DSP/BIOS™ LINK

MESSAGING COMPONENT

LNK 031 DES

Version 1.30

This page has been intentionally left blank.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

1 Introduction 7
1.1 Purpose & Scope 7
1.2 Terms & Abbreviations 7
1.3 References 7
1.4 Overview 7
2 Requirements 8
3 Assumptions 8
4 Constraints 8
5 High Level Design 10
5.1 Overview 10
5.2 DSP side 12
5.3 GPP side 15
6 Sequence Diagrams 19
6.1 Initialization 19
6.2 Finalization 21
6.3 MSGQ_open () 23
6.4 MSGQ_close () 24
6.5 MSGQ_locate () 25
6.6 LDRV_MSGQ_locateAsync 26
6.7 MSGQ_release () 27
6.8 MSGQ_alloc () 28
6.9 MSGQ_free () 29
6.10 MSGQ_put () 30
6.11 MSGQ_get () 31
6.12 MSGQ_setErrorHandler () 32
6.13 MSGQ_count () 33
7 API 34
7.1 Constants & Enumerations 34
7.2 Typedefs & Data Structures 53
7.3 API Definition 64
8 PMGR 82
8.1 API Definition 82
9 LDRV MSGQ 99
9.1 Typedefs & Data Structures 99
9.2 API Definition 112

TABLE OF FIGURES

Figure 1. Messaging in *DSPLINK*. 10
Figure 2. MSGQ and POOL component hierarchy 12
Figure 3. DSP-side component interaction diagram 13
Figure 4. GPP-side component interaction diagram....16
Figure 5. On the GPP: MSGQ initialization 19
Figure 6. On the GPP: MSGQ_transportOpen () control flow 20
Figure 7. On the GPP: MSGQ finalization....21
Figure 8. On the GPP: MSGQ_transportClose () control flow 22
Figure 9. On the GPP: MSGQ_open () control flow....23
Figure 10. On the GPP: MSGQ_close () control flow....24
Figure 11. On the GPP: MSGQ_locate () control flow 25
Figure 12. On the GPP: MSGQ_locateAsync () control flow 26
Figure 13. On the GPP: MSGQ_release () control flow 27
Figure 14. On the GPP: MSGQ_alloc () control flow 28
Figure 15. On the GPP: MSGQ_free () control flow 29
Figure 16. On the GPP: MSGQ_put () control flow....30
Figure 17. On the GPP: MSGQ_get () control flow 31
Figure 18. On the GPP: MSGQ_setErrorHandler () control flow 32
Figure 19. On the GPP: MSGQ_count () control flow....33

1 Introduction

1.1 Purpose & Scope

This document describes the design of messaging component for DSP/BIOS™ LINK.

The document is targeted at the development team of DSP/BIOS™ LINK.

1.2 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
MSGQ	Message Queue
Client	Refers to a process/ thread/ task in an operating system that uses DSP/BIOS™ LINK API. It is used to ensure that description is free from the specifics of 'unit of execution' for a particular OS.
☞	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.3 References

1.	LNK 012 DES	DSP/BIOS™ LINK Link Driver Version 1.12, dated AUG 24, 2004
2.	LNK 019 DES	DSP/BIOS™ LINK Shared Memory Processor Copy Link Driver Version 1.11, dated NOV 05, 2004
3.	LNK 041 DES	DSP/BIOS™ LINK Zero Copy Link Driver Version 0.65, dated OCT 29, 2004
4.	LNK 082 DES	DSP/BIOS™ LINK POOL Version 0.01, dated AUG 26, 2004

1.4 Overview

DSP/BIOS™ LINK is runtime software, analysis tools, and an associated porting kit that simplifies the development of embedded applications in which a general-purpose microprocessor (GPP) controls and communicates with a TI DSP. DSP/BIOS™ LINK provides control and communication paths between GPP OS threads and DSP/BIOS™ tasks, along with analysis instrumentation and tools.

The messaging component (MSGQ) provides logical connectivity between the GPP clients and DSP tasks. Unlike the data transfer channels where the client is waiting for data to arrive on a designated channel, the message transfer is completely

asynchronous. The messages may be used to intimate occurrence of an error, change in state of the system, a request based on user input, etc.

This document describes the various design alternatives to achieve the messaging functionality between GPP and DSP using DSP/BIOS™ LINK. It also gives an overview of the messaging component on the GPP and DSP-sides of *DSPLINK* and its interaction with the other components within *DSPLINK*. The document also gives a detailed design with sequence diagrams of the GPP-side MSGQ component. Detailed designs of GPP and DSP-side Message Queue Transport (MQT) components for different physical links can be found in the design document for the link drivers. For example, for designs of the shared memory MQTs, please refer to the DSP/BIOS™ LINK Shared Memory Processor Copy Link Driver design [Ref. 6] and DSP/BIOS™ LINK Zero Copy Link Driver design [Ref. 7].

On the GPP side, implementation shall utilize the services of the native OS.

On the DSP side, the implementation shall utilize the services of MSGQ module of DSP/BIOS.

2 Requirements

The basic requirements for the messaging component can be summarized as below:

R20	The messages shall be transferred at a higher priority than data channels when only one HW medium is available.
R21	Messages of fixed length and variable length shall be supported.
R22	Messaging shall work transparently over varied links between GPP & DSP.
R23	DSP/BIOS™ LINK shall support messaging with the MSGQ module.

The messaging component shall also comply with the following additional requirements:

1. The API exported by the messaging component shall be:
   - Common across different GPP operating systems
   - Similar to the API on DSP/BIOS
2. Message allocation must occur via the MSGQ component.
3. The API for sending messages must be deterministic and non-blocking.

3 Assumptions

- This document assumes that the reader is familiar with the design of the MSGQ component of DSP/BIOS™ [Ref. 5].
- The contents of the messages shall not be interpreted within the DSP/BIOS™ LINK layer.
- The messages shall not be split & joined on either sending or receiving end. User shall provide the maximum length of the message that can be transferred across GPP & DSP.

4 Constraints

The design of the messaging component in *DSPLINK* is constrained by the following:

- The DSP-side of the messaging component must match the interface of the MSGQ module in DSP/BIOS™.
- The ARM-side of the messaging component must be as similar to the DSP-side as possible. However, there may be some differences due to constraints imposed by the ARM-side OSes.

The user constraints are:

- The total message size must be greater than the size of the fixed message header. This includes the size of the complete user-defined message including the required fixed message header.
- Multiple threads/processes must not receive messages on the same MSGQ. Only a single thread/process owns the local MSGQ for receiving messages. However, multiple threads/processes may send messages to the same message queue.
- The remote MQT uses the default pool for allocating control messages required for communication with other processors. The number of control messages required depends on the frequency of usage of APIs requiring control messages, such as `MSGQ_locate ()`. The user must be aware of this usage of the pool resources by *DSPLINK*.
- The messages must have a fixed header as their first field. This header is used by the messaging component for including information required for transferring the message. The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user.
- The messages must be allocated and freed through APIs provided as part of the messaging component. Messages allocated through any other means (for example: standard OS calls) cannot be transferred using the *DSPLINK* messaging component.
- The message queue names must be unique over the complete system. This includes message queues created across all processors in the system.
- The default pool provided to the remote MQT must be opened by the user before any remote MSGQs are located, or the MQT is closed.

5 High Level Design

The basic unit of messaging from a client's perspective is a message queue. All the messages are sent to a message queue existing on the same processor or a different processor.

Each message queue shall be addressed through a unique name.

The messaging component can utilize any physical data links between GPP and DSP. This can be configured through the static configuration system.

The message queues are unidirectional. They are created on the receiving side. Senders locate the queue to which they wish to send messages. The queues may be distributed across several processors. This distribution is transparent to the users.

5.1 Overview

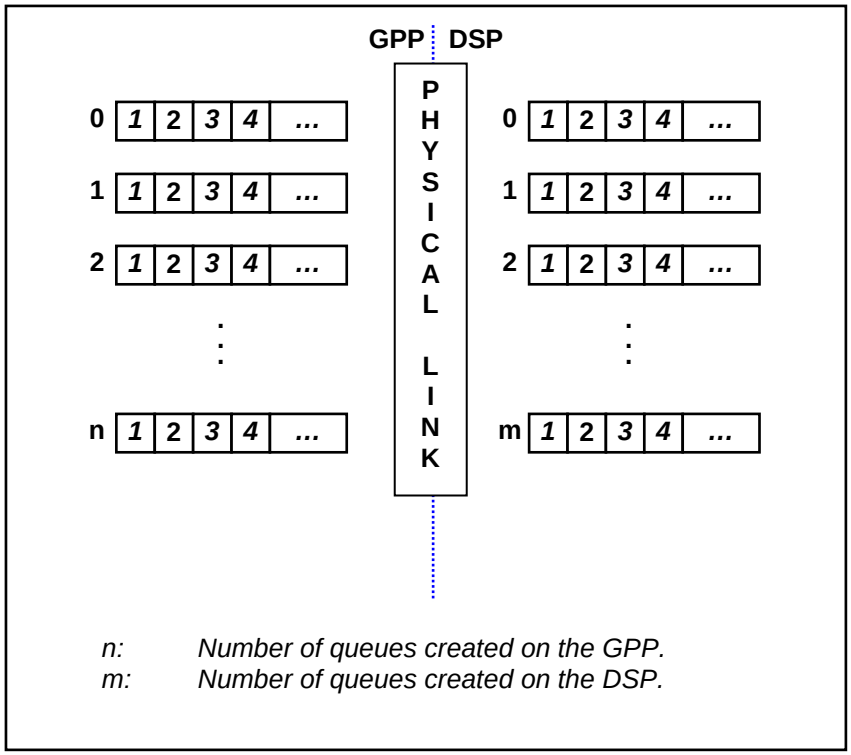

Figure 1. Messaging in *DSPLINK*.

Message

Variable sized messages can be sent using the *DSPLINK* messaging component.

The message must contain the fixed message header as the first element. This header is not modified by the user, and is used within *DSPLINK* for including information required for transferring the message. APIs are provided for accessing information in the header required by the user.

APIs provided by the messaging component are used for allocating and freeing the messages. Different pools may be specified for allocation of the messages, based on the requirement. Messages cannot be allocated on the stack or directly through the standard OS allocation and free functions.

Referencing a message queue

On both the GPP and DSP sides, a unique name is used for identifying a MSGQ. This name is unique over all processors in the system. When a message queue is opened or located, a unique handle to the message queue is returned to the user. This handle is used for all further accesses to the message queue. The unique message queue handle is a 32-bit value composed of two 16-bit values representing the processor ID and the message queue ID on the processor identified by the processor ID.

For example:

A 32-bit message queue handle `0xAAAABBBB` has the first 16-bits representing the processor ID (`0xAAAA`) and the next 16-bits representing the ID of the message queue on that processor (`0xBBBB`).

Initialization and finalization

Before using any of the messaging features, the user must initialize the MSGQ component.

On the DSP-side, once the MSGQ and POOL components are enabled through the DSP/BIOS™ static configuration, they get initialized as part of the DSP/BIOS™ boot-up and initialization process. For this, the user must define and initialize the special `MSGQ_config` and `POOL_config` structures within the application.

On the GPP-side, initialization of the MSGQ component involves initialization of the individual transports and pools. When the messaging services are no longer required, the user can finalize the individual transports and pools.

Creating and deleting a message queue

The message queue is created and deleted on the processor where the reader(s) shall be.

Sending a message

To send a message to a message queue, the user must first locate the message queue to ensure that the MSGQ exists on some processor in the system.

If the MSGQ location is successful, the user can send a message to it.

The API for sending the message is deterministic and non-blocking. However, the actual transfer of the message may not complete immediately. Especially in the case of remote MSGQs, the user must not assume that the message transfer over the physical link is complete when the API returns.

Receiving a message

For receiving a message on a particular message queue, the user can specify a timeout value to indicate the time for which the API must wait for the message to arrive, in case it is not already available. With a timeout of zero, the API returns immediately, and is non-blocking. If a message is available when the API is called, it is returned immediately, otherwise an error is returned.

Replying to a message

While sending a message, the user can choose to specify a source MSGQ for receiving reply messages. The receiver of the message may retrieve the source MSGQ, and use it for replying to the received message. This feature may be used for cases where an acknowledgement for reception of the message is desired.

5.2 DSP side

The DSP-side of the *DSPLINK* messaging component is based on the MSGQ model in DSP/BIOS™.

The MSGQ and POOL modules have a two-level architecture. The first level consists of the MSGQ API and POOL interface. The second level consists of different implementations of the Message Queue Transport (MQT) interface and POOL interface.

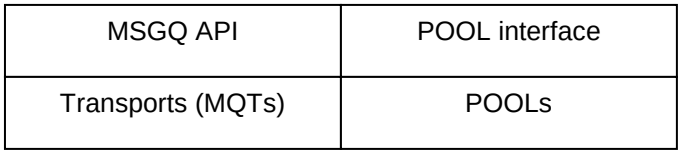

Figure 2. MSGQ and POOL component hierarchy

For further details, please refer to the MSGQ documentation [Ref. 5].

The *DSPLINK* messaging component shall implement an MQT for communication with the GPP. In addition, it shall also utilize a POOL for management of the message buffers.

5.2.1 Component interaction

The component interaction diagram gives an overview of the interaction of the various subcomponents involved in messaging. The component interaction shown is with reference to an example Processor Copy (PCPY) MQT and POOL implementation for the Shared Memory (SHM) link.

Note that the diagram does not show all the components for data transfer.

For details on the complete *DSPLINK* design, please refer to the DSP/BIOS™ LINK architecture document [Ref. 3] and the DSP/BIOS™ LINK Link Driver Design [Ref. 4].

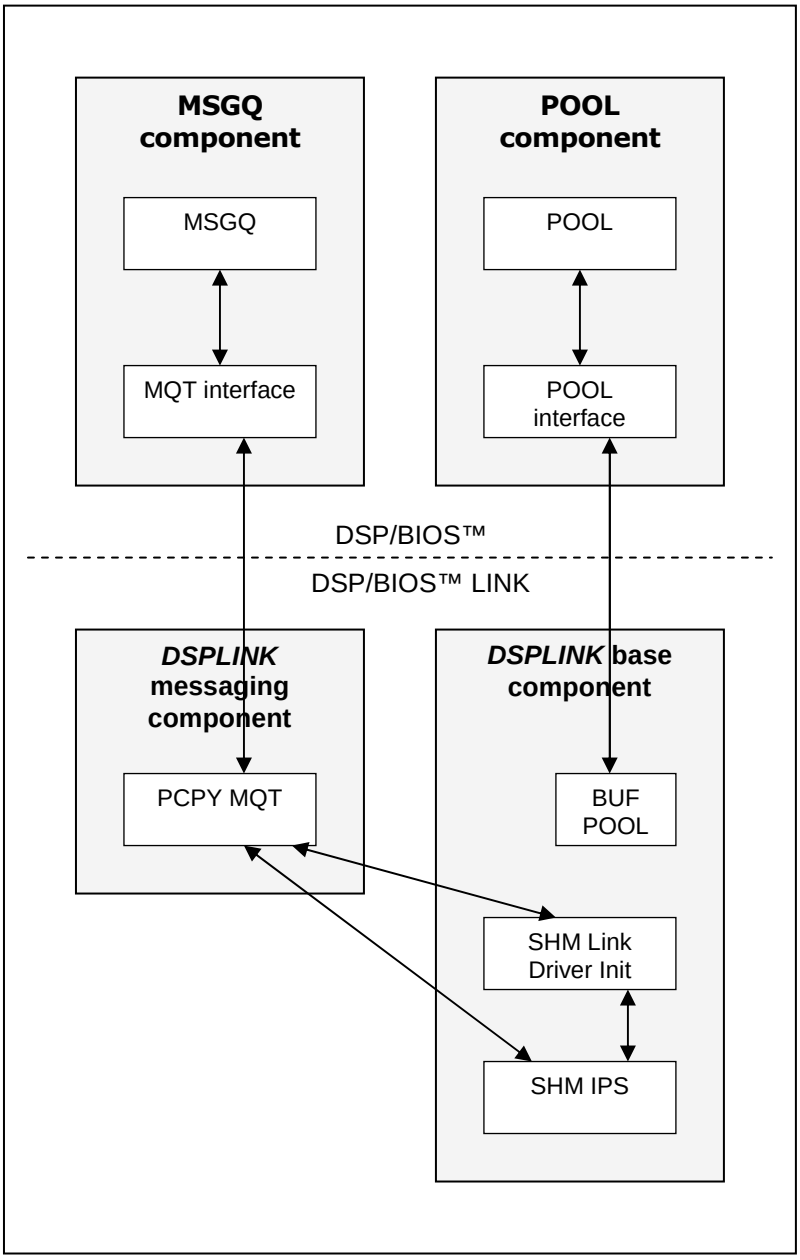

Figure 3. DSP-side component interaction diagram

5.2.2 Overview

The implementation of data transfer and messaging features shall be through completely different paths. Service common to both data transfer and messaging shall be part of the *DSPLINK* generic component.

The DSP-side messaging component shall be implemented as a separate library, utilizing the services of the generic *DSPLINK* component. In addition, the messaging functionality shall conform to the MSGQ interface of DSP/BIOS™.

Scalability for CHNL and MSGQ shall be provided through compile-time flags, which shall be set by the common configuration tool.

For more details on the *DSPLINK* driver design, please refer to the DSP/BIOS™ LINK Link Driver Design [Ref. 4].

This design allows the flexibility of an optimized and high-performance implementation of the MQT and data transfer protocol for a particular physical link. In this case, the MQT shall not be completely independent of the physical link. The implementation can however ensure that any common code between multiple MQTs is separated for code size reduction.

Transfer of messages shall be given higher priority within the IOM driver.

5.2.3 Details

The design of the DSP-side messaging component is specific to each physical link. This document gives an overview of the generic requirements to be met by any implementation of the MQT and POOL for messaging within *DSPLINK*. Details of specific MQT designs for physical links can be found in the design document for the link drivers.

MQT

The MQT shall implement the transport protocol for communication with its counterpart on the GPP.

The MQT must ensure the following:

- The MSGQs are independent of each other. No MSGQ shall be blocked due to an unclaimed message for another MSGQ.
- Messages from different senders, intended for different MSGQs, are sent through a common physical link to the DSP.
- The MQT function for sending a message is deterministic, and shall return immediately. However, actual transfer of the message to the GPP may complete later.
- Messages received from the GPP, intended for different MSGQs, are received from the physical link and forwarded to the appropriate MSGQs on the DSP.
- Messages of varying sizes are appropriately handled, with minimum wastage of memory.

POOL

The POOL must not allocate memory dynamically, since the functions for allocation and freeing of memory may be called from an HWI or SWI context.

5.3 GPP side

The GPP-side of the *DSPLINK* messaging component shall be parallel to the corresponding design on the DSP-side. The messaging API shall be similar to the one on the DSP-side, while incorporating restrictions imposed by the GPP-side OS.

The *DSPLINK* messaging component shall implement the MSGQ component, along with the specific MQTs for communication with the DSP. It shall utilize the POOL component as well as the specific POOLs required.

The messaging design shall be scalable to allow the users to scale out only the messaging component, only the channel component, or both the messaging and channel components.

5.3.1 Component interaction

The component interaction diagram gives an overview of the interaction of the various sub-components involved in messaging. The component interaction shown is with reference to an example Processor Copy (PCPY) MQT and POOL implementation for the Shared Memory (SHM) link.

Note that the diagram does not show all the components for processor and DSP control, as well as data transfer.

For details on the complete *DSPLINK* design, please refer to the DSP/BIOS™ LINK architecture document [Ref. 3] and the DSP/BIOS™ LINK Link Driver Design [Ref. 4].

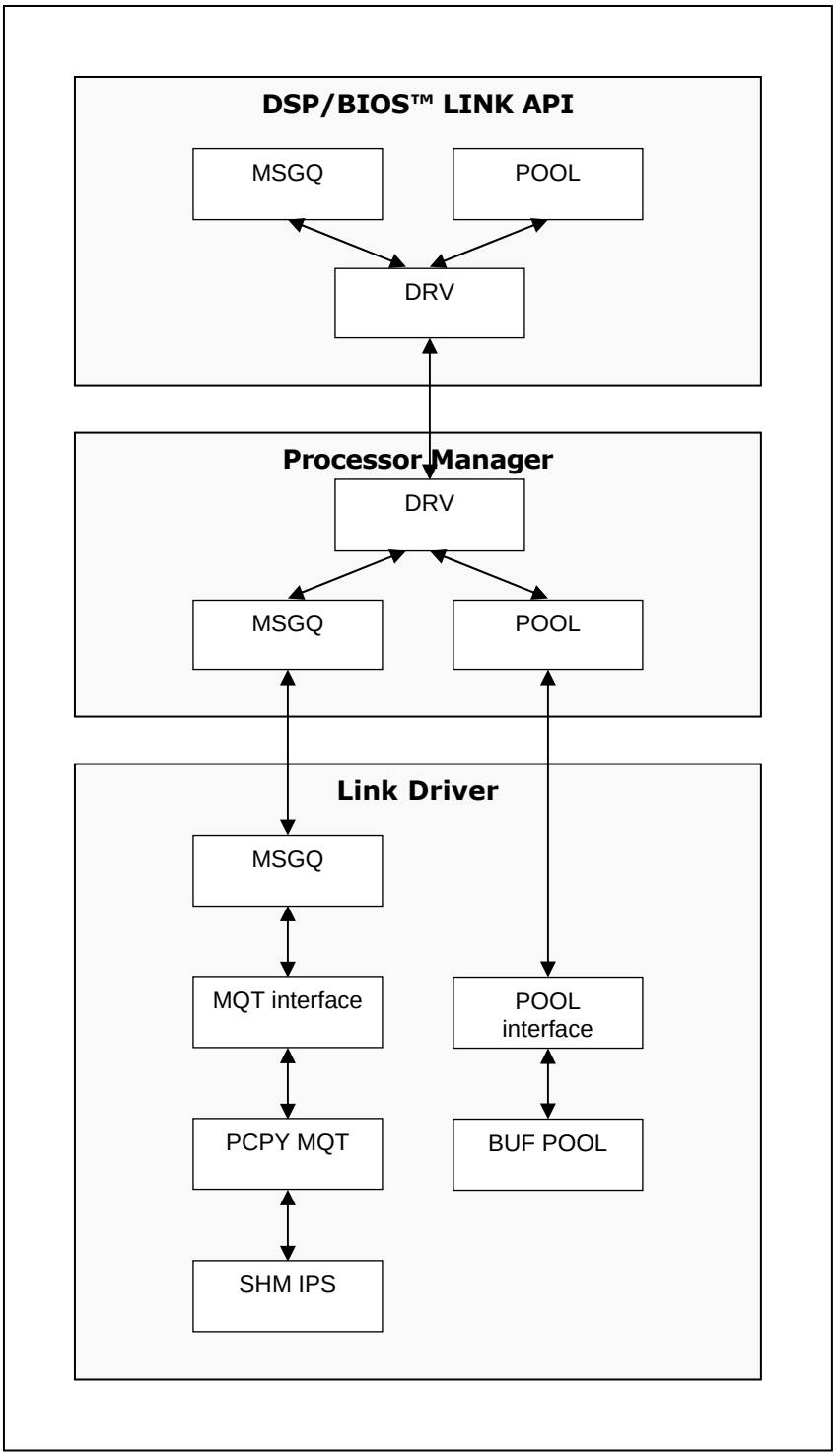

Figure 4. GPP-side component interaction diagram

5.3.2 Overview

The GPP-side messaging component design is spread across the API, PMGR and LDRV components.

An overview of the updates to each of these components is given below. These updates are detailed in later sections.

5.3.2.1 API

As part of the DSP/BIOS™ LINK API, additional APIs shall be provided to the user for utilizing the messaging feature. This includes APIs for:

- Component initialization/finalization
- Message Queue creation/deletion
- Message allocation/freeing
- Message sending/receiving
- Message Queue location/release/getting the source message queue handle

5.3.2.2 PMGR

The PMGR component shall be enhanced to support the messaging feature. The messaging sub-component within the PMGR component shall provide the counterpart to the corresponding messaging APIs.

The messaging PMGR sub-component shall utilize the services provided by the corresponding messaging sub-component within LDRV.

A scalability option shall allow the PMGR component to be scaled out of the *DSPLINK* implementation when the MSGQ-only driver is required.

5.3.2.3 LDRV

The messaging design that is specific to the link driver is part of the LDRV component.

This includes the following:

- Generic messaging protocol and local MSGQ management
- An implementation of an MQT (Message Queue Transport)

In addition, the following other *DSPLINK* components are utilized by the messaging component and shall be implemented for the specific physical link between the processors.

- Link-specific inter-processor signaling component (For example SHM IPS)
- POOL interface
- An implementation of a POOL

5.3.2.4 Others

The CFG sub-component shall be enhanced to include configuration information for the MSGQ component. This includes configuration of the different MQTs in the system.

5.3.3 Details

This document gives the detailed design of the MSGQ component.

The design of the POOL component is detailed in the DSP/BIOS™ LINK POOL design [Ref. 8]. In addition, the design of the MQT and IPS is specific to each physical link. Details of specific MQT designs for physical links can be found in the design documents for the link drivers.

MSGQ

The MSGQ component provides APIs for the various messaging actions similar to the ones provided on the DSP-side by the DSP/BIOS™ MSGQ component. The GPP-side MSGQ component spans the API, PMGR and LDRV layers. The API component provides parameter validation and a drop down into the PMGR layer, which provides the facility of ownership validation. The LDRV MSGQ layer contains the actual implementation of the MSGQ features, and also includes the MQT interface, which the specific MQT plugs into.

Configuration

The configuration shall contain dynamically configured information for the MSGQ component.

The GPP object shall contain information about the maximum number of local message queues in the system.

The configuration object shall contain information about the number of MQTs in the system.

MQTs shall be configured within the dynamic configuration. The MQT object in the configuration shall include all required information about the MQT, including interface table, the MMU entry (if any required) configured in the CFG within the MMU table referred to by the DSP that uses the MQT. In addition, there is provision for optional MQT-specific arguments to be provided by the user.

The link driver object in the CFG shall specify the MQT to be used for messaging communication with the DSP.

6 Sequence Diagrams

The following sequence diagrams show the control flow for a few of the important functions to be implemented within the *DSPLINK* messaging component.

The design of the DSP-side messaging component is specific to each physical link. This document does not give any sequence diagrams for the DSP-side MSGQ component. The sequence diagrams for specific physical links can be found in the design document for the link drivers.

This section gives the sequence diagrams for the GPP-side MSGQ component and its interaction with the MQT and POOL components.

6.1 Initialization

6.1.1 MSGQ

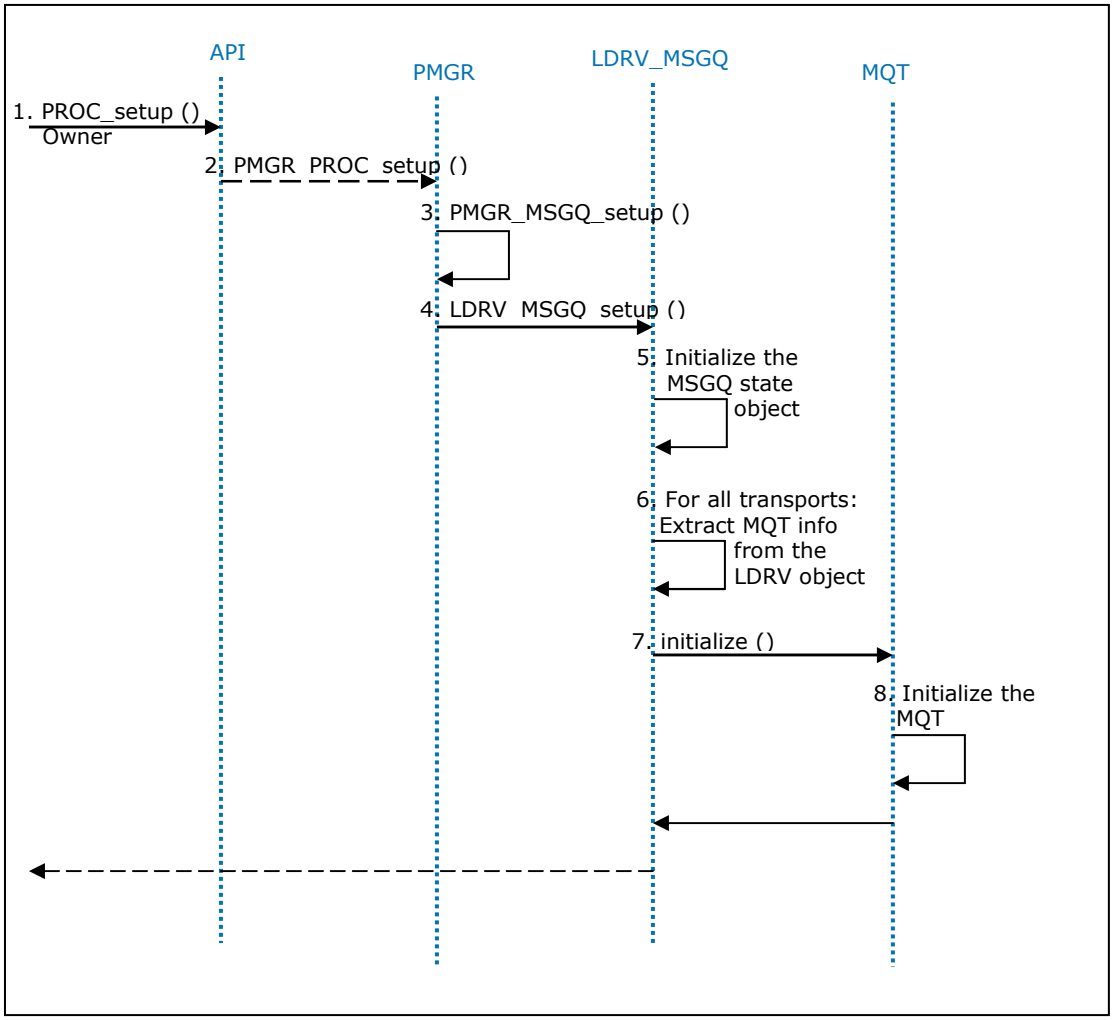

Figure 5. On the GPP: MSGQ initialization

6.1.2 Transport Open

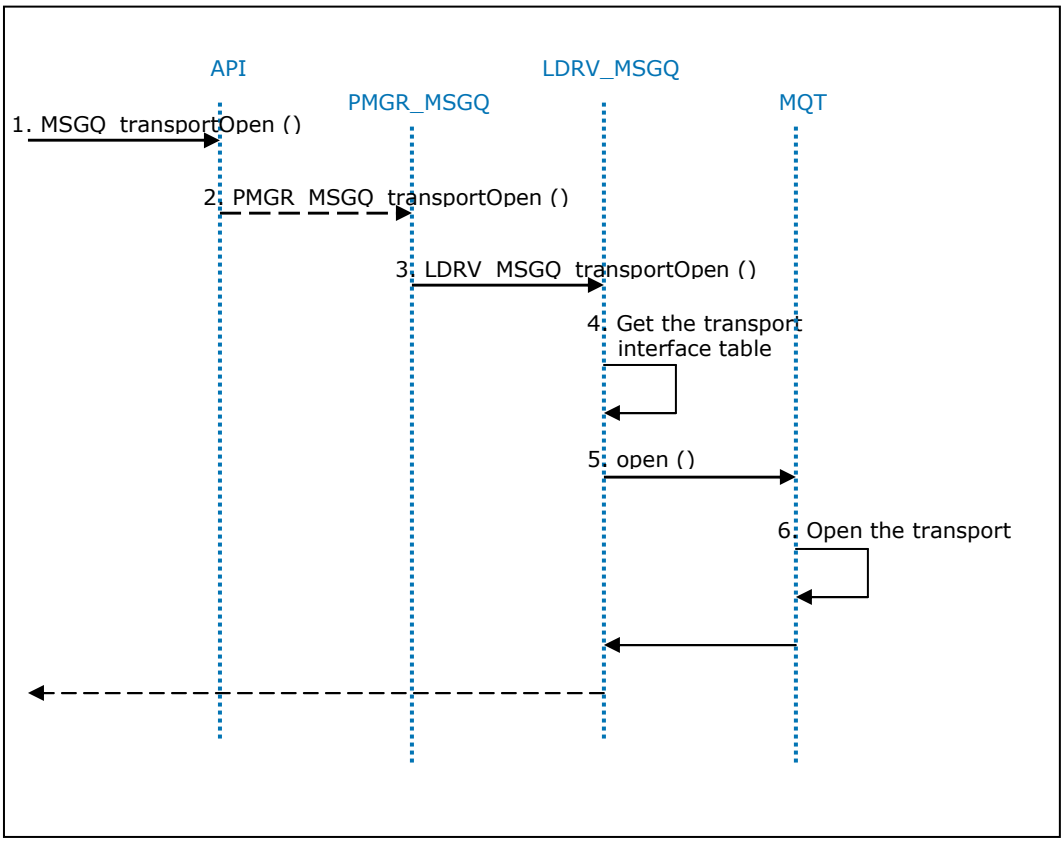

Figure 6. On the GPP: MSGQ_transportOpen () control flow

6.2 Finalization

6.2.1 MSGQ

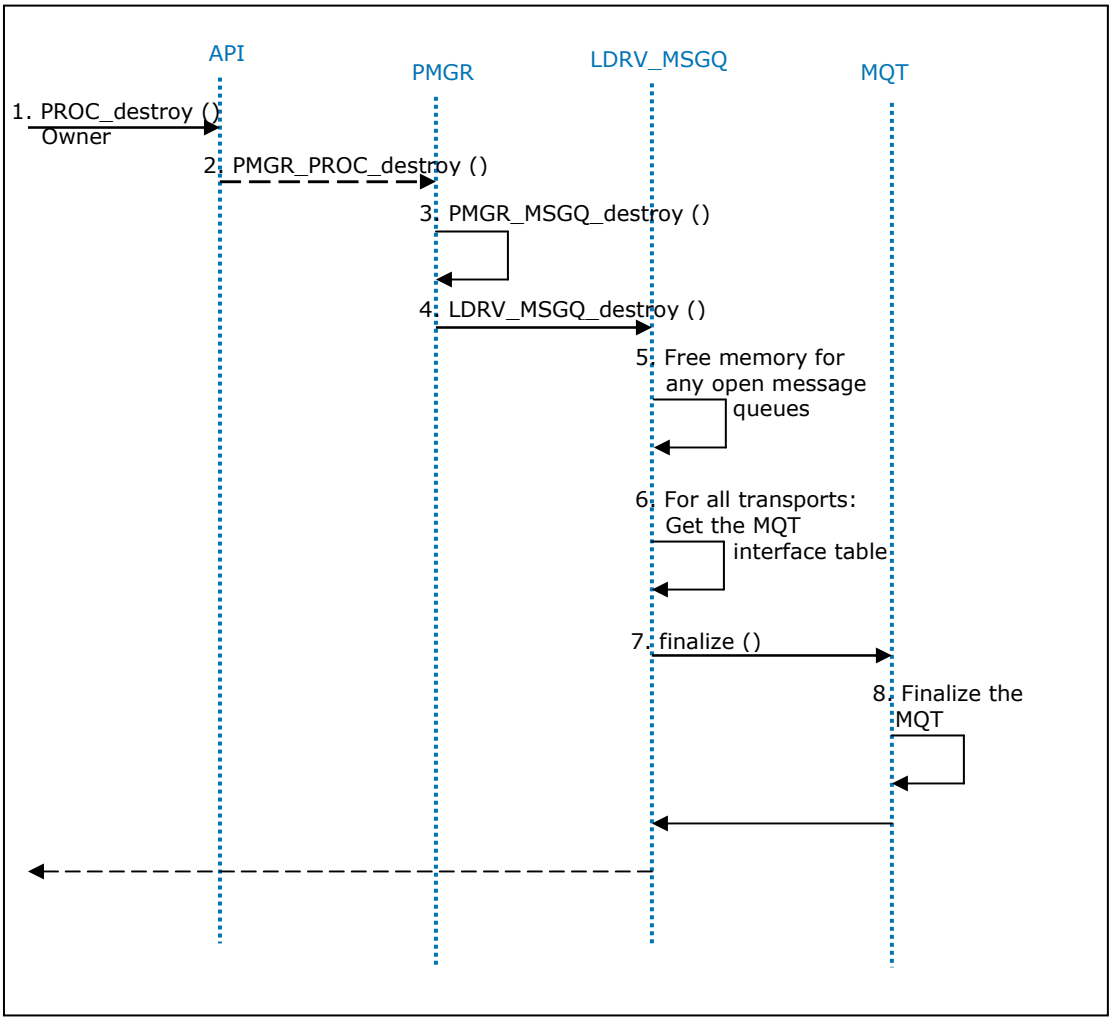

Figure 7. On the GPP: MSGQ finalization

6.2.2 Transport Close

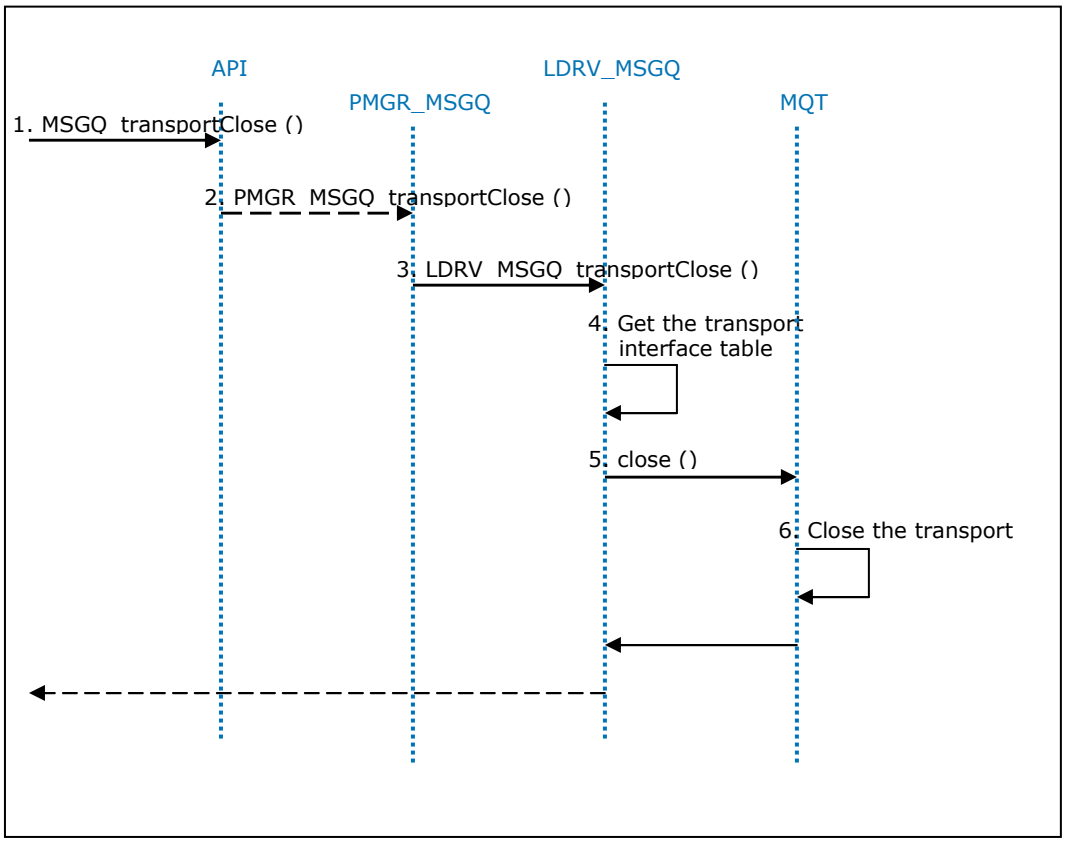

Figure 8. On the GPP: MSGQ_transportClose () control flow

6.3 MSGQ_open ()

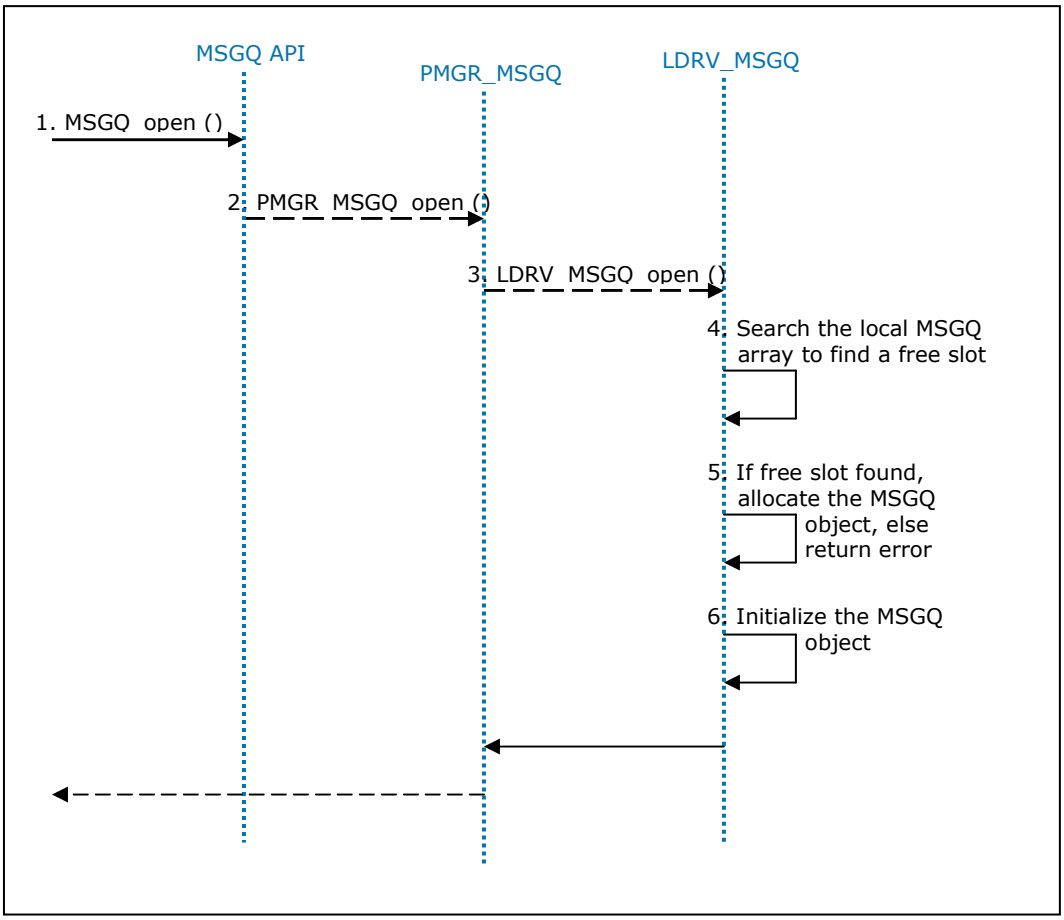

Figure 9. On the GPP: MSGQ_open () control flow

6.4 MSGQ_close ()

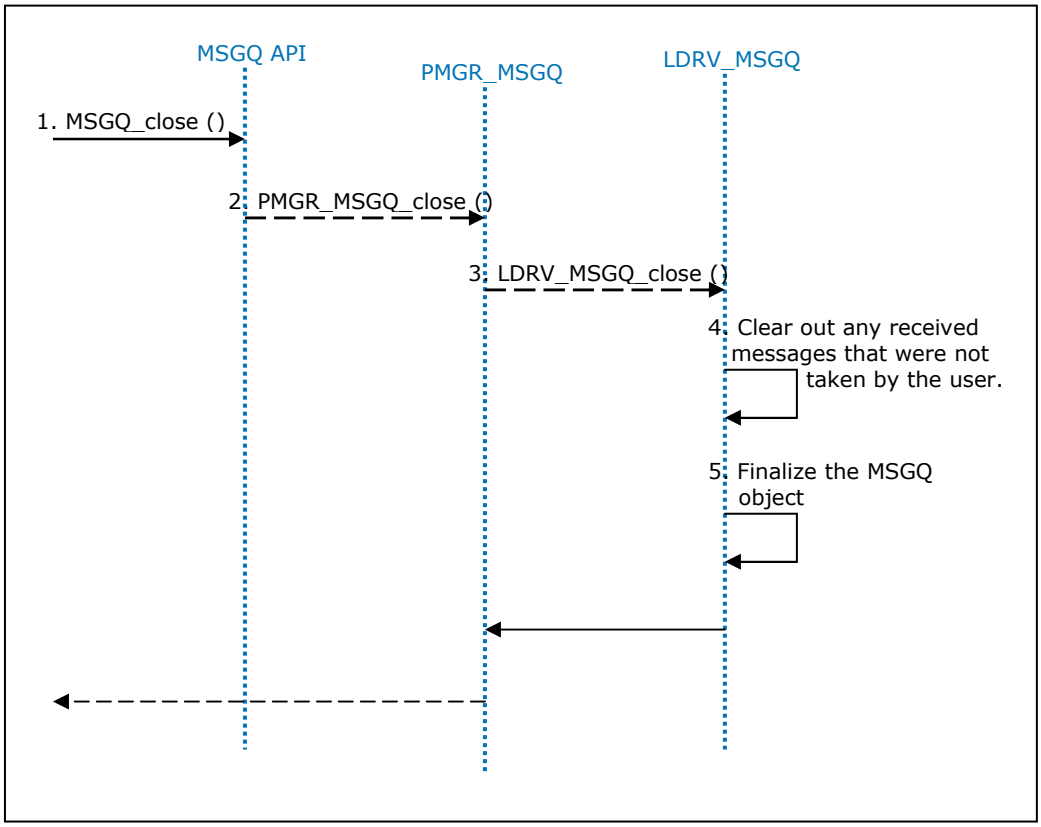

Figure 10. On the GPP: MSGQ_close () control flow

6.5 MSGQ_locate ()

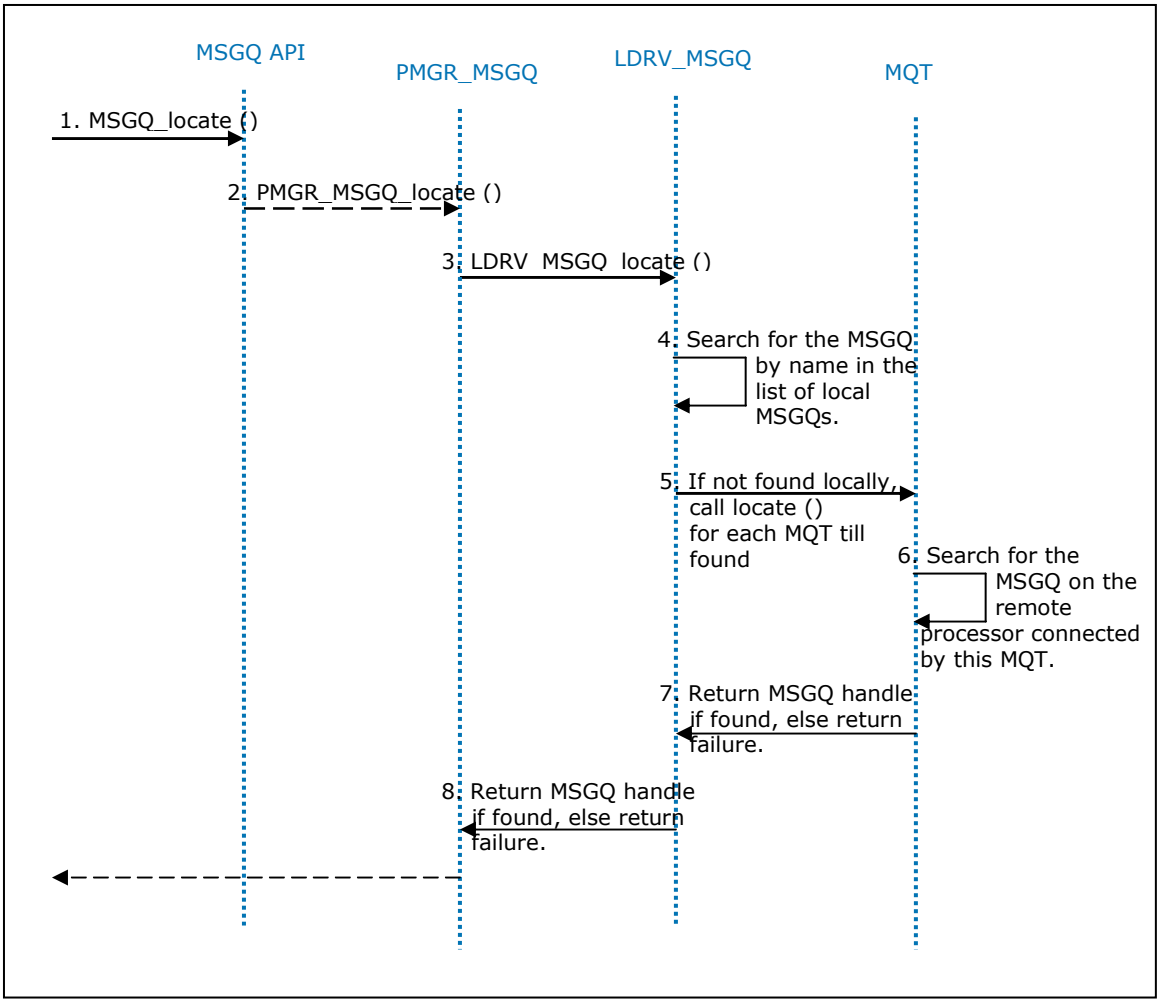

Figure 11. On the GPP: MSGQ_locate () control flow

6.6 LDRV_MSGQ_locateAsync

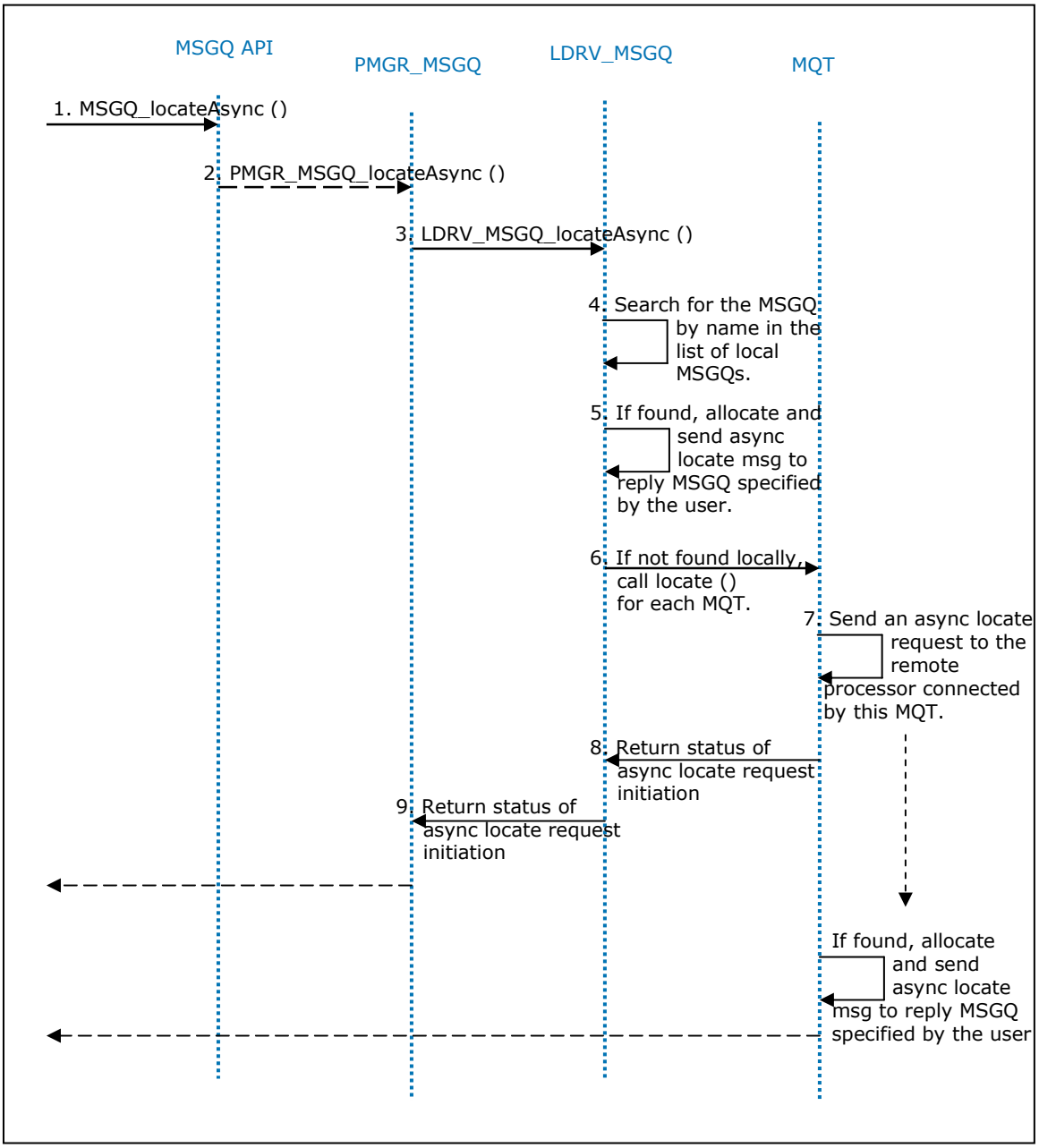

Figure 12. On the GPP: MSGQ_locateAsync () control flow

6.7 MSGQ_release ()

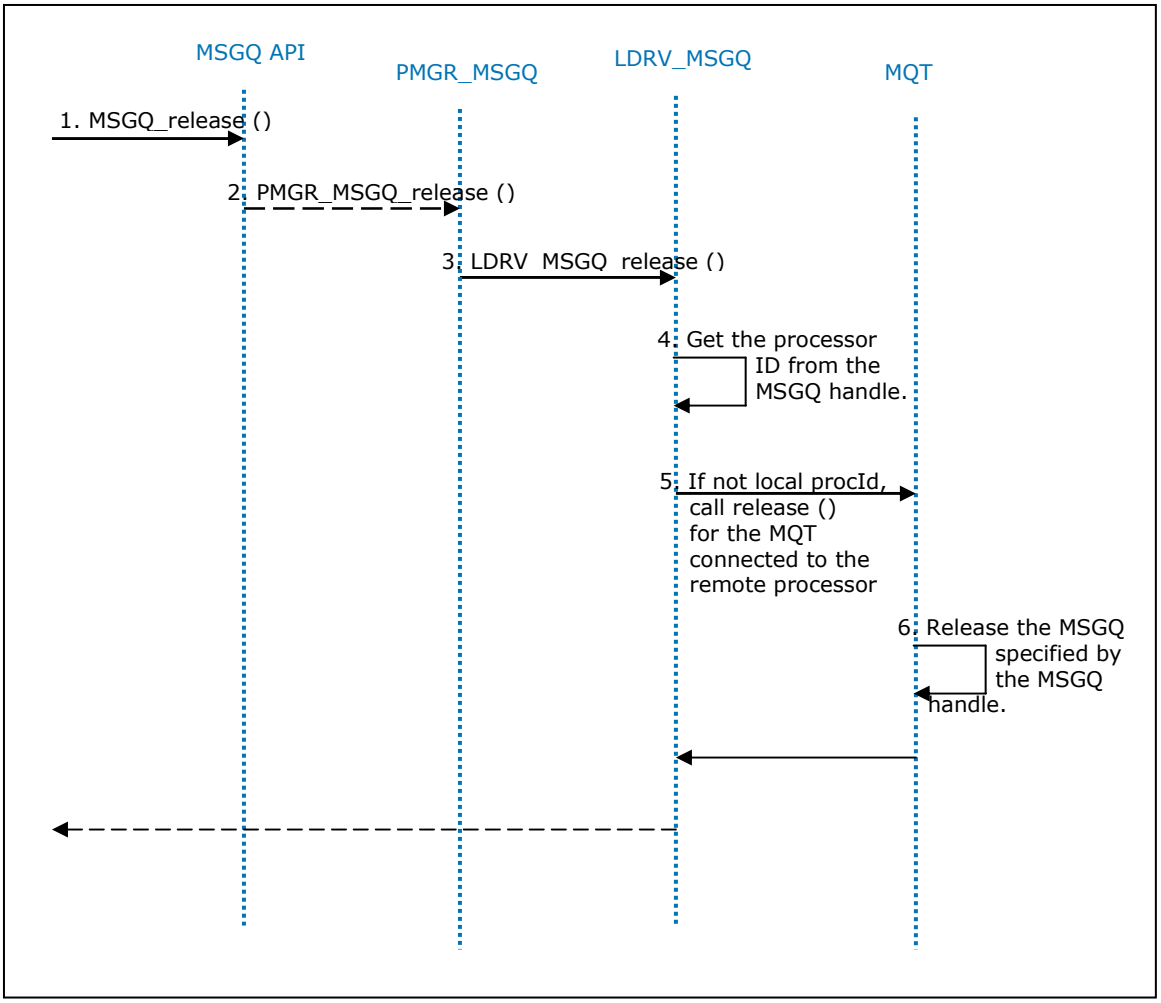

Figure 13. On the GPP: MSGQ_release () control flow

6.8 MSGQ_alloc ()

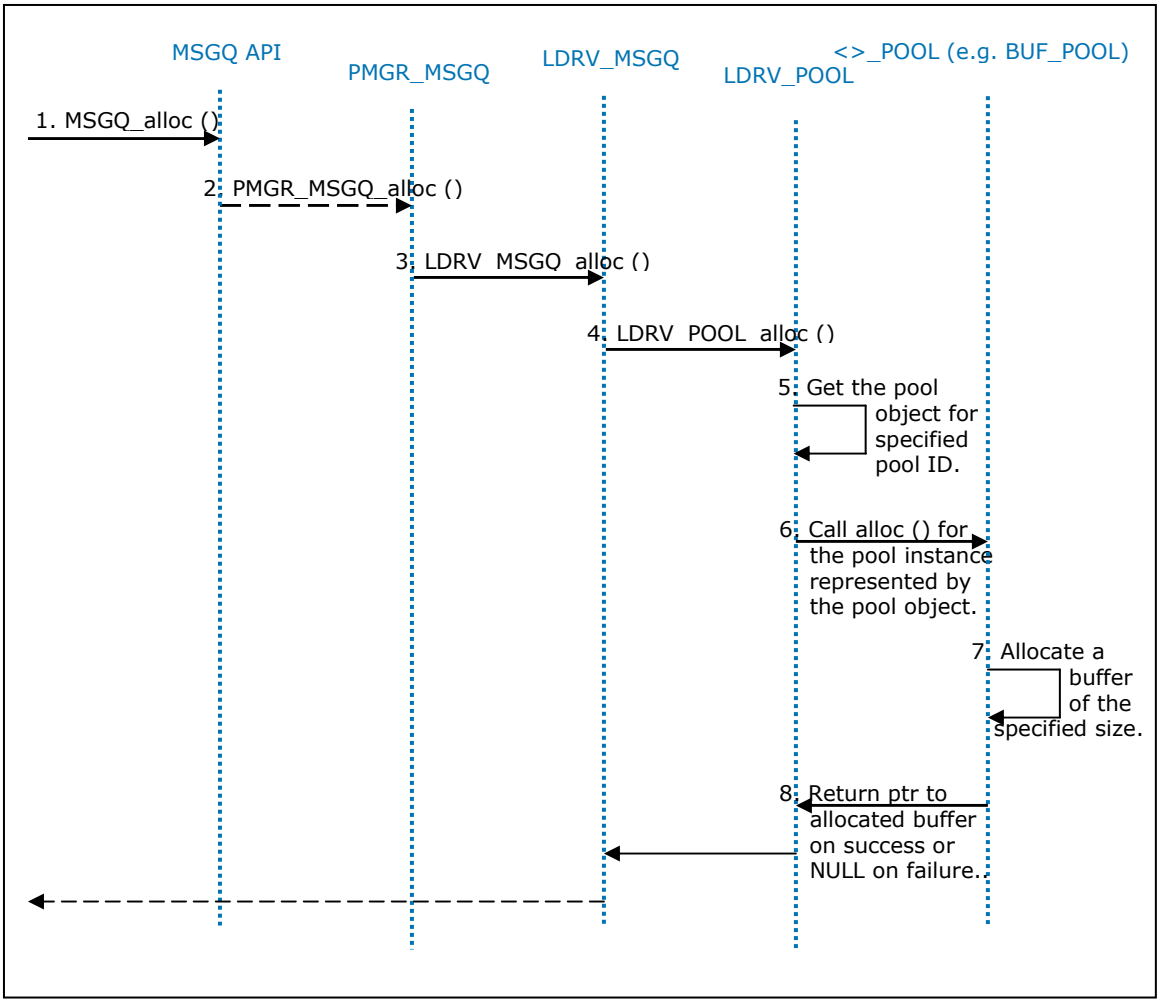

Figure 14. On the GPP: MSGQ_alloc () control flow

6.9 MSGQ_free ()

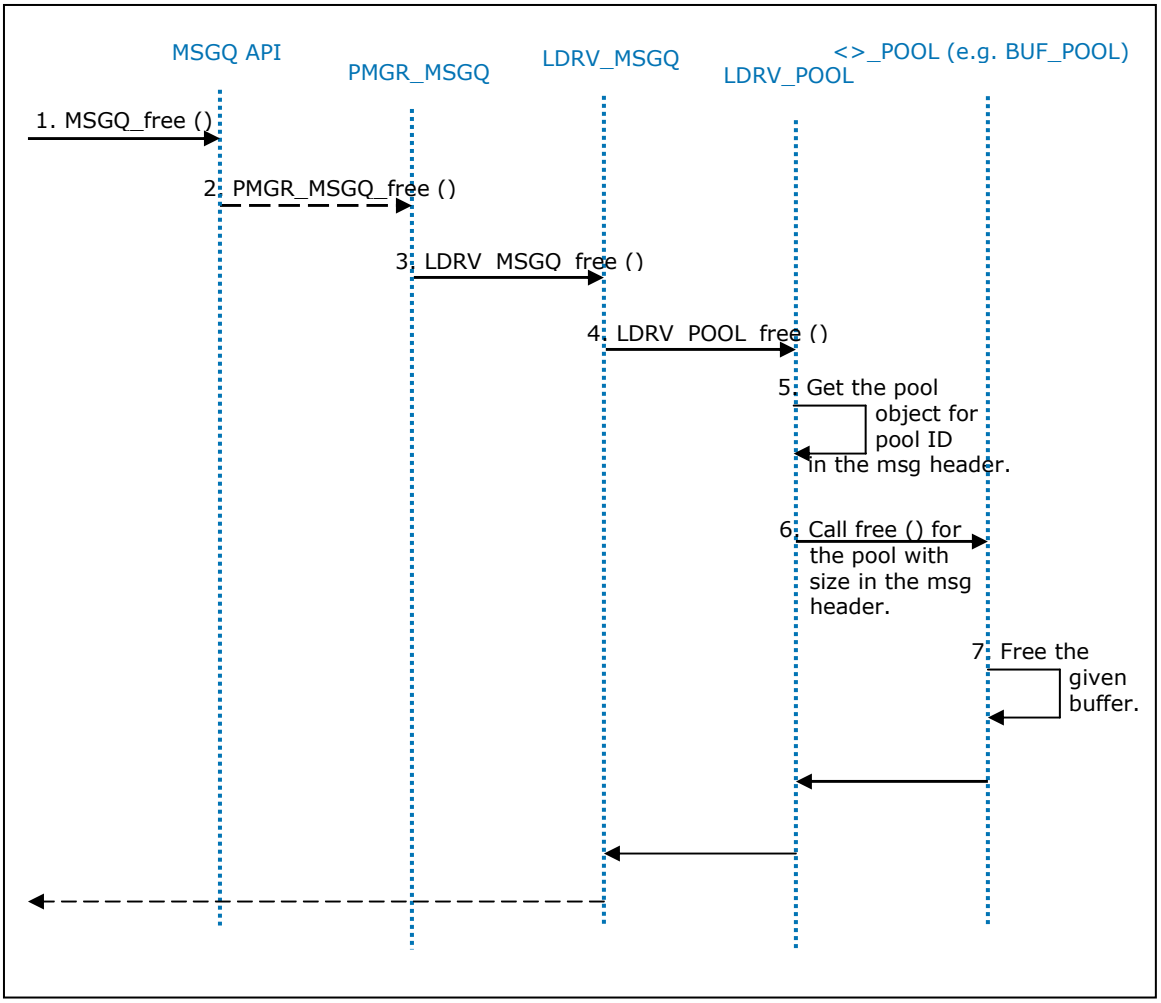

Figure 15. On the GPP: MSGQ_free () control flow

6.10 MSGQ_put ()

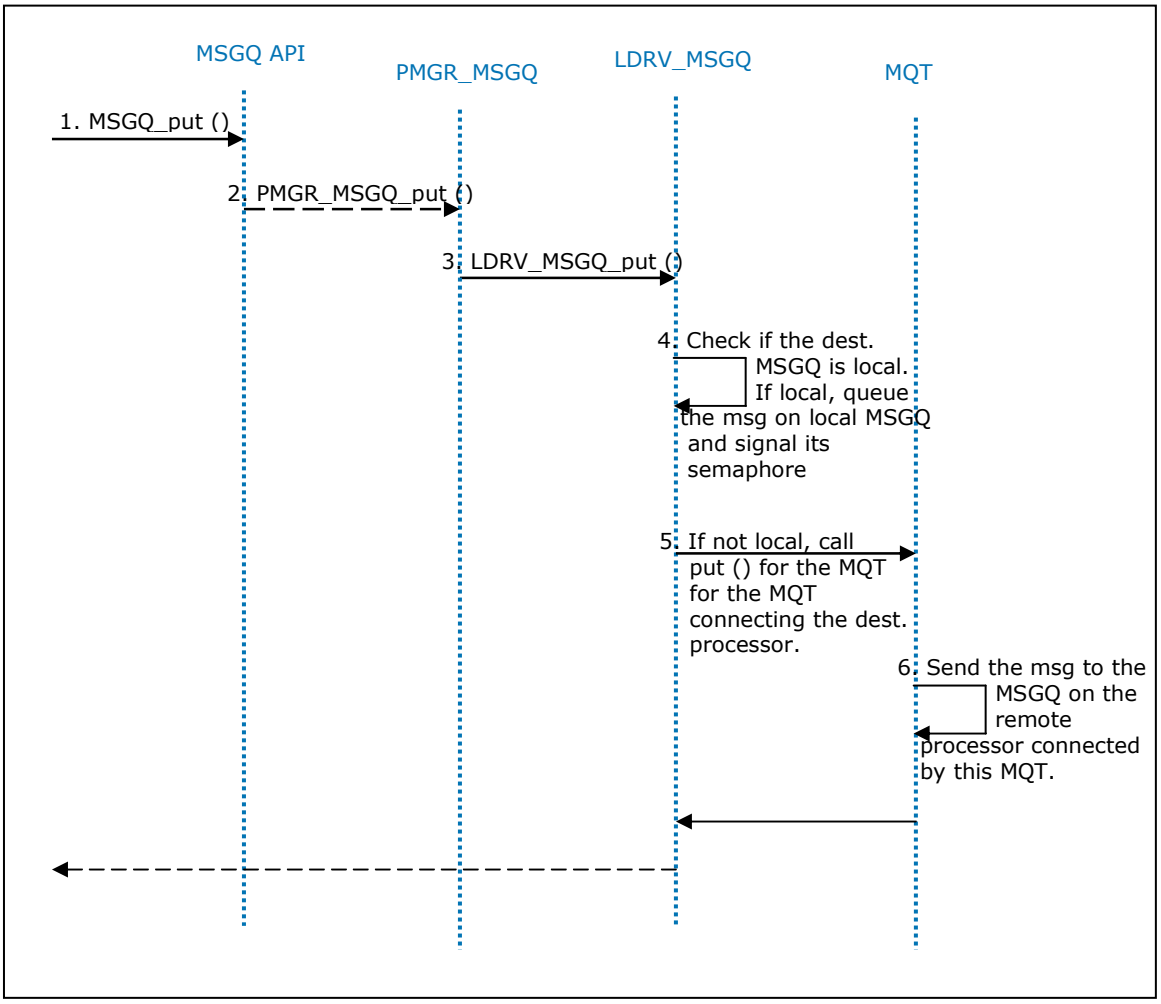

Figure 16. On the GPP: MSGQ_put () control flow

6.11 MSGQ_get ()

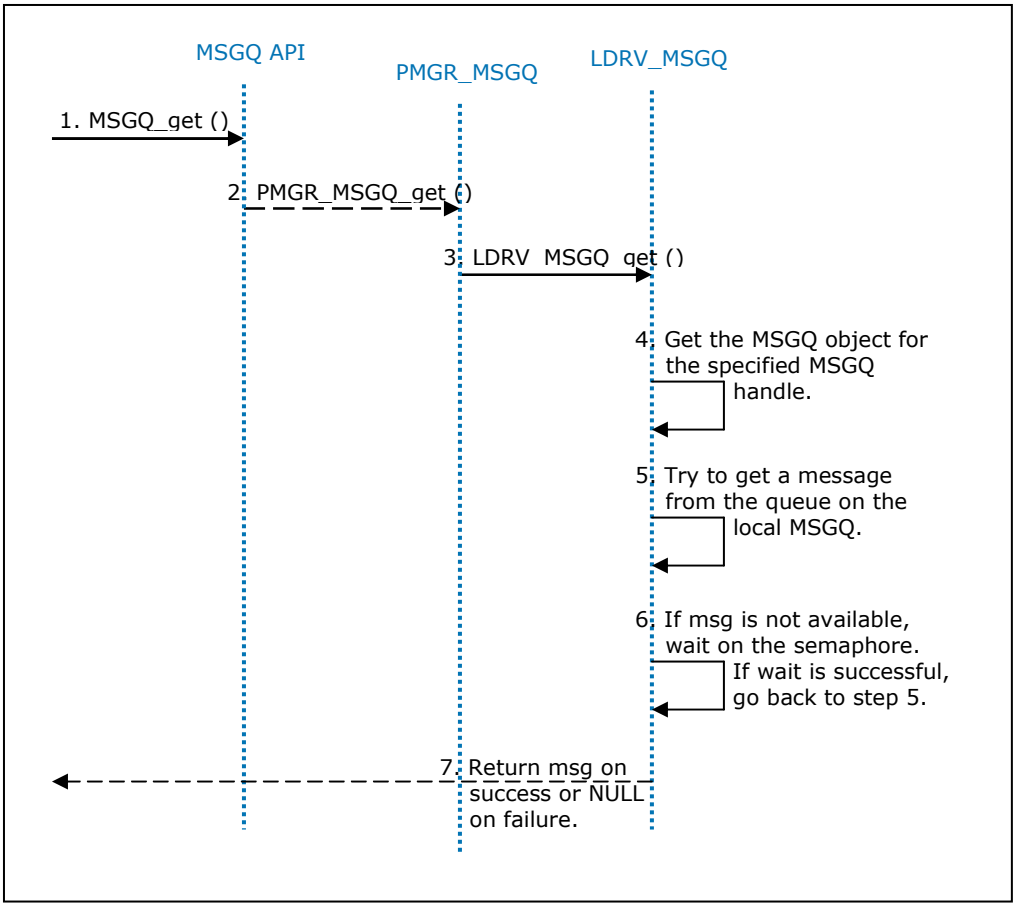

Figure 17. On the GPP: MSGQ_get () control flow

6.12 MSGQ_setErrorHandler ()

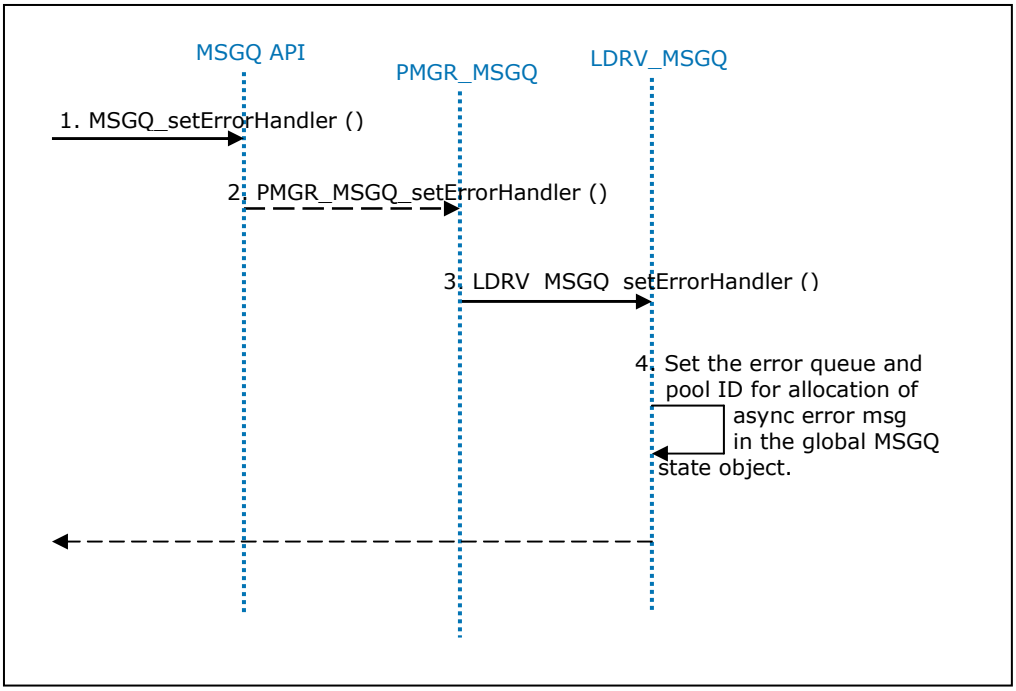

Figure 18. On the GPP: MSGQ_setErrorHandler () control flow

6.13 MSGQ_count ()

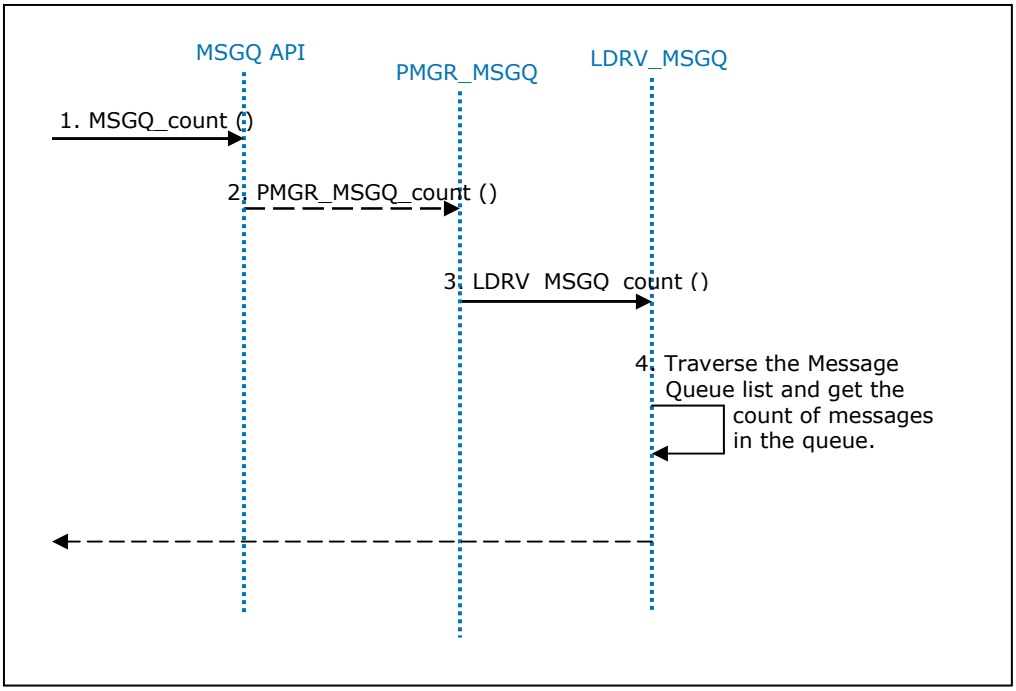

Figure 19. On the GPP: MSGQ_count () control flow

7 API

7.1 Constants & Enumerations

7.1.1 MSGQ_INVALIDMSGQ

This constant denotes an invalid message queue.

Definition

```
#define MSGQ_INVALIDMSGQ      (Uint16) 0xFFFF
```

Comments

None.

Constraints

None.

See Also

None.

7.1.2 MSGQ_INVALIDPROCID

This constant denotes an invalid processor ID.

Definition

```
#define MSGQ_INVALIDPROCID    (Uint16) 0xFFFF
```

Comments

None.

Constraints

None.

See Also

None.

7.1.3 MSGQ_INTERNALIDSSTART

This constant defines the start of internal MSGQ message ID range.

Definition

```
#define MSGQ_INTERNALIDSSTART (Uint16) 0xFF00
```

Comments

None.

Constraints

None.

See Also

None.

7.1.4 MSGQ_ASYNCLOCATEMSGID

This constant defines the asynchronous locate message ID.

Definition

```
#define MSGQ_ASYNCLOCATEMSGID (Uint16) 0xFF00
```

Comments

None.

Constraints

None.

See Also

None.

7.1.5 MSGQ_ASYNCERRORMSGID

This constant defines the asynchronous error message ID.

Definition

```
#define MSGQ_ASYNCERRORMSGID (Uint16) 0xFF01
```

Comments

None.

Constraints

None.

See Also

None.

7.1.6 MSGQ_INTERNALIDSEND

This constant defines the end of internal MSGQ message ID range.

Definition

```
#define MSGQ_INTERNALIDSEND (Uint16) 0xFF7f
```

Comments

None.

Constraints

None.

See Also

None.

7.1.7 MSGQ_MQTMSGIDSSTART

This constant defines the start of transport message ID range.

Definition

```
#define MSGQ_MQTMSGIDSSTART (Uint16) 0xFF80
```

Comments

None.

Constraints

None.

See Also

None.

7.1.8 MSGQ_MQTMSGIDSEND

This constant defines the end of transport message ID range.

Definition

```
#define MSGQ_MQTMSGIDSEND (Uint16) 0xFFFE
```

Comments

None.

Constraints

None.

See Also

None.

7.1.9 MSGQ_INVALIDMSGID

This constant is used to denote no message ID value.

Definition

```
#define MSGQ_INVALIDMSGID (Uint16) 0xFFFF
```

Comments

None.

Constraints

None.

See Also

None.

7.1.10 MSGQ_MQTERROREXIT

In an asynchronous error message, this value as the error type indicates that remote MQT has called exit.

Definition

```
#define MSGQ_MQTERROREXIT  (MSGQ_MqtError) -1
```

Comments

None.

Constraints

None.

See Also

None.

7.1.11 MSGQ_MQTFAILEDPUT

In an asynchronous error message, this value as the error type indicates that the transport failed to send a message to the remote processor.

Definition

```
#define MSGQ_MQTFAILEDPUT  (MSGQ_MqtError) -2
```

Comments

None.

Constraints

None.

See Also

None.

7.1.12 MSG_HEADER_RESERVED_SIZE

This macro defines the size of the reserved field of message header.

Definition

```
#define MSG_HEADER_RESERVED_SIZE    2
```

Comments

None.

Constraints

None.

See Also

MSGQ_MsgHeader

7.1.13 IS_VALID_MSGQ

This macro checks if a message queue is valid.

Definition

```
#define IS_VALID_MSGQ(msgq)        (msgq != MSGQ_INVALIDMSGQ)
```

Comments

None.

Constraints

None.

See Also

MSGQ_Queue

7.1.14 MSGQ_getMsgId

This macro returns the message ID of the specified message.

Definition

```
#define MSGQ_getMsgId(msg) (((MSGQ_Msg) (msg))->msgId)
```

Comments

The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user. For this purpose, macros or functions are provided to access fields within the message header.

Constraints

None.

See Also

`MSGQ_MsgHeader`

7.1.15 MSGQ_getMsgSize

This macro returns the size of the specified message.

Definition

```
#define MSGQ_getMsgSize(msg) (((MSGQ_Msg) (msg))->size)
```

Comments

The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user. For this purpose, macros or functions are provided to access fields within the message header.

Constraints

None.

See Also

`MSGQ_MsgHeader`

7.1.16 MSGQ_setMsgId

This macro sets the message ID in the specified message.

Definition

```
#define MSGQ_setMsgId(msg, id) ((MSGQ_Msg) (msg))->msgId = id
```

Comments

The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user. For this purpose, macros or functions are provided to access fields within the message header.

Constraints

None.

See Also

`MSGQ_MsgHeader`

7.1.17 MSGQ_getDstQueue

This macro returns the MSGQ_Queue handle of the destination message queue for the specified message.

Definition

```
#define MSGQ_getDstQueue(msg) ((Uint32) ID_GPP << 16) | \
                               ((MSGQ_Msg) (msg))->dstId
```

Comments

The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user. For this purpose, macros or functions are provided to access fields within the message header.

Constraints

None.

See Also

`MSGQ_MsgHeader`

7.1.18 MSGQ_setSrcQueue

This macro sets the source message queue in the specified message.

Definition

```
#define MSGQ_setSrcQueue(msg, msgq) \
                          ((MSGQ_Msg) (msg))->srcId = (MSGQ_Id) msgq ; \
                          ((MSGQ_Msg) (msg))->srcProcId = msgq >> 16
```

Comments

The contents of the message header are reserved for use internally within *DSPLINK* and should not directly be modified by the user. For this purpose, macros or functions are provided to access fields within the message header.

Constraints

None.

See Also

MSGQ_MsgHeader

7.1.19 MSGQ_isLocalQueue

This macro checks whether the specified queue is a local queue.

Definition

```
#define MSGQ_isLocalQueue(msgq) ((msgq >> 16) == ID_GPP)
```

Comments

The message queue handle is a value composed of the processor ID and message queue ID. This macro identifies whether the message queue represented by the specified handle exists on the local processor.

Constraints

None.

See Also

`MSGQ_Queue`

7.2 Typedefs & Data Structures

7.2.1 MSGQ_MqtError

This type is used for identifying types of MQT asynchronous error messages.

Definition

```
typedef Int16    MSGQ_MqtError ;
```

Comments

None.

Constraints

None.

See Also

```
MSGQ_AsyncErrorMsg
LDRV_MSGQ_sendErrorMsg ()
```

7.2.2 MSGQ_Id

This type is used for identifying a message queue on a specific processor.

Definition

```
typedef Uint16   MSGQ_Id ;
```

Comments

None.

Constraints

None.

See Also

MSGQ_Queue

7.2.3 MSGQ_Queue

This type is used for identifying a message queue across processors.

Definition

```
typedef Uint32   MSGQ_Queue ;
```

Comments

A MSGQ_Queue handle is a system-wide unique handle to the message queue, consisting of both the processor ID on which the message exists, and the message queue ID on the specific processor.

Constraints

None.

See Also

MSGQ_Id
ProcessorId

7.2.4 MSGQ_Attrs

This structure defines the attributes required during opening of the MSGQ.

Definition

```
typedef struct MSGQ_Attrs_tag {
    Uint16     dummy ;
} MSGQ_Attrs ;
```

Fields

`dummy`	Dummy placeholder field.

Comments

This structure defines the attributes structure for `MSGQ_open ()` and is provided for extensibility. No attributes are required currently, and the structure consists of a dummy placeholder field.

Constraints

None.

See Also

`MSGQ_open ()`

7.2.5 MSGQ_LocateAttrs

This structure defines the attributes required during synchronous location of a MSGQ.

Definition

```
typedef struct MSGQ_LocateAttrs_tag {
    Uint32     timeout ;
} MSGQ_LocateAttrs ;
```

Fields

`timeout`	Timeout value in milliseconds for the locate call.

Comments

This structure defines the attributes structure for `MSGQ_locate ()`.

Constraints

None.

See Also

`MSGQ_locate ()`

7.2.6 MSGQ_LocateAsyncAttrs

This structure defines the attributes required during asynchronous location of a MSGQ.

Definition

```
typedef struct MSGQ_LocateAsyncAttrs_tag {
    PoolId     poolId ;
    Pvoid      arg ;
} MSGQ_LocateAsyncAttrs ;
```

Fields

`poolId`	ID of the pool to be used for allocating asynchronous locate messages.
`arg`	User-defined argument returned with an asynchronous locate message.

Comments

This structure defines the attributes structure for `MSGQ_locateAsync ()`.

Constraints

None.

See Also

`MSGQ_locateAsync ()`

7.2.7 MSGQ_MsgHeader

This structure defines the format of the message header that must be the first field of any message.

Definition

```
typedef struct MSGQ_MsgHeader_tag {
    Uint32    reserved [MSG_HEADER_RESERVED_SIZE] ;
    Uint16    srcProcId                           ;
    Uint16    poolId                              ;
    Uint16    size                                ;
    Uint16    dstId                               ;
    Uint16    srcId                               ;
    Uint16    msgId                               ;
} MSGQ_MsgHeader ;

typedef MSGQ_MsgHeader * MSGQ_Msg ;
```

Fields

`reserved`	Reserved for use by the MQT. The MQT typically uses them as a link for queuing the messages.
`srcProcId`	Processor ID for the source message queue
`poolId`	ID of the Pool used for allocating this message.
`size`	Size of the message including the header.
`dstId`	ID of the destination message queue.
`srcId`	ID of the source message queue for reply.
`msgId`	User-specified message ID.

Comments

The message header must be the first field in the message structure defined by the user. The contents of the message header are reserved for use internally within *DSPLINK* and should not be modified directly by the user.

Constraints

None.

See Also

None

7.2.8 MSGQ_AsyncLocateMsg

This structure defines the asynchronous locate message format.

Definition

```
typedef struct MSGQ_AsyncLocateMsg_tag {
    MSGQ_MsgHeader  header ;
    MSGQ_Queue      msgqQueue ;
    Pvoid           arg ;
} MSGQ_AsyncLocateMsg ;
```

Fields

`header`	Fixed message header required for all messages.
`msgqQueue`	Reply message queue specified during `MSGQ_locateAsync ()`
`arg`	User-defined argument specified as part of the `MSGQ_LocateAsyncAttrs`

Comments

When an asynchronous location completes with success, the handle of the located message queue is sent to the user application through a message of this type.

Constraints

None.

See Also

`MSGQ_LocateAsyncAttrs`
`MSGQ_locateAsync ()`

7.2.9 MSGQ_AsyncErrorMsg

This structure defines the asynchronous error message format.

Definition

```
typedef struct MSGQ_AsyncErrorMsg_tag {
    MSGQ_MsgHeader  header ;
    MSGQ_MqtError   errorType ;
    Pvoid          arg1 ;
    Pvoid          arg2 ;
} MSGQ_AsyncErrorMsg ;
```

Fields

`header`	Fixed message header required for all messages.
`errorType`	Type of error.
`arg1`	First argument dependent on the error type. `MSGQ_MQTERROREXIT`: Processor ID of the transport. `MSGQ_MQTFAILEDPUT`: Handle of the destination message queue on which the put failed.
`arg2`	Second argument dependent on the error type. `MSGQ_MQTERROREXIT`: Not used. `MSGQ_MQTFAILEDPUT`: Status of the `MSGQ_put ()` call that failed.

Comments

The asynchronous error message is sent by the transport to a message queue registered by the user, on occurrence of an error.

The user can register an error handler MSGQ for receiving asynchronous error messages indicating transport errors. The error message is of a predefined format.

The first field after the required message header of the `MSGQ_AsyncErrorMsg` asynchronous error message indicates the error type. The argument fields in the error message hold different values for each error type.

Constraints

The asynchronous error message is sent by the transport only if the user has registered an error-handler message queue with the MSGQ component.

See Also

`MSGQ_MqtError`
`MSGQ_setErrorHandler ()`

7.2.10 MSGQ_Instrument

This structure defines the instrumentation data for a message queue.

Definition

```
#if defined (DDSP_PROFILE)
typedef struct MSGQ_Instrument_tag {
    MSGQ_Queue   msgqQueue ;
    Uint32      transferred ;
    Uint32      queued ;
} MSGQ_Instrument ;
#endif /* if defined (DDSP_PROFILE) */
```

Fields

`msgqQueue`	Message queue handle. If `MSGQ_INVALIDMSGQ`, indicates that the message queue has not been opened.
`transferred`	Number of messages transferred on this MSGQ.
`queued`	Number of messages currently queued on this MSGQ, pending calls to get them.

Comments

This structure is available to the user applications through the profiling feature.

Constraints

This structure is defined only if profiling is enabled within *DSPLINK*.

See Also

`MSGQ_instrument ()`

7.2.11 MSGQ_Stats

This structure defines the instrumentation data for MSGQs on the local processor.

Definition

```
#if defined (DDSP_PROFILE)
typedef struct MSGQ_Stats_tag {
    MSGQ_Instrument msgqData [MAX_MSGQS] ;
} MSGQ_Stats ;
#endif /* if defined (DDSP_PROFILE) */
```

Fields

`msgqData`	Instrumentation data for the local MSGQs.

Comments

This structure is available to the user applications through the profiling feature.

Constraints

This structure is defined only if profiling is enabled within *DSPLINK*.

See Also

`MSGQ_Instrument`
`MSGQ_instrument ()`

7.3 API Definition

7.3.1 MSGQ_transportOpen

This function initializes the transport associated with the specified processor.

Syntax

```
DSP_STATUS MSGQ_transportOpen (ProcessorId procId,
                               Pvoid       attrs) ;
```

Arguments

IN ProcessorId procId

ID of the Processor for which the transport is to be opened.

IN Pvoid attrs

Attributes for initialization of the transport. The structure of the expected attributes is specific to a transport.

Return Value

DSP_SOK	The MQT component has been successfully opened.
DSP_SALREADYOPENED	The MSGQ transport for the specified processor has already been opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EINVALIDARG	Invalid argument.
DSP_EACCESSDENIED	Transport already open.
DSP_EFAIL	General failure.

Comments

The transport corresponding to the processor ID specified in the call should be configured in the CFG.

When any client wishes to use messaging with a specific DSP, it needs to open the MSGQ transport for the DSP by calling this API specifying the required DSP ID.

This API carries out all initialization required to be able to use messaging with the specified DSP ID from the calling process. This API can be successfully called once by every process in the system after calling PROC_attach (). However, it is not necessary that each process must call the API if another process has already previously opened the transport.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

The configuration of the MQTs is done as part of the CFG. This includes configuration of the fixed attributes specific to each MQT. This configuration also defines the IDs of

the MQTs. These IDs must be used while deciding the attributes required by each MQT.

See Also

`MSGQ_transportClose ()`

7.3.2 MSGQ_transportClose

This function finalizes the transport associated with the specified processor.

Syntax

```
DSP_STATUS MSGQ_transportClose (ProcessorId procId) ;
```

Arguments

IN	ProcessorId	procId

ID of the Processor for which the transport is to be closed.

Return Value

DSP_SOK	The MQT component has been successfully closed.
DSP_SCLOSED	The final process has closed the MSGQ transport.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EINVALIDARG	Invalid argument.
DSP_EOPENED	The MSGQ transport was not opened.
DSP_EACCESSDENIED	The MSGQ transport was not opened in this process.
DSP_EFAIL	General failure.

Comments

All applications/processes can call this API once they no longer need to use *DSPLINK* messaging for sending/receiving messages to/from the specific processor. Once this API has been called, the process cannot perform any further messaging activities specific to the DSP.

This API finalizes the *DSPLINK* Message Queue Transport for the specified processor ID in the calling process. This API can be successfully called once by every process in the system. However, if the `MSGQ_transportOpen ()` API for the specific processor ID was not called in the process, `MSGQ_transportClose ()` must not be called.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

None.

See Also

`MSGQ_transportOpen ()`

7.3.3 MSGQ_open

This function opens the message queue to be used for receiving messages, identified through the specified message queue name.

Syntax

```
DSP_STATUS MSGQ_open (Pstr        queueName,
                      MSGQ_Queue * msgqQueue,
                      MSGQ_Attrs * attrs) ;
```

Arguments

IN	Pstr	queueName

Name of the message queue to be opened.

OUT	MSGQ_Queue *	msgqQueue

Location to store the handle to the message queue.

IN OPT	MSGQ_Attrs *	attrs

Optional attributes for creation of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully opened.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	Attempt to open more than number of message queues configured.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This API is called only for receiver message queues. To send a message to any MSGQ, its existence is verified through an MSGQ_locate () call, following which messages can be sent to it.

The attributes parameter is provided for future extensibility and can be passed as NULL.

Constraints

None.

See Also

MSGQ_Queue
MSGQ_Attrs
MSGQ_close ()
MSGQ_locate ()

7.3.4 MSGQ_close

This function closes the message queue identified by the specified MSGQ handle.

Syntax

```
DSP_STATUS MSGQ_close (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue to be closed.

Return Value

DSP_SOK	The message queue has been successfully closed.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This API is called only for receiver message queues.

Constraints

None.

See Also

MSGQ_Queue
MSGQ_open ()

7.3.5 MSGQ_locate

This function synchronously locates the message queue identified by the specified MSGQ name and returns a handle to the located message queue.

Syntax

```
DSP_STATUS MSGQ_locate (Pstr              queueName,
                        MSGQ_Queue *       msgqQueue,
                        MSGQ_LocateAttrs * attrs) ;
```

Arguments

IN	Pstr	queueName	
	Name of the message queue to be located.		
OUT	MSGQ_Queue *	msgqQueue	
	Location to store the handle to the located message queue.		
IN OPT	MSGQ_LocateAttrs *	attrs	
	Optional attributes for location of the MSGQ.		

Return Value

DSP_SOK	The message queue has been successfully located.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	The specified message queue could not be located.
DSP_ETIMEOUT	Timeout occurred while locating the MSGQ.
DSP_ENOTCOMPLETE	Operation not complete when WAIT_NONE was specified as timeout.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This API is called to get a handle to a message queue that may exist on any processor in the system. The message queue handle obtained after successful completion of this API can be used for sending a message to the located MSGQ

Constraints

The default pool specified by the user for internal use by an MQT must be configured before this API can be called for that MQT.

It may happen that the MSGQ exists when the `MSGQ_locate ()` call is made, but is deleted shortly after. In that case, it cannot be ensured that an `MSGQ_put ()` call successfully transfers the message to the destination MSGQ.

See Also

`MSGQ_Queue`

```
MSGQ_LocateAttrs
MSGQ_put ()
MSGQ_release ()
```

7.3.6 MSGQ_locateAsync

This function asynchronously locates the message queue identified by the specified MSGQ name. An attempt is made to asynchronously locate the message queue. If the message queue is found, an `MSGQ_AsyncLocateMsg` message is sent to the specified reply message queue.

Syntax

```
DSP_STATUS MSGQ_locateAsync (Pstr                    queueName,
                             MSGQ_Queue               replyQueue,
                             MSGQ_LocateAsyncAttrs *  attrs) ;
```

Arguments

`IN` `Pstr` `queueName`

Name of the message queue to be located.

`IN` `MSGQ_Queue` `replyQueue`

Location to store the handle to the located message queue.

`IN` `MSGQ_LocateAsyncAttrs *` `attrs`

Attributes for asynchronous location of the MSGQ.

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This API is called to get a handle to a message queue that may exist on any processor in the system. Before sending a message to the remote MSGQ, a handle to the message queue must be obtained by calling this API, and then waiting for a response `MSGQ_AsyncLocateMsg` message on the reply message queue passed to the function.

Constraints

The default pool specified by the user for internal use by an MQT must be configured before this API can be called for that MQT.

See Also

```
MSGQ_Queue
MSGQ_LocateAsyncAttrs
MSGQ_put ()
MSGQ_release ()
```

7.3.7 MSGQ_release

This function releases the message queue identified by the MSGQ handle that was located earlier.

Syntax

```
DSP_STATUS MSGQ_release (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue to be released.

Return Value

DSP_SOK	The message queue has been successfully released.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	The message queue has not been previously located.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This API is the counterpart to the `MSGQ_locate ()` and `MSGQ_locateAsync ()` APIs. It releases any resources allocated during the locate APIs. Once the MSGQ has been released, it needs to be located once again before sending a message to it.

The application can also use this API for carrying out the cleanup required after a remote MSGQ has been deleted.

Constraints

None.

See Also

```
MSGQ_Queue
MSGQ_locate ()
MSGQ_locateAsync ()
```

7.3.8 MSGQ_alloc

This function allocates a message, and returns the pointer to the user.

Syntax

```
DSP_STATUS MSGQ_alloc (PoolId poolId, Uint16 size, MSGQ_Msg * msg) ;
```

Arguments

IN	PoolId	poolId

ID of the Pool to be used for allocating this message.

IN	Uint16	size

Size of the message to be allocated.

OUT	MSGQ_Msg *	msg

Location to receive the allocated message.

Return Value

DSP_SOK	The message has been successfully allocated.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This API allocates a message that shall be used during MSGQ_put () API calls.

Constraints

Once this message has been transferred through MSGQ_put (), the receiver owns it. Following this, the sender must not attempt to free this message.

See Also

MSGQ_MsgHeader
MSGQ_put ()

7.3.9 MSGQ_free

This function frees a message.

Syntax

```
DSP_STATUS MSGQ_free (MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Msg	msg

Pointer to the message to be freed.

Return Value

DSP_SOK	The message has been successfully freed.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This API frees a message that was received through an MSGQ_get () call or MSGQ_alloc () call. Once this message has been received through MSGQ_get (), the receiver owns it, and can free it if so desired. The message can also be reused for sending it to a MSGQ, as long as it fits within the existing message size.

Constraints

None.

See Also

MSGQ_MsgHeader
MSGQ_get ()

7.3.10 MSGQ_put

This function sends a message to the specified MSGQ.

Syntax

```
DSP_STATUS MSGQ_put (MSGQ_Queue msgqQueue, MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the destination MSGQ.

IN	MSGQ_Msg	msg

Pointer to the message to be sent to the destination MSGQ.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This function must be non-blocking and deterministic.

Constraints

The successful completion of this API does not guarantee completion of actual transfer over the physical link.

See Also

```
MSGQ_Queue
MSGQ_MsgHeader
MSGQ_get ()
```

7.3.11 MSGQ_get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS MSGQ_get (MSGQ_Queue  msgqQueue,
                     Uint32     timeout,
                     MSGQ_Msg *  msg) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the MSGQ on which the message is to be received.

IN	timeout	timeout

Timeout value to wait for the message (in milliseconds).

OUT	MSGQ_Msg *	msg

Location to receive the message.

Return Value

DSP_SOK	The message has been successfully received.
DSP_EINVALIDARG	Invalid argument.
DSP_ETIMEOUT	Timeout occurred while receiving the message.
DSP_ENOTCOMPLETE	Operation not complete when WAIT_NONE was specified as timeout.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

A timeout of zero can be specified if this API is desired to be non-blocking. In that case, a message is taken from the MSGQ if it is already available. Otherwise, an error is returned.

After the message has been received, it is owned by the receiver application, and can be freed by the application whenever so desired, or reused.

Constraints

None.

See Also

MSGQ_Queue
MSGQ_MsgHeader
MSGQ_put ()

7.3.12 MSGQ_getSrcQueue

This function returns a handle to the source message queue of a message to be used for replying to the message.

Syntax

```
DSP_STATUS MSGQ_getSrcQueue (MSGQ_Msg msg, MSGQ_Queue * msgqQueue) ;
```

Arguments

IN	MSGQ_Msg	msg

Message, whose source MSGQ handle is to be returned.

OUT	MSGQ_Queue *	msgqQueue

Location to retrieve the handle to the source MSGQ.

Return Value

DSP_SOK	The reply information has been successfully retrieved.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	Source information has not been provided by the sender.
DSP_EFAIL	General failure.

Comments

This API is used for extracting information required for sending a reply message back to the application that had sent the message. If an application expects a reply message, it must specify the handle to the MSGQ of a local MSGQ for receiving the reply message from the remote processor.

After getting the reply MSGQ handle, the user can send a reply message using MSGQ_put ().

Constraints

A reply message cannot be sent back if the source application has not specified the source MSGQ handle.

See Also

MSGQ_Queue
MSGQ_MsgHeader

7.3.13 MSGQ_count

This API returns the count of the number of messages in a local message queue.

Syntax

```
DSP_STATUS MSGQ_count (MSGQ_Queue msgqQueue, Uint16 * count) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the MSGQ for which the count is to be retrieved.

OUT	Uint16 *	count

Location to receive the message count.

Return Value

DSP_SOK	The count has been successfully retrieved.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This API is used to retrieve the count of the number of messages currently queued up within a local message queue.

Constraints

This API is not thread-safe, and must be called only by the reader of the message queue.

See Also

MSGQ_Queue

7.3.14 MSGQ_setErrorHandler

This API allows the user to designate a MSGQ as an error-handler MSGQ to receive asynchronous error messages from the transports.

Syntax

```
DSP_STATUS MSGQ_setErrorHandler (MSGQ_Queue errorQueue, PoolId poolId)
;
```

Arguments

IN	MSGQ_Queue	errorQueue

Handle to the message queue to receive the error messages.

IN	PoolId	poolId

ID indicating the pool to be used for allocating the error messages.

Return Value

DSP_SOK	The error handler has been successfully set.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

The user can designate any message queue as an error handler MSGQ using this API. The same MSGQ can also be used for receiving other messages, apart from the error messages. After this API has been called, the transport notifies the user of any asynchronous error occurring during its operations, by sending a message to the designated error handler MSGQ. The format of the error message and the different types of errors that are notified are fixed.

Constraints

The error handler MSGQ must be created before this API can be called.

See Also

MSGQ_MqtError
MSGQ_AsyncErrorMsg

7.3.15 MSGQ_instrument

This function gets the instrumentation information related to the specified message queue.

Syntax

```
DSP_STATUS MSGQ_instrument (MSGQ_Queue        msgqQueue,
                            MSGQ_Instrument * retVal) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue.

OUT	MSGQ_Instrument *	retVal

Location to retrieve the instrumentation information.

Return Value

DSP_SOK	The instrumentation information has been successfully retrieved.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

None.

Constraints

This function is defined only if profiling is enabled within *DSPLINK*.

See Also

MSGQ_Instrument

7.3.16 MSGQ_debug

This function prints the current status of the MSGQ subcomponent.

Syntax

```
Void MSGQ_debug (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue.

Return Value

None.

Comments

None.

Constraints

This function is defined only for debug builds.

See Also

None.

8 PMGR

8.1 API Definition

8.1.1 PMGR_MSGQ_setup

This function initializes the MSGQ component.

Syntax

```
DSP_STATUS PMGR_MSGQ_setup () ;
```

Arguments

None.

Return Value

DSP_SOK	The messaging component has been successfully initialized.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function is called from PMGR_PROC_setup () for the first calling process. It passes down the call into the Link Driver layer.

Constraints

None.

See Also

LDRV_MSGQ_setup ()

8.1.2 PMGR_MSGQ_destroy

This function finalizes the MSGQ component.

Syntax

```
DSP_STATUS PMGR_MSGQ_destroy () ;
```

Arguments

None.

Return Value

DSP_SOK	The messaging component has been successfully finalized.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function is called from PMGR_PROC_destroy () for the last calling process. It passes down the call into the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

See Also

LDRV_MSGQ_destroy ()

8.1.3 PMGR_MSGQ_transportOpen

This function initializes the transport associated with the specified processor.

Syntax

```
DSP_STATUS PMGR_MSGQ_transportOpen (ProcessorId procId, Pvoid attrs) ;
```

Arguments

IN	ProcessorId	procId

ID of the Processor for which the transport is to be opened.

IN	Pvoid	attrs

Attributes for initialization of the transport. The structure of the expected attributes is specific to a transport.

Return Value

DSP_SOK	The MQT component has been successfully opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`attrs` must be valid.

See Also

```
MSGQ_transportOpen ()
LDRV_MSGQ_transportOpen ()
```

8.1.4 PMGR_MSGQ_transportClose

This function finalizes the transport associated with the specified processor.

Syntax

```
DSP_STATUS PMGR_MSGQ_transportClose (ProcessorId procId) ;
```

Arguments

IN	ProcessorId	procId

ID of the Processor for which the transport is to be closed.

Return Value

DSP_SOK	The MQT component has been successfully closed.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

See Also

```
MSGQ_transportClose ()
LDRV_MSGQ_transportClose ()
```

8.1.5 PMGR_MSGQ_open

This function opens the message queue to be used for receiving messages, identified through the specified message queue name.

Syntax

```
DSP_STATUS PMGR_MSGQ_open (Pstr        queueName,
                           MSGQ_Queue * msgqQueue,
                           MSGQ_Attrs * attrs) ;
```

Arguments

IN	Pstr	queueName

Name of the message queue to be opened.

OUT	MSGQ_Queue *	msgqQueue

Location to store the handle to the message queue.

IN OPT	MSGQ_Attrs *	attrs

Optional attributes for creation of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully created.
DSP_ENOTFOUND	Attempt to open more than number of message queues configured.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function updates ownership information for the MSGQ and passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

queueName must be valid.

msgqQueue must be a valid pointer.

See Also

MSGQ_open ()
LDRV_MSGQ_open ()

8.1.6 PMGR_MSGQ_close

This function closes the message queue identified by the specified MSGQ handle.

Syntax

```
DSP_STATUS PMGR_MSGQ_close (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MsgQueue	msgQueue

Handle to the message queue to be closed.

Return Value

DSP_SOK	The message queue has been successfully deleted.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EACCESSDENIED	Access denied. Only the client who had successfully opened the message queue may call this function.
DSP_EFAIL	General failure.

Comments

This function updates ownership information for the MSGQ and passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

msgqQueue must be a valid pointer.

Client must be the owner of the MSGQ.

See Also

```
MSGQ_close ()
LDRV_MSGQ_close ()
```

8.1.7 PMGR_MSGQ_locate

This function synchronously locates the message queue identified by the specified MSGQ name and returns a handle to the located message queue.

Syntax

```
DSP_STATUS PMGR_MSGQ_locate (Pstr              queueName,
                             MSGQ_Queue *       msgqQueue,
                             MSGQ_LocateAttrs * attrs) ;
```

Arguments

IN	Pstr	queueName

Name of the message queue to be located.

OUT	MSGQ_Queue *	msgqQueue

Location to store the handle to the located message queue.

IN OPT	MSGQ_LocateAttrs *	attrs

Optional attributes for location of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully located.
DSP_ENOTFOUND	The specified message queue could not be located.
DSP_ETIMEOUT	Timeout occurred while locating the MSGQ.
DSP_ENOTCOMPLETE	Operation not complete when WAIT_NONE was specified as timeout.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

queueName must be valid.

msgqQueue must be a valid pointer.

See Also

MSGQ_locate ()
LDRV_MSGQ_locate ()

8.1.8 PMGR_MSGQ_locateAsync

This function asynchronously locates the message queue identified by the specified MSGQ name. An attempt is made to asynchronously locate the message queue. If the message queue is found, an `MSGQ_AsyncLocateMsg` message is sent to the specified reply message queue.

Syntax

```
DSP_STATUS PMGR_MSGQ_locateAsync (Pstr                    queueName,
                                  MSGQ_Queue              replyQueue,
                                  MSGQ_LocateAsyncAttrs *  attrs) ;
```

Arguments

`IN` `Pstr` `queueName`

Name of the message queue to be located.

`IN` `MSGQ_Queue` `replyQueue`

Message queue to be used to receive the response message for asynchronous location.

`IN` `MSGQ_LocateAsyncAttrs *` `attrs`

Attributes for asynchronous location of the MSGQ.

Return Value

`DSP_SOK`	The message queue has been successfully located.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`queueName` must be valid.

`replyQueue` must be a valid pointer.

`attrs` must be a valid pointer.

See Also

`MSGQ_locateAsync ()`
`LDRV_MSGQ_locateAsync ()`

8.1.9 PMGR_MSGQ_release

This function releases the message queue identified by the MSGQ handle that was located earlier.

Syntax

```
DSP_STATUS PMGR_MSGQ_release (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue to be released.

Return Value

DSP_SOK	The message queue has been successfully released.
DSP_ENOTFOUND	The message queue has not been previously located.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

See Also

```
MSGQ_release ()
LDRV_MSGQ_release ()
```

8.1.10 PMGR_MSGQ_alloc

This function allocates a message, and returns the pointer to the user.

Syntax

```
DSP_STATUS PMGR_MSGQ_alloc (PoolId      poolId,
                            Uint16      size,
                            MSGQ_Msg *   msg) ;
```

Arguments

`IN` `PoolId` `poolId`

ID of the Pool to be used for allocating this message.

`IN` `Uint16` `size`

Size of the message to be allocated.

`OUT` `MSGQ_Msg *` `msg`

Location to receive the allocated message.

Return Value

`DSP_SOK`	The message has been successfully allocated.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EFAIL`	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msg` must be a valid pointer.

`size` must be a greater than size of `MSGQ_MsgHeader`.

See Also

`MSGQ_alloc ()`
`LDRV_MSGQ_alloc ()`

8.1.11 PMGR_MSGQ_free

This function frees a message.

Syntax

```
DSP_STATUS PMGR_MSGQ_free (MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Msg	msg

Pointer to the message to be freed.

Return Value

DSP_SOK	The message has been successfully freed.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msg` must be valid.

See Also

```
MSGQ_free ()
LDRV_MSGQ_free ()
```

8.1.12 PMGR_MSGQ_put

This function sends a message to the specified MSGQ.

Syntax

```
DSP_STATUS PMGR_MSGQ_put (MSGQ_Queue msgqQueue, MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the destination MSGQ.

IN	MSGQ_Msg	msg

Pointer to the message to be sent to the destination MSGQ.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

`msg` must be valid.

See Also

```
MSGQ_put ()
LDRV_MSGQ_put ()
```

8.1.13 PMGR_MSGQ_get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS PMGR_MSGQ_get (MSGQ_Queue   msgqQueue,
                          Uint32      timeout,
                          MSGQ_Msg *   msg) ;
```

Arguments

`IN` `MSGQ_Queue` `msgqQueue`

Handle to the MSGQ on which the message is to be received.

`IN` `Uint32` `timeout`

Timeout value to wait for the message (in milliseconds).

`OUT` `MSGQ_Msg *` `msg`

Location to receive the message.

Return Value

`DSP_SOK`	The message has been successfully received.
`DSP_ETIMEOUT`	Timeout occurred while receiving the message.
`DSP_ENOTCOMPLETE`	Operation not complete when `WAIT_NONE` was specified as timeout.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

None.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

`msg` must be a valid pointer.

See Also

`MSGQ_get ()`
`LDRV_MSGQ_get ()`

8.1.14 PMGR_MSGQ_count

This function returns the count of the number of messages in a local message queue.

Syntax

```
DSP_STATUS PMGR_MSGQ_count (MSGQ_Queue msgqQueue, Uint16 * count) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the MSGQ for which the count is to be retrieved.

OUT	Uint16 *	count

Location to receive the message count.

Return Value

DSP_SOK	The count has been successfully retrieved.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

None.

Constraints

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

`count` must be a valid pointer.

See Also

```
MSGQ_count ()
LDRV_MSGQ_count ()
```

8.1.15 PMGR_MSGQ_setErrorHandler

This function allows the user to designate a MSGQ as an error-handler MSGQ to receive asynchronous error messages from the transports.

Syntax

```
DSP_STATUS PMGR_MSGQ_setErrorHandler (MSGQ_Queue errorQueue,
                                      PoolId    poolId) ;
```

Arguments

IN	MSGQ_Queue	errorQueue

Handle to the message queue to receive the error messages.

IN	PoolId	poolId

ID indicating the pool to be used for allocating the error messages.

Return Value

DSP_SOK	The error handler has been successfully set.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

PMGR MSGQ component must be initialized before calling this function.

See Also

MSGQ_setErrorHandler
LDRV_MSGQ_setErrorHandler

8.1.16 PMGR_MSGQ_instrument

This function gets the instrumentation information related to the specified message queue.

Syntax

```
DSP_STATUS PMGR_MSGQ_instrument (MSGQ_Queue        msgqQueue,
                                 MSGQ_Instrument * retVal) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue.

OUT	MSGQ_Instrument *	retVal

Location to retrieve the instrumentation information.

Return Value

DSP_SOK	The instrumentation information has been successfully retrieved.
DSP_EFAIL	General failure.

Comments

This function passes on the call from the API layer to the Link Driver layer.

Constraints

This function is defined only if profiling is enabled within *DSPLINK*.

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

`retVal` must be a valid pointer.

See Also

```
MSGQ_Instrument
MSGQ_instrument
LDRV_MSGQ_instrument
```

8.1.17 PMGR_MSGQ_debug

This function prints the status of the MSGQ subcomponent.

Syntax

```
Void PMGR_MSGQ_debug (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue	

Handle to the message queue.

Return Value

None.

Comments

This function prints any status of the MSGQ subcomponent contained within the PMGR layer, and passes down the call into the LDRV layer.

Constraints

This function is defined only for debug builds.

PMGR MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

See Also

`MSGQ_debug`
`LDRV_MSGQ_debug`

9 LDRV MSGQ

9.1 Typedefs & Data Structures

9.1.1 FnMqtInitialize

This type defines the MQT initialization function.

Definition

```
typedef Void (*FnMqtInitialize) () ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

MQT_Interface

9.1.2 FnMqtFinalize

This type defines the MQT finalization function.

Definition

```
typedef Void (*FnMqtFinalize) () ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.3 FnMqtOpen

This type defines the MQT open function.

Definition

```
typedef DSP_STATUS (*FnMqtOpen) (LDRV_MSGQ_TransportHandle mqtHandle,
                                 Pvoid                     mqtAttrs) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.4 FnMqtClose

This type defines the MQT close function.

Definition

```
typedef DSP_STATUS (*FnMqtClose) (LDRV_MSGQ_TransportHandle mqtHandle)
;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.5 FnMqtLocate

This type defines the MQT function for locating a MSGQ identified by the specified MSGQ name.

Definition

```
typedef DSP_STATUS (*FnMqtLocate) (LDRV_MSGQ_TransportHandle mqtHandle,
                                   Pstr                      queueName,
                                   Bool                      sync,
                                   MSGQ_Queue *              msgqQueue,
                                   Pvoid                     attrs) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.6 FnMqtRelease

This type defines the MQT function for releasing a MSGQ identified by the MSGQ handle that was located earlier.

Definition

```
typedef
DSP_STATUS (*FnMqtRelease) (LDRV_MSGQ_TransportHandle mqtHandle,
                            MSGQ_Queue                msgqQueue) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.7 FnMqtPut

This type defines the MQT function for sending a message.

Definition

```
typedef DSP_STATUS (*FnMqtPut) (LDRV_MSGQ_TransportHandle mqtHandle,
                                MSGQ_Msg                  msg) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MQT_Interface`

9.1.8 FnMqtDebug

This type defines the MQT function for printing debug information.

Definition

```
typedef
DSP_STATUS (*FnMqtDebug) (LDRV_MSGQ_TransportHandle mqtHandle) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

This type is only defined if debugging is enabled.

See Also

`MQT_Interface`

9.1.9 LDRV_MSGQ_State

This structure defines the MSGQ state object. It includes all global information required by the MSGQ component.

Definition

```
typedef struct LDRV_MSGQ_State_tag {
    LDRV_MSGQ_Handle *       msgqHandles ;
    Uint16                   maxMsgqs ;
    Uint16                   numDsps ;
    LDRV_MSGQ_TransportObj * transports ;
    Bool                     doPowerCtrl [MAX_DSPS] ;
    MSGQ_Queue               errorQueue ;
    PoolId                   errorPoolId ;
} LDRV_MSGQ_State ;
```

Fields

`msgqHandles`	Array of handles to message queue objects.
`maxMsgqs`	Maximum number of message queues on the GPP.
`numDsps`	Number of DSPs in the system.
`transports`	Array of transport objects, one for every processor in the system.
`doPowerCtrl`	Indicates whether power control of the DSPs should be done within DSPLINK.
`errorQueue`	Handle to the MSGQ registered by the user as an error handler. If no error handler MSGQ has been registered by the user, the value of this field is `MSGQ_INVALIDMSGQ`.
`errorPoolId`	ID of the Pool to be used for allocating the asynchronous error messages, if the user has registered an error handler MSGQ. If no error handler MSGQ has been registered by the user, the value of this field is `POOL_INVALIDID`.

Comments

The MSGQ state object is filled with information extracted from the CFG during the call to `LDRV_MSGQ_setup ()`.

Constraints

None.

See Also

`LDRV_MSGQ_setup ()`

9.1.10 LDRV_MSGQ_Object

This structure defines the MSGQ object. It includes all information specific to a particular MSGQ.

Definition

```
struct LDRV_MSGQ_Object_tag {
    Char8            name [DSP_MAX_STRLEN] ;
    MSGQ_Queue       msgqQueue ;
    List *           queue ;
    Pvoid            ntfyHandle ;
    MsgqPend         pend ;
    MsgqPost         post ;
    Bool             defaultNtfyHandle ;
#if defined (DDSP_PROFILE)
    MSGQ_Stats       msgqStats ;
#endif /* if defined (DDSP_PROFILE) */
} ;
```

Fields

`name`	System-wide unique message queue name.
`msgqQueue`	Message queue handle.
`queue`	Queue of received messages.
`ntfyHandle`	Pointer to the notification object for the message queue.
`pend`	Function to be used to wait to receive a message.
`post`	Function to be used to indicate arrival of a message.
`defaultNtfyHandle`	Indicates whether the notify handle in the message queue object was created internally.
`msgqStats`	Instrumentation information for the Message Queue. Defined only if profiling is enabled.

Comments

The MSGQ object is created during the `MSGQ_open ()` function.

The default notify handle used internally within the MSGQ object is a binary semaphore.

Constraints

None.

See Also

`LDRV_MSGQ_State`
`MSGQ_open ()`

9.1.11 LDRV_MSGQ_TransportObj

This structure defines the common attributes of the transport object. There is one instance of the transport object per MQT in the system.

Definition

```
struct LDRV_MSGQ_TransportObj_tag {
    MQT_Interface *        mqtInterface ;
    Pvoid                  object ;
    ProcessorId            dspId ;
} ;
```

Fields

`mqtInterface`	Pointer to the function table of the MQT represented by the transport object.
`object`	Transport-specific object.
`dspId`	Processor identifier.

Comments

The LDRV MSGQ component maintains an array of the MSGQ transport objects. These are used to identify the MQTs existing in the system.

The transport objects are initialized during `LDRV_MSGQ_setup ()` through configuration information obtained from the CFG. One MSGQ transport object is configured for every processor in the system. The MQT state information is filled in during `LDRV_MSGQ_transportOpen ()`.

Constraints

None.

See Also

`LDRV_MSGQ_setup ()`
`LDRV_MSGQ_transportOpen ()`

9.1.12 MQT_Interface

This structure defines the function pointer table that must be implemented for every MQT in the system.

Definition

```
typedef struct MQT_Interface_tag {
    FnMqtInitialize   initialize ;
    FnMqtFinalize     finalize   ;
    FnMqtOpen         open       ;
    FnMqtClose        close      ;
    FnMqtLocate       locate     ;
    FnMqtRelease      release    ;
    FnMqtPut          put        ;
#if defined (DDSP_DEBUG)
    FnMqtDebug        debug      ;
#endif /* defined (DDSP_DEBUG) */
} MQT_Interface ;
```

Fields

`initialize`	Pointer to MQT initialization function.
`finalize`	Pointer to MQT finalization function.
`open`	Pointer to MQT open function.
`close`	Pointer to MQT close function.
`locate`	Pointer to MQT function for locating a MSGQ.
`release`	Pointer to MQT function for releasing a MSGQ.
`put`	Pointer to MQT function for sending a message.
`debug`	Pointer to MQT debug function.

Comments

Each MQT in the system must implement a set of functions with defined interfaces. These functions must then be exported through a function pointer table, of type `MQT_Interface`.

Constraints

None.

See Also

`None.`

9.1.13 LDRV_MQT_Config

This structure defines the MQT object stored in the LDRV object.

Definition

```
typedef struct LDRV_MQT_Config_tag {
    Uint32             maxMsgSize ;
    Uint32             ipsId ;
    Uint32             ipsEventNo ;
    Uint32             arg1 ;
    Uint32             arg2 ;
} LDRV_MQT_Config ;
```

Fields

`maxMsgSize`	Maximum size of message supported by MQT. May be -1 if there is no limit on maximum message size for the MQT.
`ipsId`	ID of the IPS to be used (if any). A value of -1 indicates that no IPS is required by the MQT.
`ipsEventNo`	IPS Event number associated with MQT (if any). A value of -1 indicates that no IPS is required by the MQT.
`arg1`	First optional argument for this MQT. The significance of this argument is specific to the MQT.
`arg2`	Second optional argument for this MQT. The significance of this argument is specific to the MQT.

Comments

An array of MQT objects is maintained within the LDRV_MQT module. These hold all MQT information obtained through the CFG.

Constraints

None.

See Also

None.

9.2 API Definition

9.2.1 LDRV_MSGQ_setup

This function initializes the MSGQ component.

Syntax

```
DSP_STATUS LDRV_MSGQ_setup () ;
```

Arguments

None.

Return Value

DSP_SOK	The messaging component has been successfully initialized.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function initializes the MSGQ component. It sets up the MSGQ state object with information obtained from the LDRV object. It also initializes the individual MQTs configured in the system.

Constraints

The LDRV_MSGQ component must not be initialized.

See Also

LDRV_MSGQ_destroy ()

9.2.2 LDRV_MSGQ_destroy

This function finalizes the MSGQ component.

Syntax

```
DSP_STATUS LDRV_MSGQ_destroy () ;
```

Arguments

None.

Return Value

DSP_SOK	The messaging component has been successfully finalized.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function finalizes the MSGQ component. It also finalizes the individual MQTs configured in the system.

Constraints

The LDRV_MSGQ component must be initialized.

See Also

LDRV_MSGQ_setup ()

9.2.3 LDRV_MSGQ_transportOpen

This function initializes the transport associated with the specified processor.

Syntax

```
DSP_STATUS LDRV_MSGQ_transportOpen (ProcessorId procId, Pvoid attrs) ;
```

Arguments

IN	ProcessorId	procId

ID of the Processor for which the transport is to be opened.

IN	Pvoid	attrs

Attributes for initialization of the transport. The structure of the expected attributes is specific to a transport.

Return Value

DSP_SOK	The MQT component has been successfully opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function calls the open () function of the MQT identified through the processor ID. It initializes the MQT using the provided attributes.

The static configuration of the MQTs is done as part of the CFG. This includes configuration of the fixed attributes specific to each MQT, including its function table interface.

Constraints

The LDRV_MSGQ component must be initialized.

attrs must be valid.

procId must be valid.

See Also

LDRV_MSGQ_transportClose ()

9.2.4 LDRV_MSGQ_transportClose

This function finalizes the transport associated with the specified processor.

Syntax

```
DSP_STATUS LDRV_MSGQ_transportClose (ProcessorId procId) ;
```

Arguments

IN	ProcessorId	procId

ID of the Processor for which the transport is to be closed.

Return Value

DSP_SOK	The MQT component has been successfully closed.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function calls the `close ()` function of the MQT identified through the processor ID.

Constraints

The LDRV_MSGQ component must be initialized.

`procId` must be valid.

See Also

`LDRV_MSGQ_transportOpen ()`

9.2.5 LDRV_MSGQ_open

This function opens the message queue to be used for receiving messages, identified through the specified message queue name.

Syntax

```
DSP_STATUS LDRV_MSGQ_open (Pstr         queueName,
                           MSGQ_Queue * msgqQueue,
                           MSGQ_Attrs * attrs) ;
```

Arguments

IN	Pstr	queueName

Name of the message queue to be created.

OUT	MSGQ_Queue *	msgqQueue

Optional attributes for creation of the MSGQ.

IN OPT	MSGQ_Attrs *	attrs

Location to store the handle to the message queue.

Return Value

DSP_SOK	The message queue has been successfully created.
DSP_ENOTFOUND	Attempt to open more than number of message queues configured.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function creates and initializes an instance of the LDRV_MSGQ_Object object representing a local message queue.

Constraints

The LDRV_MSGQ component must be initialized.

queueName must be valid.

msgqQueue must be valid.

See Also

MSGQ_Queue
MSGQ_Attrs
LDRV_MSGQ_close ()
LDRV_MSGQ_locate ()

9.2.6 LDRV_MSGQ_close

This function closes the message queue identified by the specified MSGQ handle.

Syntax

```
DSP_STATUS LDRV_MSGQ_close (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue to be deleted.

Return Value

DSP_SOK	The message queue has been successfully deleted.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function deletes the instance of the LDRV_MSGQ_Object object represented by the specified message queue handle.

Constraints

The LDRV_MSGQ component must be initialized.

msgqQueue must be valid.

See Also

MSGQ_Queue
LDRV_MSGQ_open ()

9.2.7 LDRV_MSGQ_locate

This function synchronously locates the message queue identified by the specified MSGQ name and returns a handle to the located message queue.

Syntax

```
DSP_STATUS LDRV_MSGQ_locate (Pstr               queueName,
                             MSGQ_Queue *       msgqQueue,
                             MSGQ_LocateAttrs * attrs) ;
```

Arguments

`IN` `Pstr` `queueName`

Name of the message queue to be located.

`OUT` `MSGQ_Queue *` `msgqQueue`

Location to store the handle to the located message queue.

`IN OPT` `MSGQ_LocateAttrs *` `attrs`

Optional attributes for location of the MSGQ.

Return Value

`DSP_SOK`	The message queue has been successfully located.
`DSP_ENOTFOUND`	The specified message queue could not be located.
`DSP_ETIMEOUT`	Timeout occurred while locating the MSGQ.
`DSP_ENOTCOMPLETE`	Operation not complete when `WAIT_NONE` was specified as timeout.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This function searches within its own list of MSGQs and interacts with the remote MQTs to locate the MSGQ as specified by the user. If not found locally, the call passes down to the remote MQTs.

Constraints

The LDRV_MSGQ component must be initialized.

`queueName` must be valid.

`msgqQueue` must be a valid pointer.

See Also

```
MSGQ_Queue
MSGQ_LocateAttrs
LDRV_MSGQ_put ()
LDRV_MSGQ_release ()
```

9.2.8 LDRV_MSGQ_locateAsync

This function asynchronously locates the message queue identified by the specified MSGQ name. An attempt is made to asynchronously locate the message queue. If the message queue is found, an MSGQ_AsyncLocateMsg message is sent to the specified reply message queue.

Syntax

```
DSP_STATUS LDRV_MSGQ_locateAsync (Pstr                    queueName,
                                  MSGQ_Queue              replyQueue,
                                  MSGQ_LocateAsyncAttrs *  attrs) ;
```

Arguments

IN	Pstr	queueName

Name of the message queue to be located.

IN	MSGQ_Queue	replyQueue

Location to store the handle to the located message queue.

IN	MSGQ_LocateAsyncAttrs *	attrs

Optional attributes for location of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully located.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function first searches within its own list of MSGQs. If not found locally, it sends an asynchronous locate request to all the remote MQTs. The remote MQT that is able to successfully locate the message queue sends an MSGQ_AsyncLocateMsg message to the reply message queue specified by the user. If the message queue was not found in the system, no message is sent to the reply message queue.

Constraints

The LDRV_MSGQ component must be initialized.

queueName must be valid.

replyQueue must be valid.

attrs must be valid.

See Also

MSGQ_Queue
MSGQ_LocateAsyncAttrs
LDRV_MSGQ_put ()
LDRV_MSGQ_release ()

9.2.9 LDRV_MSGQ_release

This function releases the message queue identified by the MSGQ handle that was located earlier.

Syntax

```
DSP_STATUS LDRV_MSGQ_release (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue to be released.

Return Value

DSP_SOK	The message queue has been successfully released.
DSP_ENOTFOUND	The message queue was not previously located.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function releases the MSGQ as specified by the user. If not local, the call passes down to the remote MQTs.

Constraints

The LDRV_MSGQ component must be initialized.

msgqQueue must be valid.

See Also

MSGQ_Queue
LDRV_MSGQ_locate ()

9.2.10 LDRV_MSGQ_alloc

This function allocates a message, and returns the pointer to the user.

Syntax

```
DSP_STATUS LDRV_MSGQ_alloc (PoolId      poolId,
                            Uint16      size,
                            MSGQ_Msg *  msg) ;
```

Arguments

`IN` `PoolId` `poolId`

ID of the Pool to be used for allocating this message.

`IN` `Uint16` `size`

Size (in bytes) of the message to be allocated.

`OUT` `MSGQ_Msg *` `msg`

Location to receive the allocated message.

Return Value

`DSP_SOK`	The message has been successfully allocated.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EFAIL`	General failure.

Comments

This function interacts with the specified pool to allocate a message of specified size.

Constraints

The LDRV_MSGQ component must be initialized.

`msg` must be a valid pointer.

`size` must be greater than size of `MSGQ_MsgHeader`.

See Also

`MSGQ_MsgHeader`
`LDRV_MSGQ_put ()`
`LDRV_MSGQ_free ()`

9.2.11 LDRV_MSGQ_free

This function frees a message.

Syntax

```
DSP_STATUS LDRV_MSGQ_free (MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Msg	msg

Pointer to the message to be freed.

Return Value

DSP_SOK	The message has been successfully freed.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function interacts with the MQA to free the specified message. The MQA to be used, and all other information required for freeing the message, such as size of the message, are obtained from the message header.

Constraints

The LDRV_MSGQ component must be initialized.

msg must be valid.

See Also

```
MSGQ_MsgHeader
LDRV_MSGQ_get ()
LDRV_MSGQ_alloc ()
```

9.2.12 LDRV_MSGQ_put

This function sends a message to the specified MSGQ.

Syntax

```
DSP_STATUS LDRV_MSGQ_put (MSGQ_Queue msgqQueue, MSGQ_Msg msg) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the destination MSGQ.

IN	MSGQ_Msg	msg

Pointer to the message to be sent to the destination MSGQ.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_ENOTFOUND	The message queue does not exist.
DSP_EFAIL	General failure.

Comments

This function sends the message to the destination MSGQ. If the MSGQ is not local, the call passes down to the remote MQTs.

Constraints

The LDRV_MSGQ component must be initialized.

msgqQueue must be valid.

msg must be valid.

See Also

MSGQ_Queue
MSGQ_MsgHeader
LDRV_MSGQ_get ()

9.2.13 LDRV_MSGQ_get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS LDRV_MSGQ_get (MSGQ_Queue    msgqQueue,
                          Uint32        timeout,
                          MSGQ_Msg *    msg) ;
```

Arguments

`IN` `MSGQ_Queue` `msgqQueue`

Handle to the MSGQ on which the message is to be received.

`IN` `Uint32` `timeout`

Timeout value to wait for the message (in milliseconds).

`OUT` `MSGQ_Msg *` `msg`

Location to receive the message.

Return Value

`DSP_SOK`	The message has been successfully received.
`DSP_ETIMEOUT`	Timeout occurred while receiving the message.
`DSP_ENOTCOMPLETE`	Operation not complete when `WAIT_NONE` was specified as timeout.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This function queues up the received message on the appropriate MSGQ.

Constraints

The LDRV_MSGQ component must be initialized.

`msgqQueue` must be valid.

`msg` must be a valid pointer.

See Also

`MSGQ_Queue`
`MSGQ_MsgHeader`
`LDRV_MSGQ_put ()`

9.2.14 LDRV_MSGQ_count

This function returns the count of the number of messages in a local message queue.

Syntax

```
DSP_STATUS PMGR_MSGQ_count (MSGQ_Queue msgqQueue, Uint16 * count) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the MSGQ for which the count is to be retrieved.

OUT	Uint16 *	count

Location to receive the message count.

Return Value

DSP_SOK	The count has been successfully retrieved.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This function traverses the list within the Message Queue object and returns the count of the number of messages queued within the list to the caller.

Constraints

LDRV MSGQ component must be initialized before calling this function.

`msgqQueue` must be valid.

`count` must be a valid pointer.

See Also

`MSGQ_Queue`

9.2.15 LDRV_MSGQ_setErrorHandler

This function allows the user to designate a MSGQ as an error-handler MSGQ to receive asynchronous error messages from the transports.

Syntax

```
DSP_STATUS LDRV_MSGQ_setErrorHandler (MSGQ_Queue  errorQueue,
                                      PoolId      poolId) ;
```

Arguments

`IN` `MSGQ_Queue` `errorQueue`

Handle to the message queue to receive the error messages.

`IN` `PoolId` `poolId`

ID indicating the pool to be used for allocating the error messages.

Return Value

`DSP_SOK`	The error handler has been successfully set.
`DSP_EFAIL`	General failure.

Comments

This function registers the error handler MSGQ within its state object. After the error handler MSGQ has been set, the MSGQ component responds to `LDRV_MSGQ_sendErrorMsg ()` calls from the transport by allocating and sending the appropriate asynchronous error message to the error handler MSGQ.

Constraints

The error handler MSGQ must be created before this API can be called.

The LDRV_MSGQ component must be initialized.

`errorQueue` must be valid.

See Also

`MSGQ_AsyncErrorMsg`
`LDRV_MSGQ_sendErrorMsg ()`

9.2.16 LDRV_MSGQ_instrument

This function gets the instrumentation information related to the specified message queue.

Syntax

```
DSP_STATUS LDRV_MSGQ_instrument (MSGQ_Queue        msgqQueue,
                                 MSGQ_Instrument * retVal) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue

Handle to the message queue.

OUT	MSGQ_Instrument *	retVal

Location to retrieve the instrumentation information.

Return Value

DSP_SOK	The instrumentation information has been successfully retrieved.
DSP_EFAIL	General failure.

Comments

None.

Constraints

This function is defined only if profiling is enabled within *DSPLINK*.

The LDRV_MSGQ component must be initialized.

msgqQueue must be valid.

retVal must be a valid pointer.

See Also

MSGQ_Instrument

9.2.17 LDRV_MSGQ_debug

This function prints the status of the MSGQ subcomponent.

Syntax

```
Void LDRV_MSGQ_debug (MSGQ_Queue msgqQueue) ;
```

Arguments

IN	MSGQ_Queue	msgqQueue	

Handle to the message queue.

Return Value

None.

Comments

None.

Constraints

This function is defined only for debug builds.

`msgqQueue` must be valid.

See Also

None.

9.2.18 LDRV_MSGQ_locateLocal

This function locates a local message queue identified by the specified MSGQ name and returns a handle to the located message queue if found.

Syntax

```
DSP_STATUS LDRV_MSGQ_locateLocal (Pstr         queueName,
                                  MSGQ_Queue * msgqQueue) ;
```

Arguments

IN	Pstr	queueName	
	Name of the message queue to be located.		
OUT	MSGQ_Queue *	msgqQueue	
	Location to store the handle to the located message queue.		

Return Value

DSP_SOK	The specified message queue was successfully located.
DSP_ENOTFOUND	The specified message queue could not be located.
DSP_EFAIL	General failure.

Comments

This function searches within the local MSGQ list for the specified message queue identified by its name.

This function is called internally by the LDRV MSGQ component and the transports.

Constraints

queueName must be valid.

msgqQueue must be valid.

See Also

MSGQ_Queue

9.2.19 LDRV_MSGQ_sendErrorMsg

This function sends an asynchronous error message of a particular type to the user-defined error handler MSGQ.

Syntax

```
DSP_STATUS LDRV_MSGQ_sendErrorMsg (MSGQ_MqtError  errorType,
                                   Pvoid          arg1,
                                   Pvoid          arg2) ;
```

Arguments

IN	MSGQ_MqtError	errorType

Type of the error.

IN	Pvoid	arg1

First argument dependent on the error type.

IN	Pvoid	arg2

Second argument dependent on the error type.

Return Value

DSP_SOK	The error message has been successfully sent.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This function sends an error message to the user-defined error handler MSGQ. It is called by the transports on occurrence of any of a set of predefined asynchronous errors.

This function is called internally by the transports.

Constraints

This function sends an error message only if the user has registered an error handler MSGQ through a call to the MSGQ_setErrorHandler () function.

See Also

MSGQ_MqtError
MSGQ_AsyncErrorMsg
LDRV_MSGQ_setErrorHandler ()

9.2.20 LDRV_MSGQ_notImpl

Represents a function that is not implemented and returns status accordingly.

Syntax

```
DSP_STATUS LDRV_MSGQ_notImpl () ;
```

Arguments

None.

Return Value

DSP_ENOTIMPL	This function is not implemented.

Comments

This function should be used in interface tables where some functions are not being implemented.

Constraints

None.

See Also

MQT_Interface