 32bit variable
uint32_tDebug_marker;

/* This function represents the general processing of the TA
 */
void foo()
{
    int v,u;

    TEE_DLMPrintf(DLM_FILTER_CORE,"Starting processing in foo()");

    for(v=0;v<10;v++)
        {
        TEE_DLMPrintf(DLM_FILTER_CORE,"Let us try with v=%d",v);
        u=5/(v-5);   // this represents a code error and
                     // panics due to divide by zero when v=5
        }

    TEE_DLMPrintf(DLM_FILTER_CORE,"Ending foo()");
}

/* This function is called by creating a TA instance
*/
TEE_Result TA_CreateEntryPoint()
{
    TEE_Result ret;
    Debug_marker = 0xFFFFDEAD;
    TEE_DebugSetMarker(&Debug_marker);



    ret = TEE_DLMServiceOpen();
    TEE_DLMPrintf(DLM_FILTER_INIT,"DLM Open");
    // Assign a variable to report state to debugger
    return (ret);
}

/* This function is called by destroying a TA instance
*/
void TA_DestroyEntryPoint( void )
{
    Debug_marker = 0x0000DEAD;

    TEE_DLMPrintf(DLM_FILTER_INIT,"DLM Closed");
    TEE_DLMServiceClose();
    return;
}

/* This function is called by destroying a TA instance
*/
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
    void*        sessionContext,
    uint32_t     commandID,
    uint32_t     paramTypes,
    TEE_Param    params[4] )
{
    switch (commandID) {
        case EXAMPLE_Do_2FOO:
            TEE_DLMPrintf(DLM_FILTER_CMD,"do 2 foo");
            foo();
            foo();
            break;
        case EXAMPLE_Do_1FOO:
            TEE_DLMPrintf(DLM_FILTER_CMD,"do 1 foo");
            foo();
            break;
        default:
            TEE_DLMPrintf(DLM_FILTER_CMD,"default ??");
            break;
    }

return TEE_SUCCESS    ;
}       // end of DLM TA example
```