DSP/BIOS™ LINK

MESSAGING COMPONENT

LNK 031 DES

Version 1.10

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

TABLE OF FIGURES

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
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.
O	This bullet indicates important information. Please read such text carefully.
q	This bullet indicates additional information.

1.3 References

1.	LNK 001 PRD	DSP/BIOS™ LINK Product Requirements Document Version 1.00, dated JUN 12, 2002
2.	LNK 002 ARC	DSP/BIOS™ LINK High Level Architecture Version 1.02, dated JUL 15, 2003
3.		Messaging Comp. Design Document Version 1.00, dated NOV 05, 2003
4.		MSGQ Module: DSP/BIOS Support for Variable Length Messaging (SPRA987)

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 elaborates the low-level design details for the messaging component.

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:

R27	Provide logical connectivity between the GPP and DSP clients.
R28	The messages shall be transferred at a higher priority than data channels when only one HW medium is available.
R29	DSP/BIOS™ LINK shall use the services from the MSGQ module on BIOS for exchanging messages between clients executing on GPP and DSP.
R30	Messages of fixed length and variable length shall be supported.
R31	Messaging shall work transparently over varied links between GPP & DSP.

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. 3 and Ref. 4].
- § 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.

- The user configures the maximum message size that the remote MQT can transfer. The user must ensure that the size of messages transferred between the GPP and DSP is less than or equal to the MQT maximum message size.
- 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 allocator 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 allocator 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 queues on the DSP used for inter-processor transfer through DSPLINK must be created with specific names expected by the DSPLINK MQT. The names must be of the following format:

  `DSPLINK_DSP<PROCESSORID>MSGQ<MSGQID>`

  `<MSGQID>` can have values from `00` to `'n-1'` where `'n'` is the maximum number of message queues on the processor.
  `<PROCESSORID>` corresponds to the processor id of the DSP (used on GPP side to reference each DSP). The processor ID ranges in value from 00 to ‘n-1’, where ‘n’ is the maximum number of DSPs in the system.

  The message queues on the GPP used for inter-processor transfer through DSPLINK must be located by DSP applications with specific names expected by the DSPLINK MQT. The names must be of the following format:

  `DSPLINK_GPPMSGQ<MSGQID>`

  `<MSGQID>` can have values from `00` to `'n-1'` where `'n'` is the maximum number of message queues on the GPP.
- The message queue names must be unique over the complete system. This includes message queues created across all processors in the system.
- The default allocator 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 the processor identifier, and a message queue identifier.

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

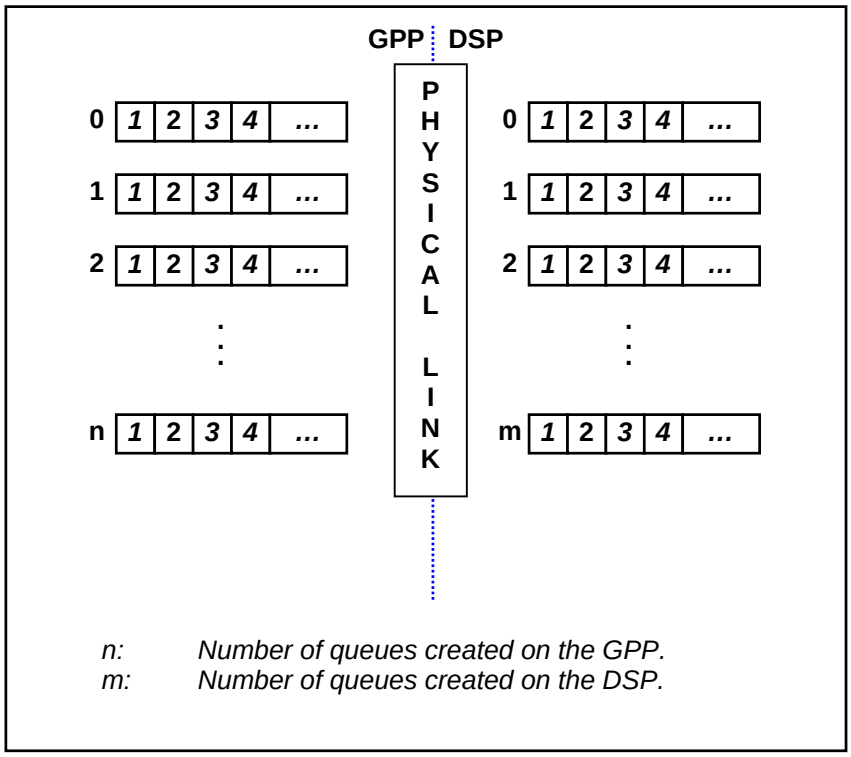

Figure 1. Messaging in *DSPLINK*.

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 allocators 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 the GPP-side, a MSGQ ID is used to identify a message queue. The ID is unique on a single processor. In addition, specification of the processor ID allows unique identification of MSGQs spread over multiple processors in the system.

On the DSP-side, a unique name is used for identifying a MSGQ. A fixed message name prefix is appended to the message ID to generate the MSGQ name on the DSP-side. This name is unique over all processors in the system.

When extended to multiple processors, each processor shall have a different name prefix for the message names, ensuring their uniqueness over the complete system.

Initialization and finalization

Before using any of the messaging features, the user must initialize the MSGQ component. This involves initialization of the MSGQ component followed by that of the individual MQTs and MQAs.

When the messaging services are no longer required, the user can finalize the individual MQTs and MQAs, followed by the global finalization of the MSGQ component.

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

For sending a message to the 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 reply MSGQ. On receiving the message, the receiver may extract information about the reply MSGQ from the message, 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 module has a two-level architecture. The first level is the MSGQ API. The second level is the Message Queue Transport interface (MQT) and Message Queue Allocator interface (MQA).

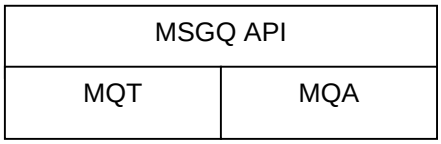

Figure 2. MSGQ component hierarchy

For further details, please refer to the MSGQ design document [Ref. 3].

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

5.2.1 Component interaction

The component interaction diagram gives an overview of the interaction of the messaging component with existing components within DSPLINK. It also specifies the sub-components involved in messaging.

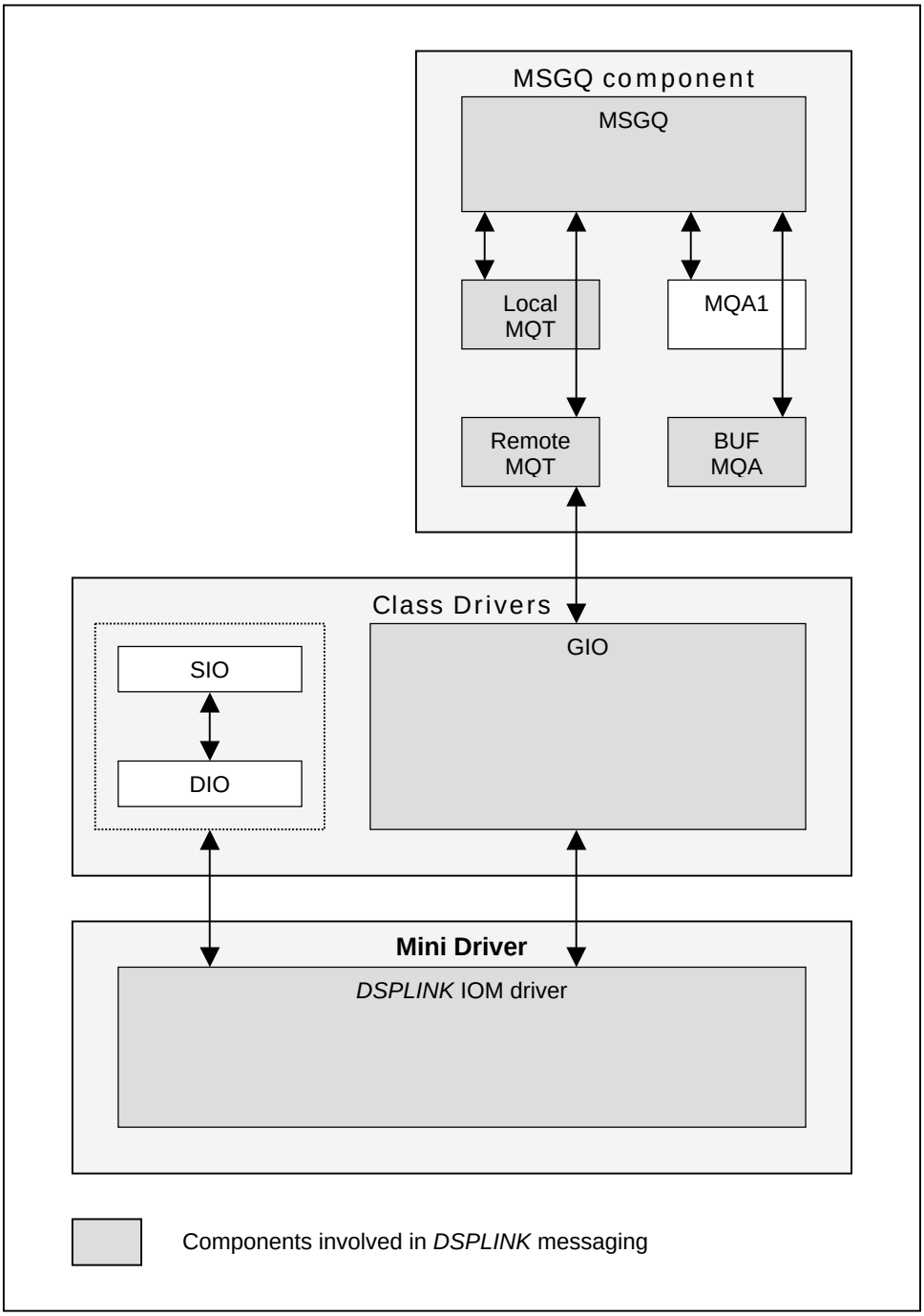

Figure 3. DSP-side component interaction diagram

5.2.2 Overview

The DSP-side messaging component shall utilize the services of the IOM link driver, through GIO API calls.

The MQT shall not implement the class driver functionality, but shall use the functionality provided by the GIO class driver. The IOM link driver shall not be modified to implement additional commands for messaging. The MQT shall treat the driver as a low-level driver, which provides the READ & WRITE functionality.

The IOM driver shall not interpret the contents of the packet that is sent to it. It shall only transfer the packet on the specified channel. The complete protocol for the messaging shall be present within the MQT. In this case, two channels (outgoing & incoming) shall be reserved for the messaging path.

This approach provides good portability, since the IOM driver shall be agnostic of the messaging protocol. The IOM driver does not need to be modified, and all the protocol required is only within the MQT.

It also allows for a simpler MQT, which shall only need to implement the MQT protocol, and shall not need to implement class driver functionality of buffer management and synchronization.

This alternative is also suited for transport links such as HPI, for which it may not be possible to have a separate physical-layer protocol for messaging.

To give higher priority to messaging, the messaging channels shall always be checked first for data availability. Only if no messages are available for transfer, the data channels shall be checked in round-robin fashion.

5.2.3 Details

This section elaborates the high-level design of the remote MQT and the buffer MQA for communication with the GPP.

Remote MQT

The remote MQT shall implement the transport protocol for communication with its counterpart on the GPP. The MQT shall create and manage the two channels to be used for messaging with 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 multiplexed onto a single channel to the GPP.
- `mqtPut ()` is deterministic, and shall return immediately. However, actual transfer of the message to the GPP may complete at a later time.
- Messages received from the GPP, intended for different MSGQs, are demultiplexed from a single channel from the GPP.
- Messages of varying sizes are appropriately handled, with minimum wastage of memory.

To ensure the above requirements, the MQT shall be configured with the maximum size of messages in the system. The MQT shall always ensure that there is at least one buffer issued to the receiver channel (from GPP), so that the GPP is never blocked for sending messages. The size of the issued buffer shall be large enough to accommodate any message arriving from the GPP.

Buffer MQA

The MQA must not allocate memory dynamically, since the `mqaAlloc ()` and `mqaFree ()` calls may be made from an HWI or SWI context.

The MQA shall manage a set of pools of fixed-size buffers. The configuration of the MQA shall be done at the time of initialization, through a set of parameters. These

parameters shall define the number of buffer pools, size of each pool, and any additional information that may be required.

The figure below shows the architecture of the MQA:

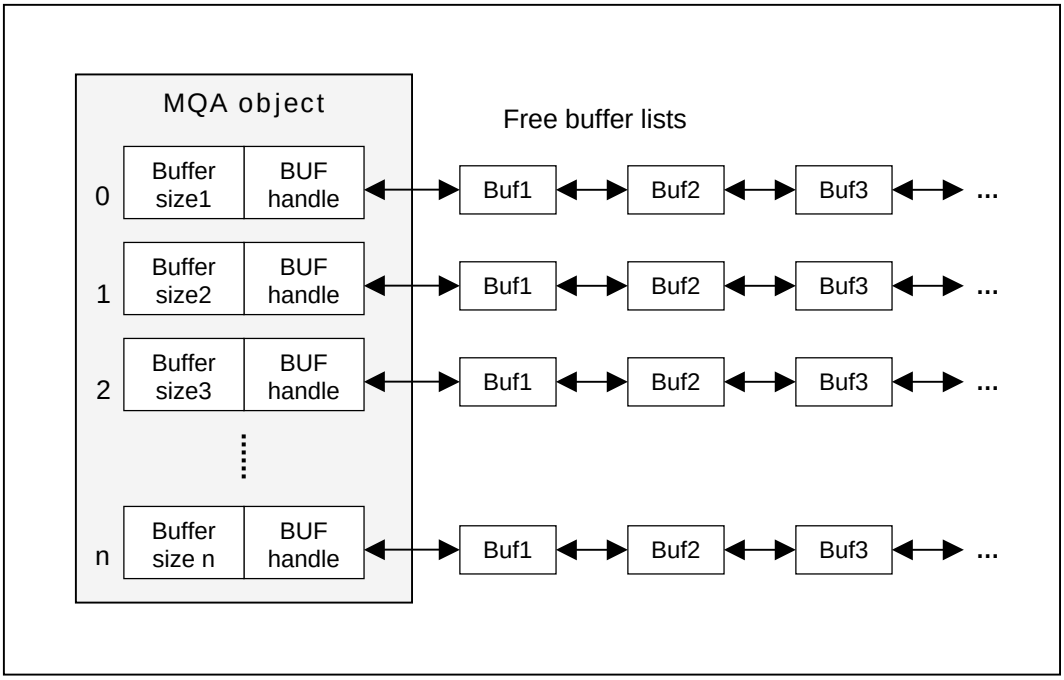

Figure 4. Buffer MQA architecture

On receiving an allocation request, the MQA shall search through the list of buffer sizes to find a buffer of the requested size. An available buffer shall then be removed from the corresponding free list, and returned to the user. If the list is empty, a NULL pointer shall be returned.

When a buffer is to be freed, the MQA shall find the buffer list of the appropriate size, and add the buffer to the free list.

The BUF module within DSP/BIOS™ shall be used for implementation of the buffer pools.

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 MQT for communication with the GPP, as well as the MQA required.

A local MQT shall also be implemented for managing the local message queues. This also enables messaging within the GPP, if applications so desire.

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 messaging component with existing components within DSPLINK. It also specifies the sub-components involved in messaging.

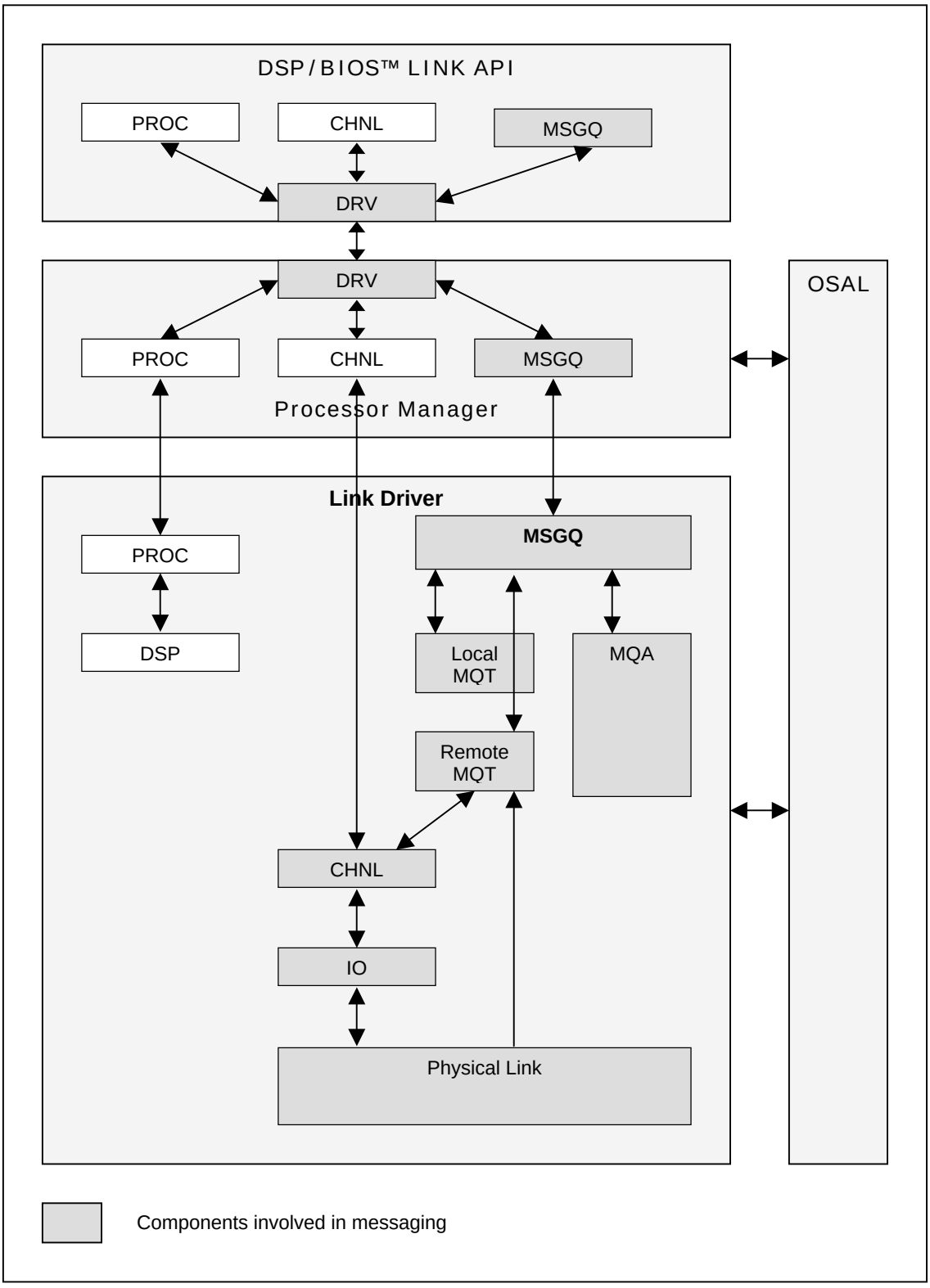

Figure 5. GPP-side component interaction diagram

5.3.2 Overview

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

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
- Local and remote MQTs (Message Queue Transport Interfaces)
- MQA (Message Queue Allocator Interface)
- Link-specific protocol (For example SHM messaging protocol)

5.3.2.4 OSAL

The CFG sub-component shall be enhanced to include configuration information for the MSGQ component.

This includes the following:

- Configuration of the different MQTs in the system
- Configuration of the different MQAs in the system

5.3.3 Details

CFG

The CFG shall contain statically configured information for the MSGQ component.

The driver object shall contain information about the number of MQAs and MQTs in the system. It shall also contain a Local MQT ID field, which corresponds to one of the MQTs that shall be configured as part of the CFG.

```
[DRIVER]
...
NUMMQAS          | N |   1
NUMMQTS          | N |   2
LOCALMQT         | N |   0
[/DRIVER]
```

MQAs shall be configured statically.

```
[MQA0]
NAME             | S |   MQABUF
INTERFACE        | A |   MQABUF_Interface
[/MQA0]
```

MQTs shall be configured statically. Each MQT shall specify the physical link ID used by it for communicating with the remote processor.

The physical link ID field shall be ignored for the local MQT.

```
[MQT0]
NAME             | S |   LOCALMQT
INTERFACE        | A |   LMQT_Interface
LINKID           | N |   0
[/MQT0]

[MQT1]
NAME             | S |   REMOTEMQT
INTERFACE        | A |   RMQT_Interface
LINKID           | N |   0
[/MQT1]
```

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

```
[DSP0]
...
MQTID            | N |   1
[/DSP0]
```

LDRV

§ In addition to the data transfer channels, two channels shall be reserved for messaging, one for messages to DSP and one for messages from DSP. The IDs of messaging channels shall be the two beyond the maximum ID in the system (MAX_CHANNELS).

§ To give priority to message transfer, the physical link layer shall always check messaging channels first for available messages. Following this, data channels shall be checked in round-robin fashion.

§ Messaging can be configured either to use the same physical link as used by the channels, or a different physical link for each processor.

- Local messaging within the GPP-side shall be directly taken care of by the local MQT, and does not need to use any link between the processors.
- The messaging implementation shall utilize services provided by the LDRV_CHNL subcomponent for channel management of the two messaging channels.
- Any messages with an ID_RMQT_CTRL destination ID shall indicate control messages intended for use by the MQT. These shall be used by the MQTs for internal communication, for example: locate, delete notification, finalization.
- The design of the MQA on the GPP-side is the same as that on the DSP-side.

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.

6.1 DSP side

The sequence diagrams indicate the control flow through the MQT and MQA on the DSP-side, and their interaction with the MSGQ & GIO components.

- The MQTs in all sequence diagrams are mentioned as either the ‘Local’ or ‘Remote’ MQTs, as applicable.
- The dashed arrow in all sequence diagrams indicates an indirect control transfer, which does not happen through a direct function call.

6.1.1 MSGQ_init ()

Note: MSGQ_init() calls the mqaOpen() and mqtOpen() for all allocators and transports, but only the BUF based allocator and remote transport call sequence is shown.

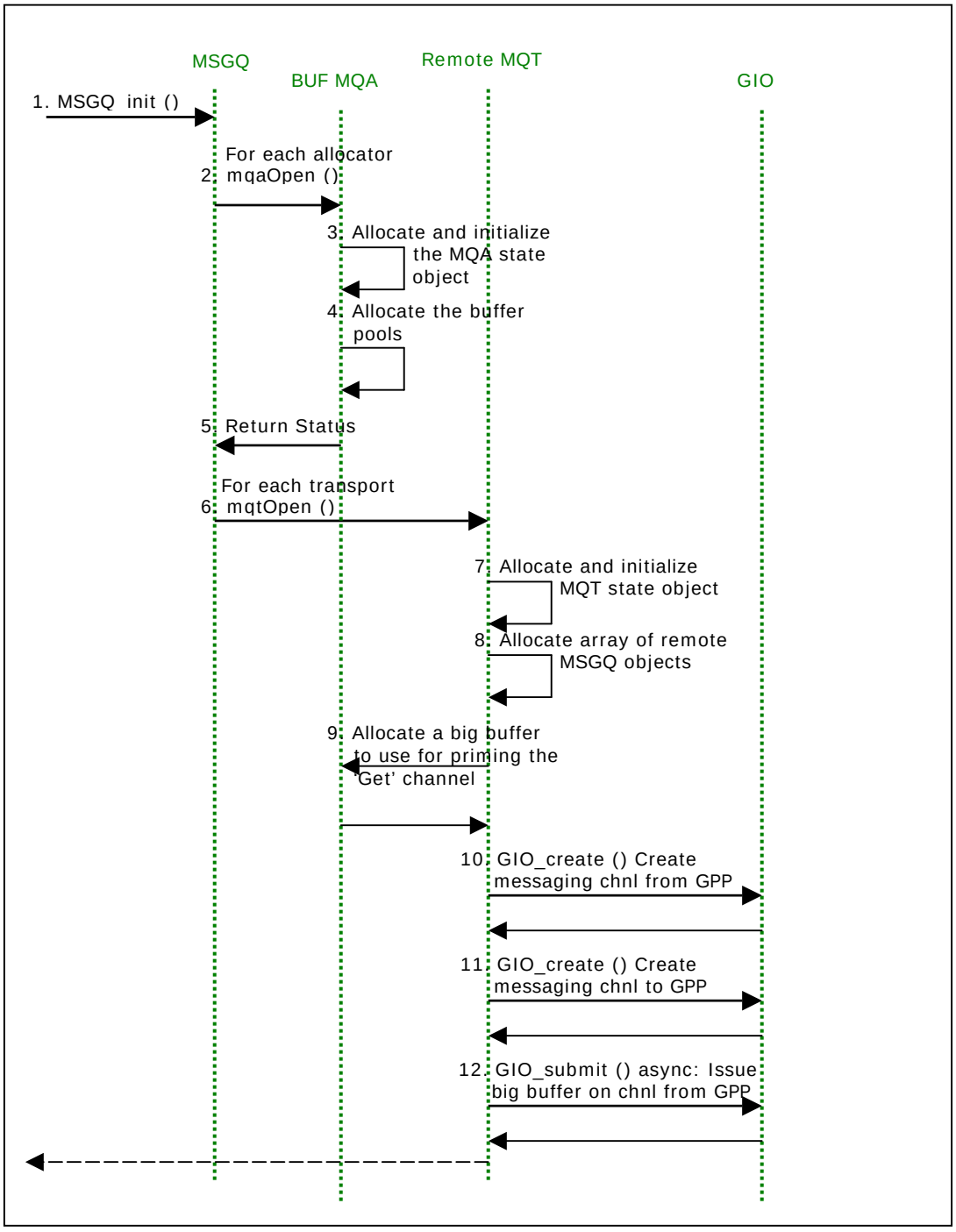

Figure 6. On the DSP: MSGQ_init () control flow

6.1.2 MSGQ_locate ()

Two cases are considered for MSGQ_locate (): Caller (Calls MSGQ_locate () for a remote MSGQ) & Receiver (Receives locate request from remote MQT on the GPP).

Caller (Calls MSGQ_locate () for a remote MSGQ):

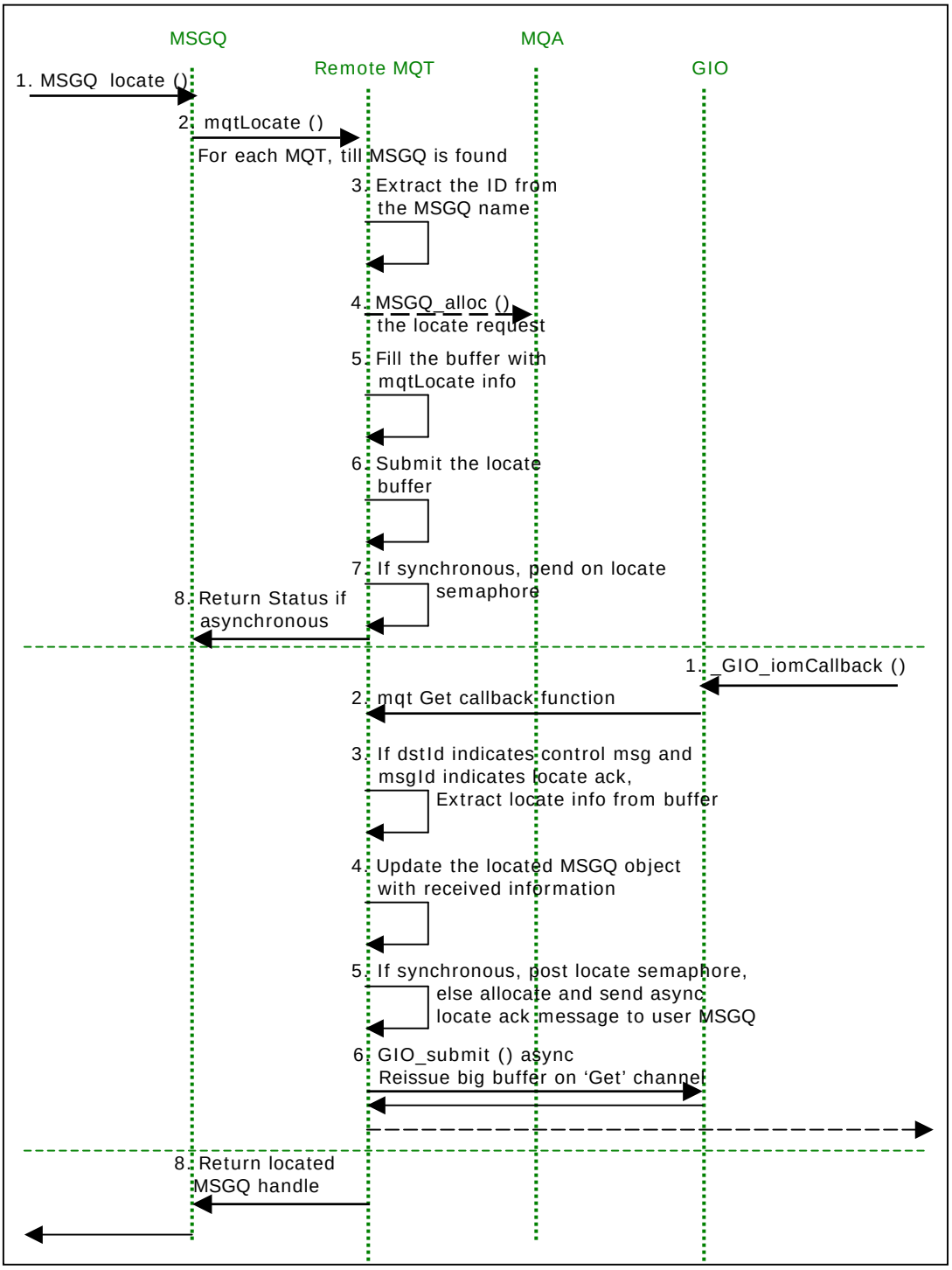

Figure 7. On the DSP: MSGQ_locate () control flow (Caller)

Receiver (Receives a locate request from remote MQT on the GPP):

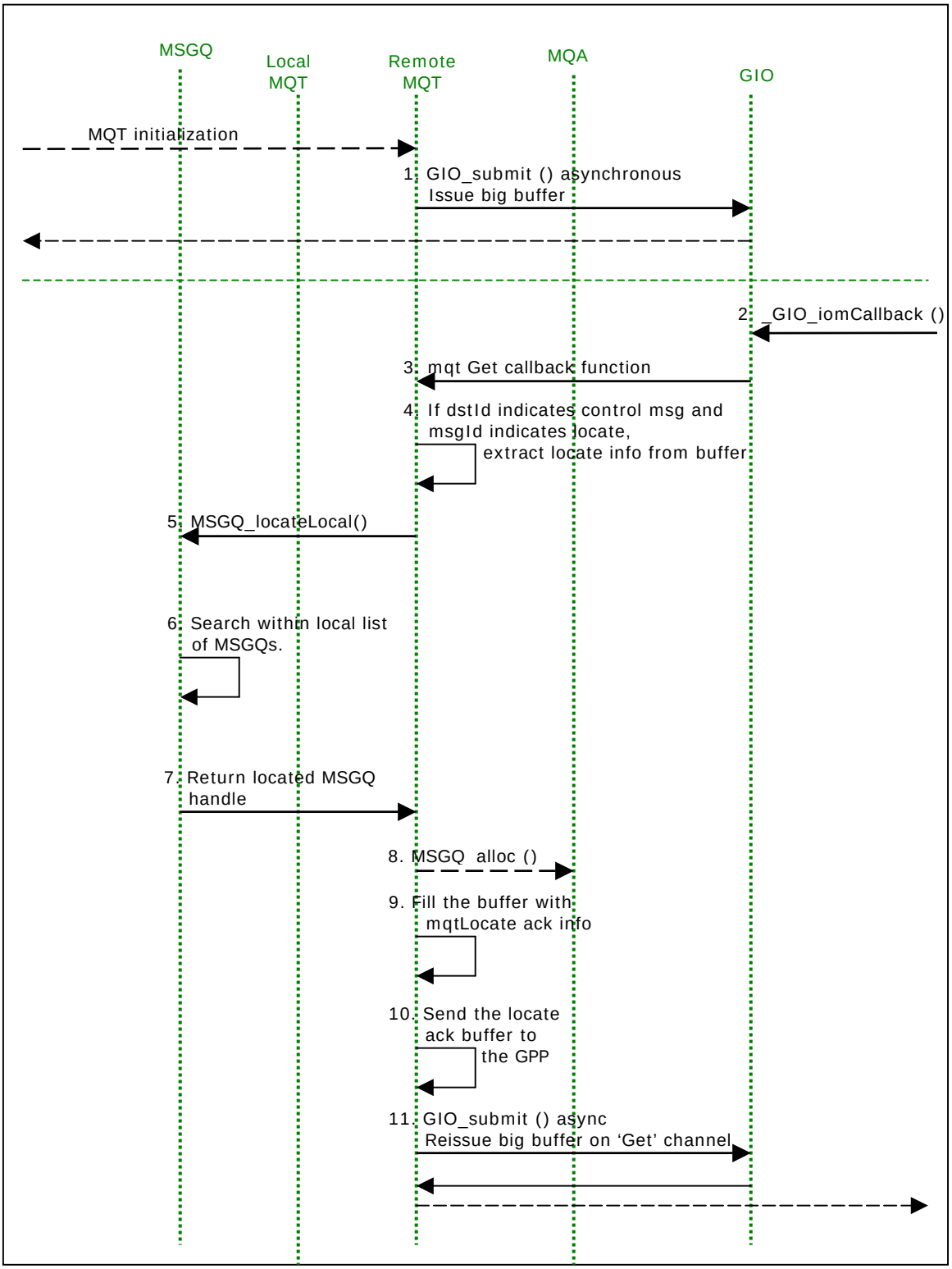

Figure 8. On the DSP: MSGQ_locate () control flow (Receiver)

q If the MSGQ cannot be located, the returned MSGQ handle shall be NULL.

- The mqtLocate information packet is identified by an ID_MQTDSPLINK_CTRL value in the dstId field of the message header to indicate a control message, in addition to a specific command in the msgId field of the message header.

- If mqtLocate is called on the same remote MSGQ more than once, it shall result in the MQT querying its counterpart on the remote processor each time. Existing information in the MSGQ object list of the remote MQT shall not be used. This shall enable detection of intermediate MSGQ deletion and always provide the latest run-time information on the MSGQs to the calling application.

- If multiple applications simultaneously make MSGQ_locate () calls for the same MSGQ, all these applications shall pend on the same 'locate' semaphore. The locate acknowledgement messages received from the GPP unblock the applications in the order that their MSGQ_locate () calls were made.

6.1.3 MSGQ_put ()

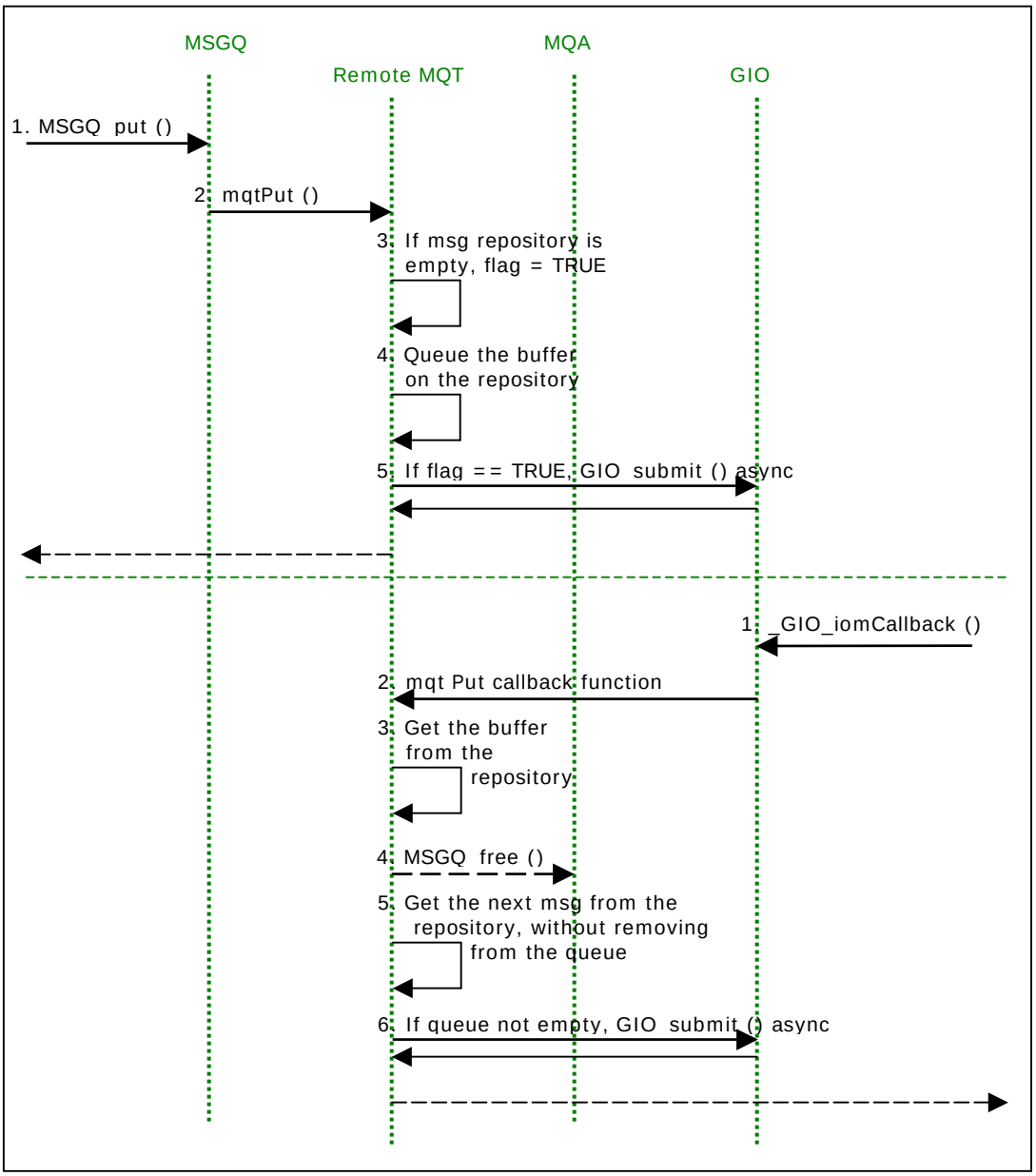

Figure 9. On the DSP: MSGQ_put () control flow

6.1.4 MSGQ_get ()

Two cases need to be considered for MSGQ_get (), when the GPP-side sends messages to a DSP-side MSGQ.

Message received from GPP before the MSGQ_get () call:

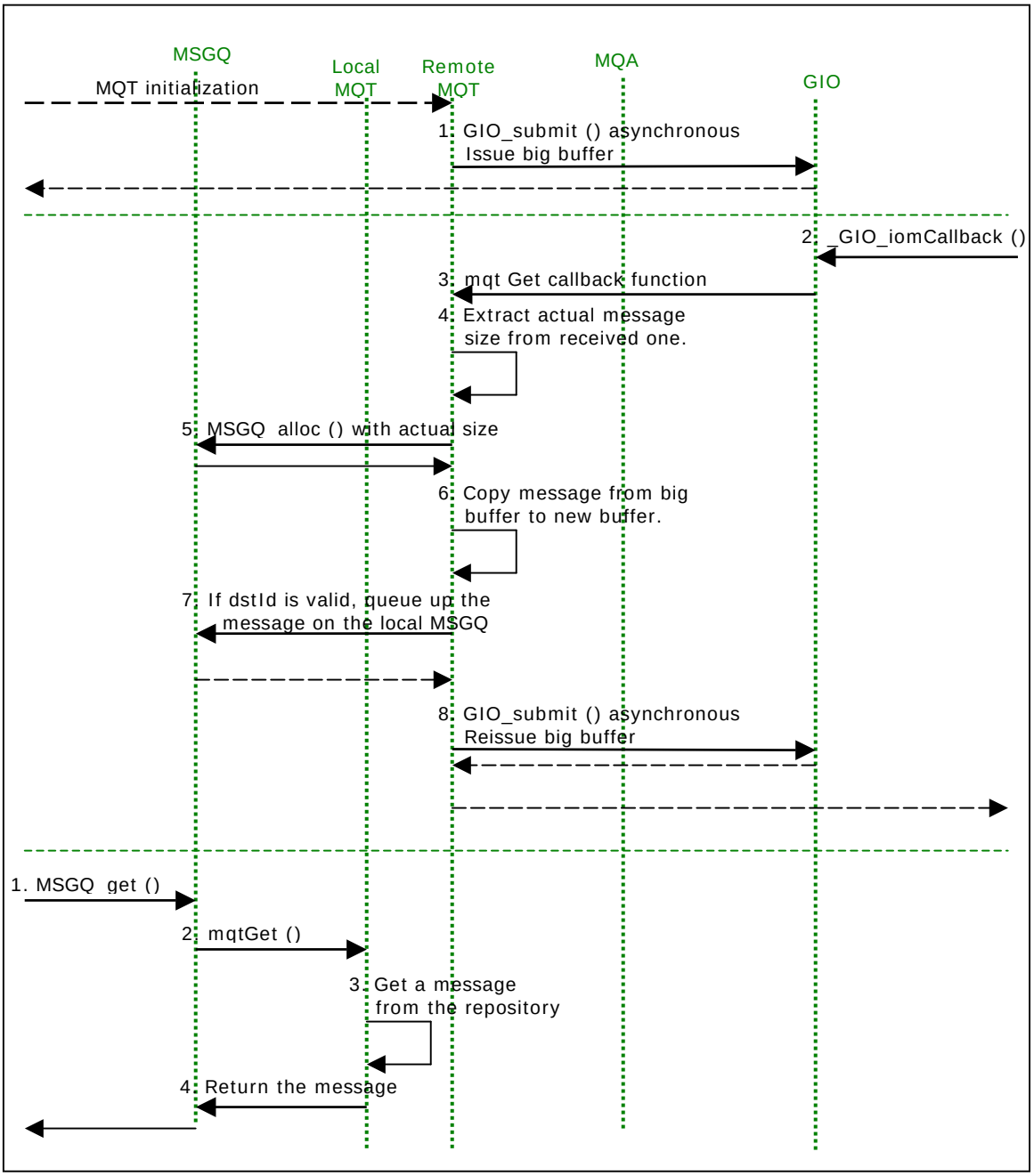

Figure 10. On the DSP: MSGQ_get () control flow (Message received from GPP before the MSGQ_get () call)

Message received from GPP after the MSGQ_get () call:

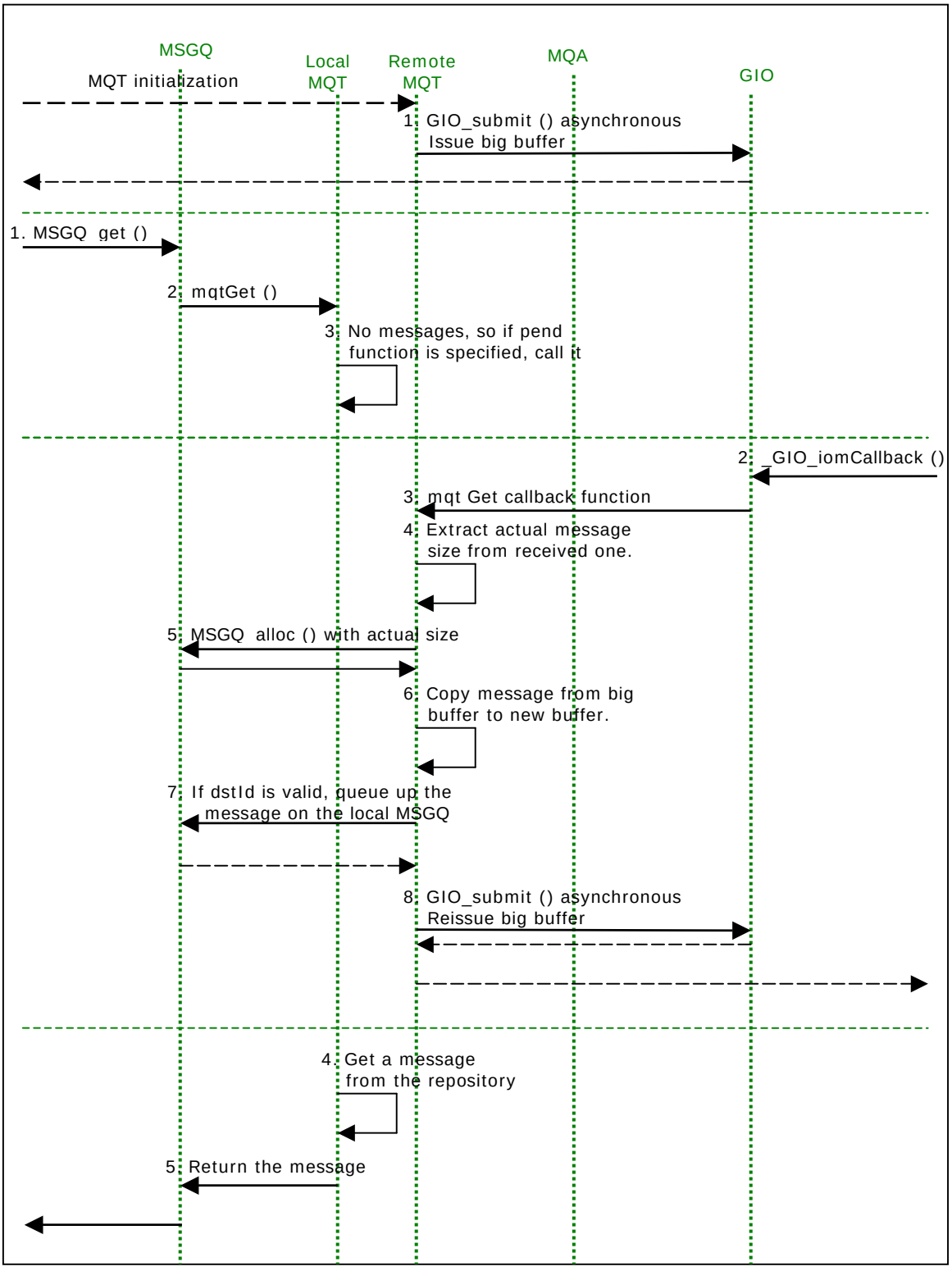

Figure 11. On the DSP: MSGQ_get () control flow (Message received from GPP after the MSGQ_get () call)

6.2 GPP side

6.2.1 Initialization

6.2.1.1 MSGQ

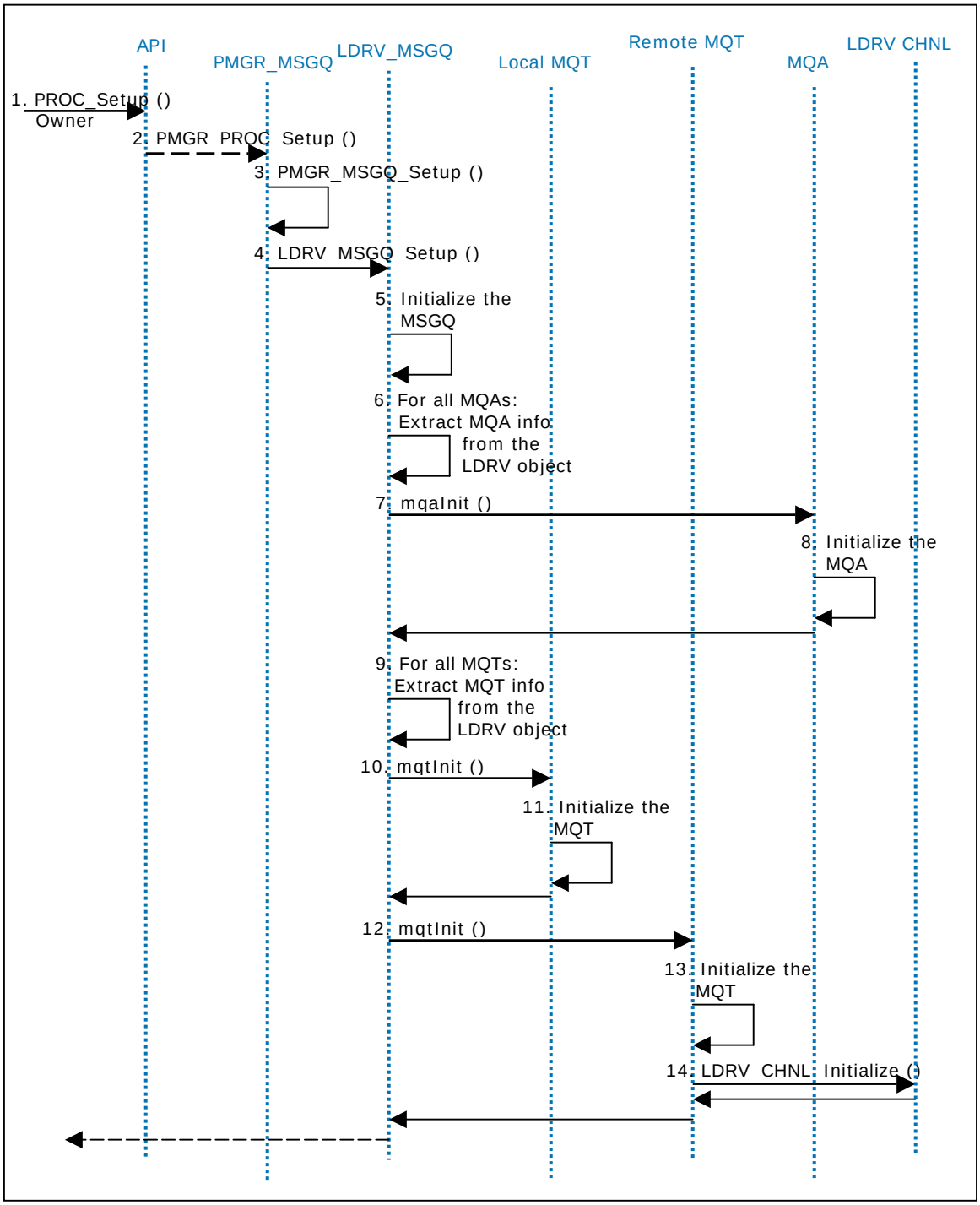

Figure 12. On the GPP: MSGQ initialization

6.2.1.2 Buffer MQA

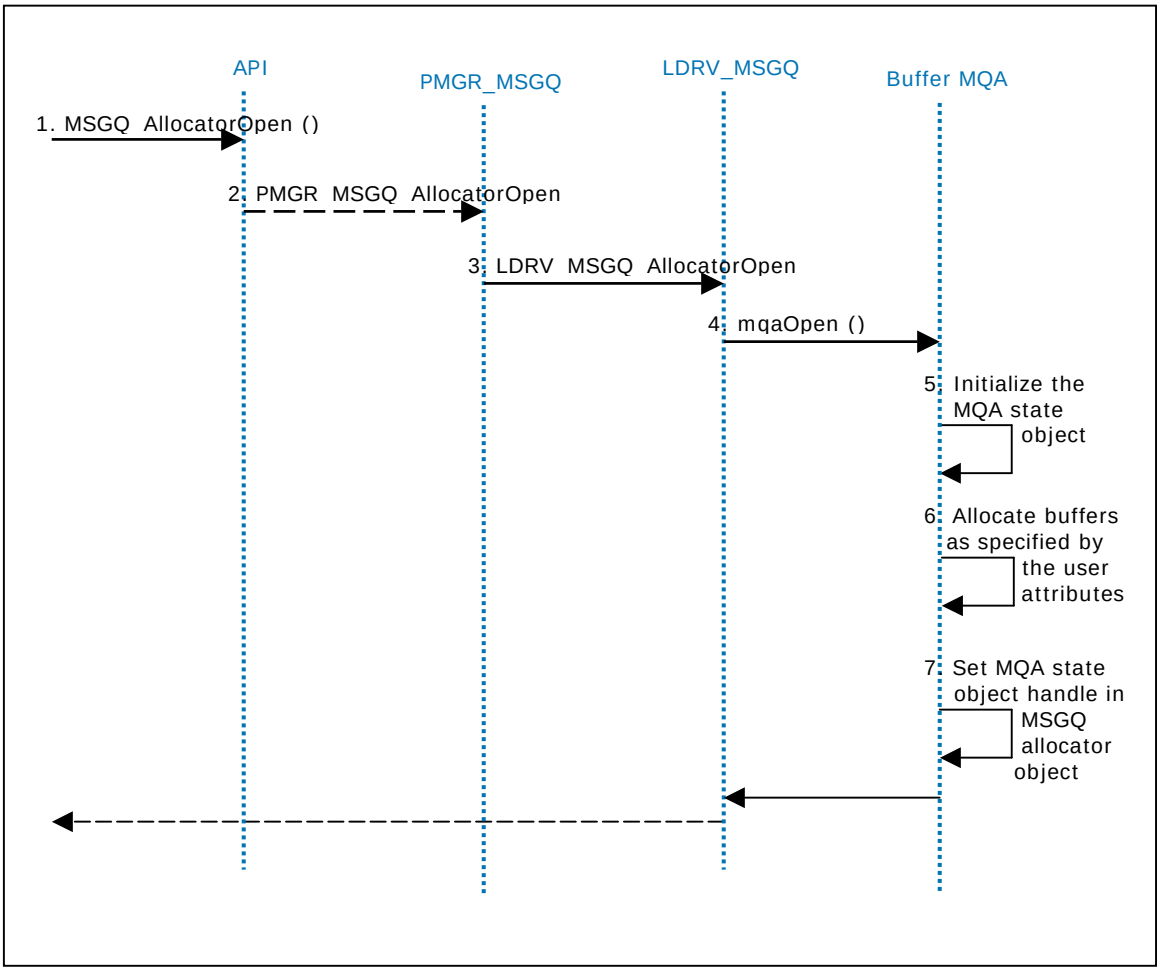

Figure 13. On the GPP: MSGQ_AllocatorOpen () control flow

6.2.1.3 *Local MQT*

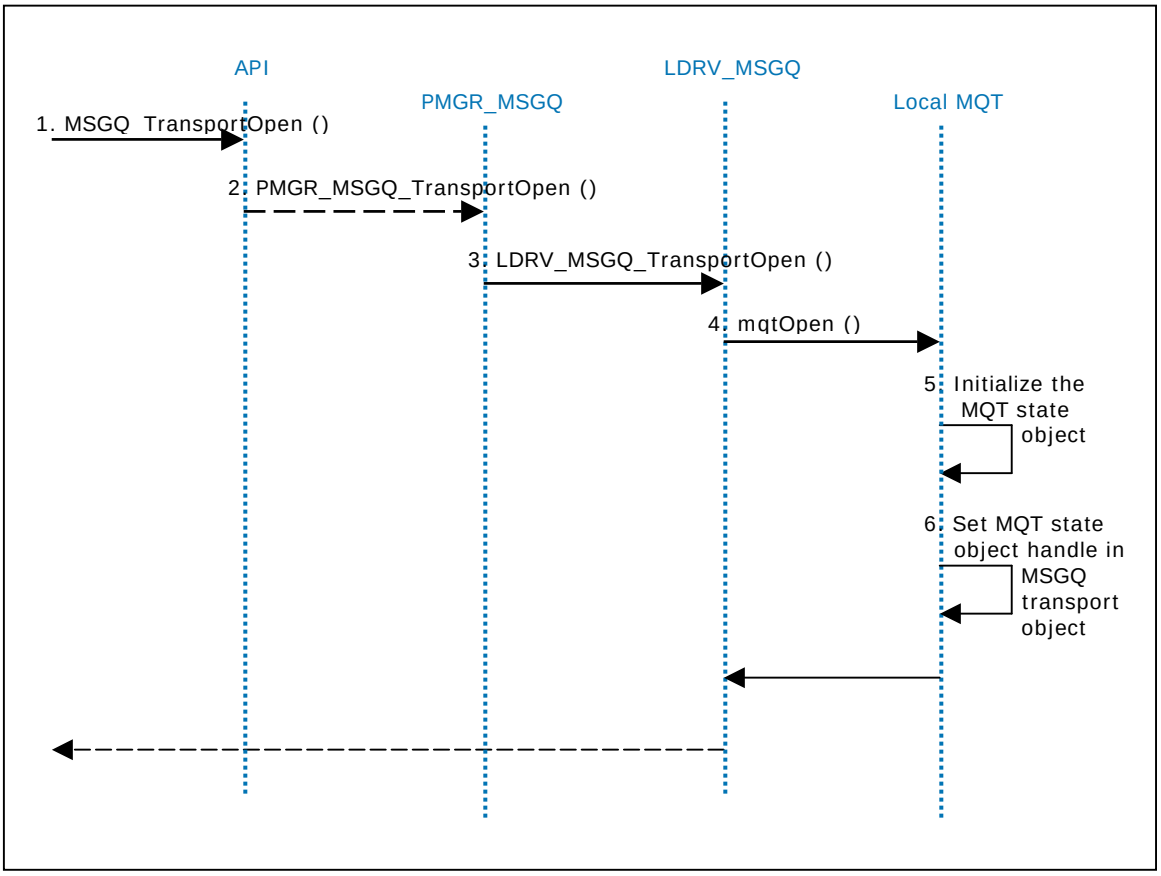

Figure 14. On the GPP: MSGQ_TransportOpen () control flow - Local MQT

6.2.1.4 Remote MQT

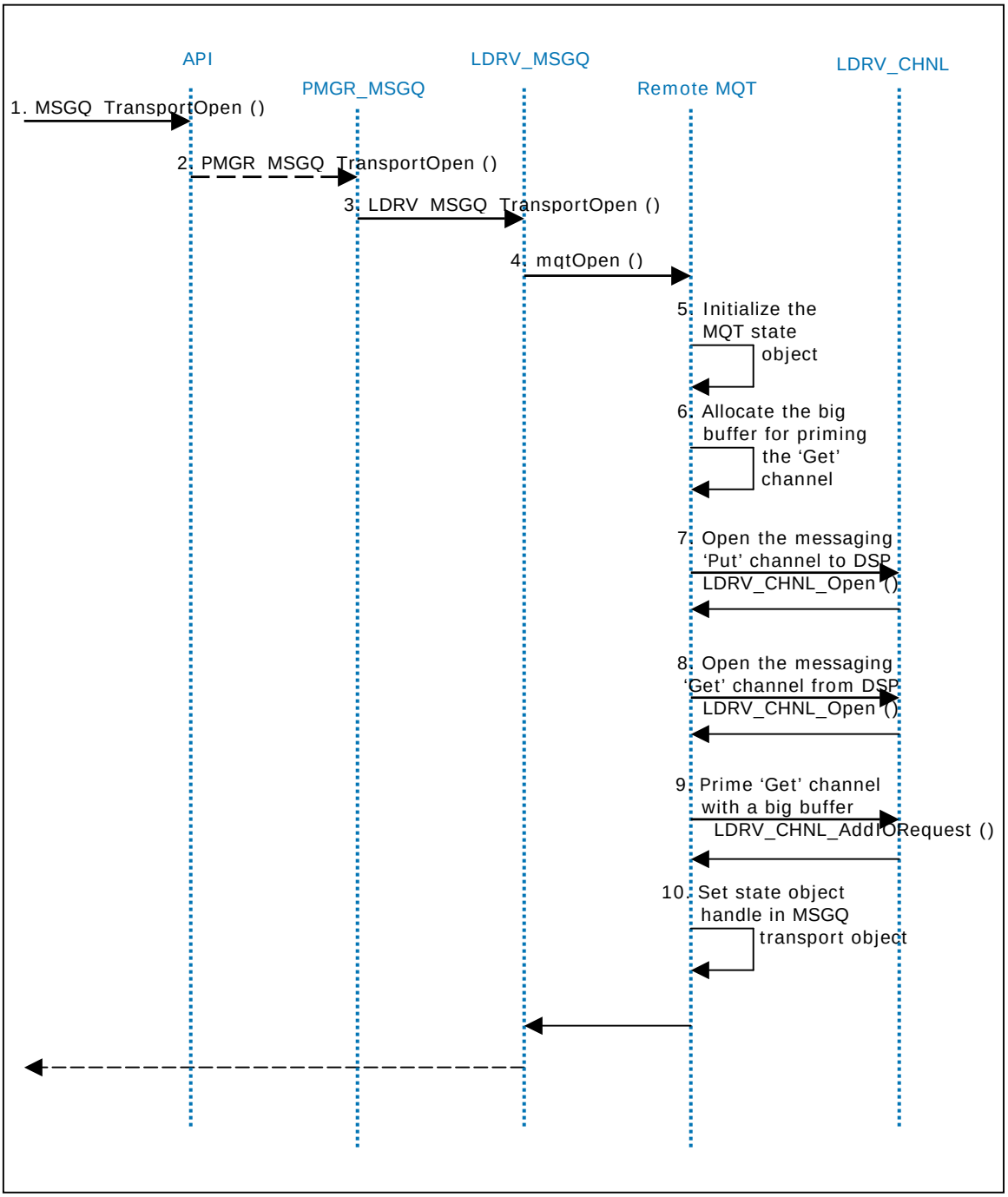

Figure 15. On the GPP: MSGQ_TransportOpen () control flow - Remote MQT

6.2.2 Finalization

6.2.2.1 MSGQ

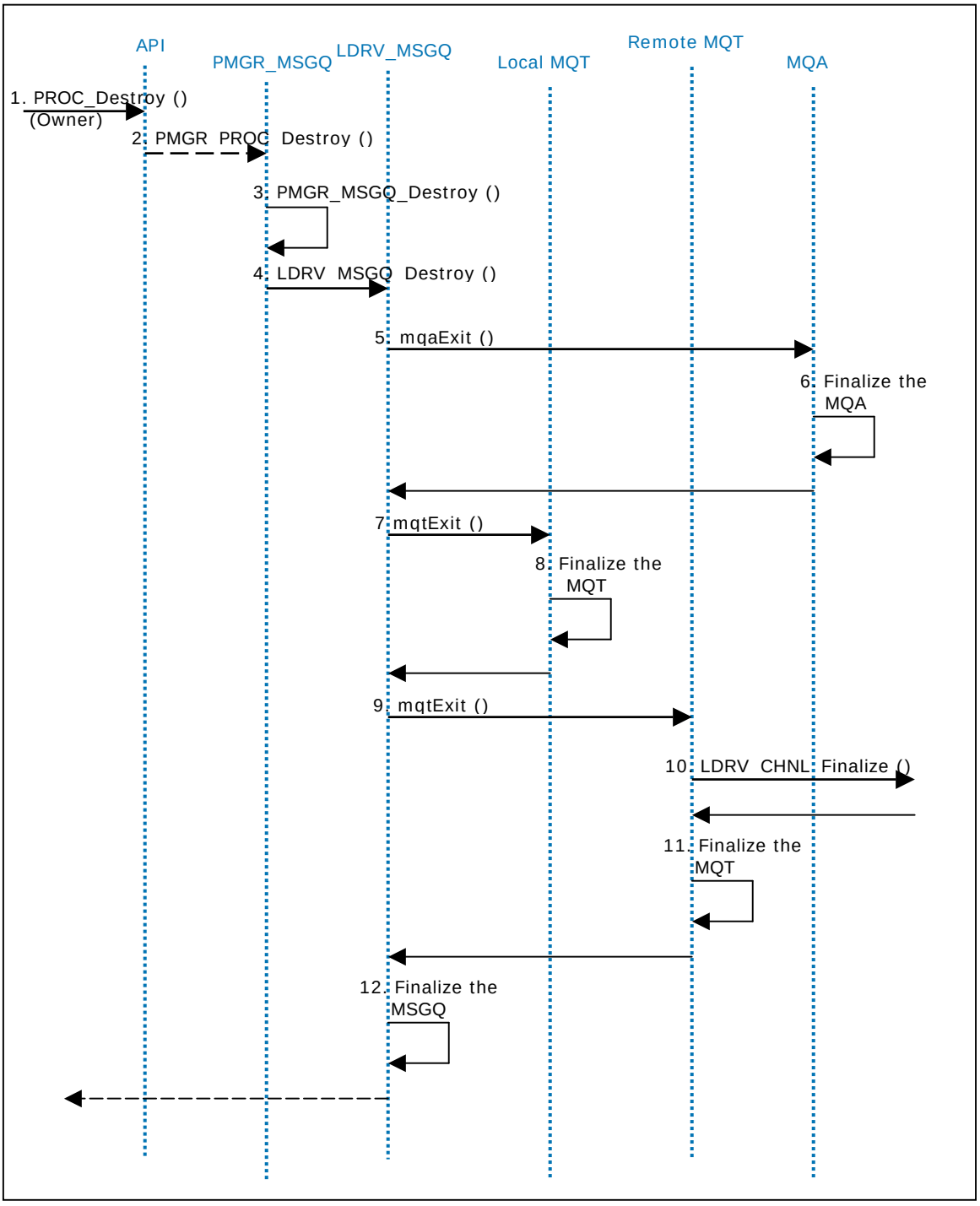

Figure 16. On the GPP: MSGQ finalization

6.2.2.2 Buffer MQA

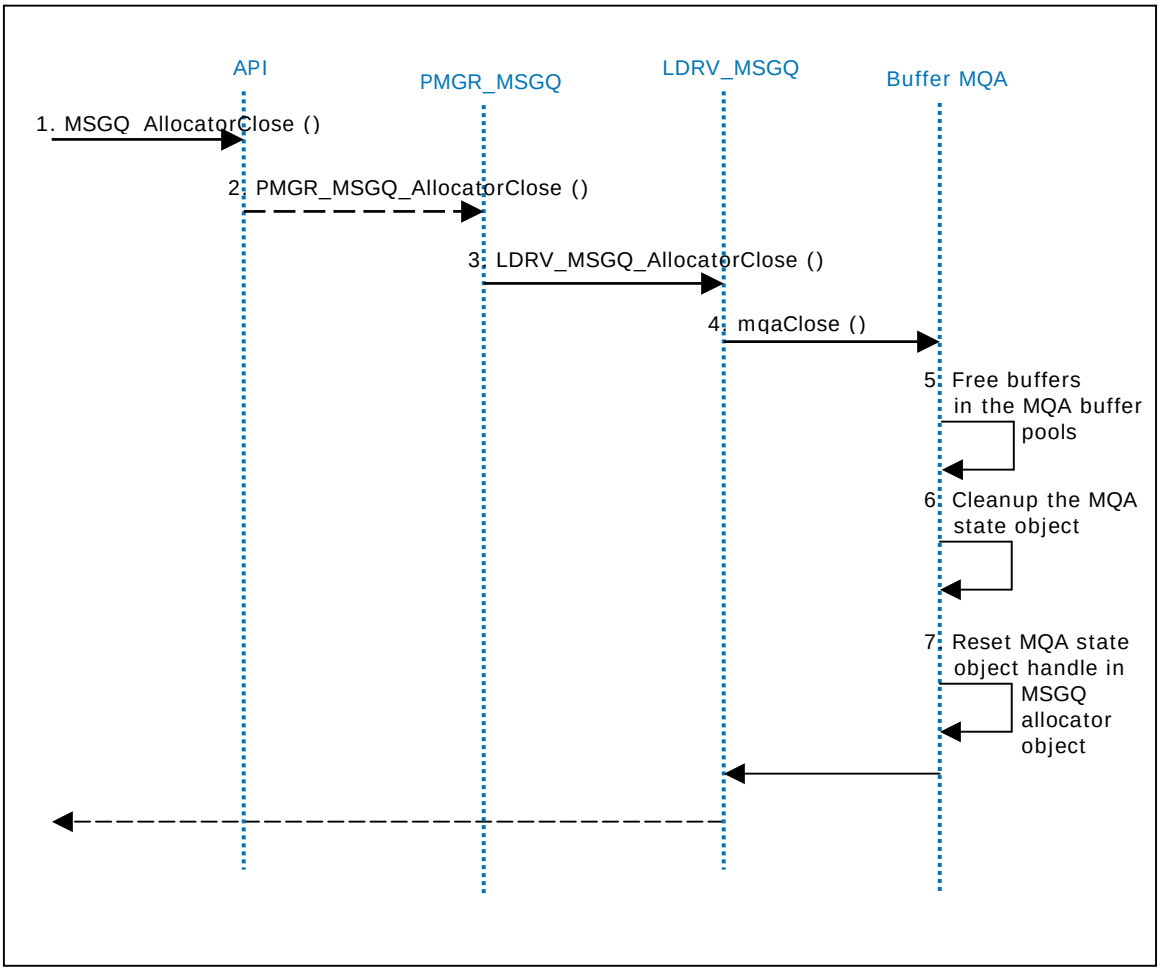

Figure 17. On the GPP: MSGQ_AllocatorClose () control flow

6.2.2.3 *Local MQT*

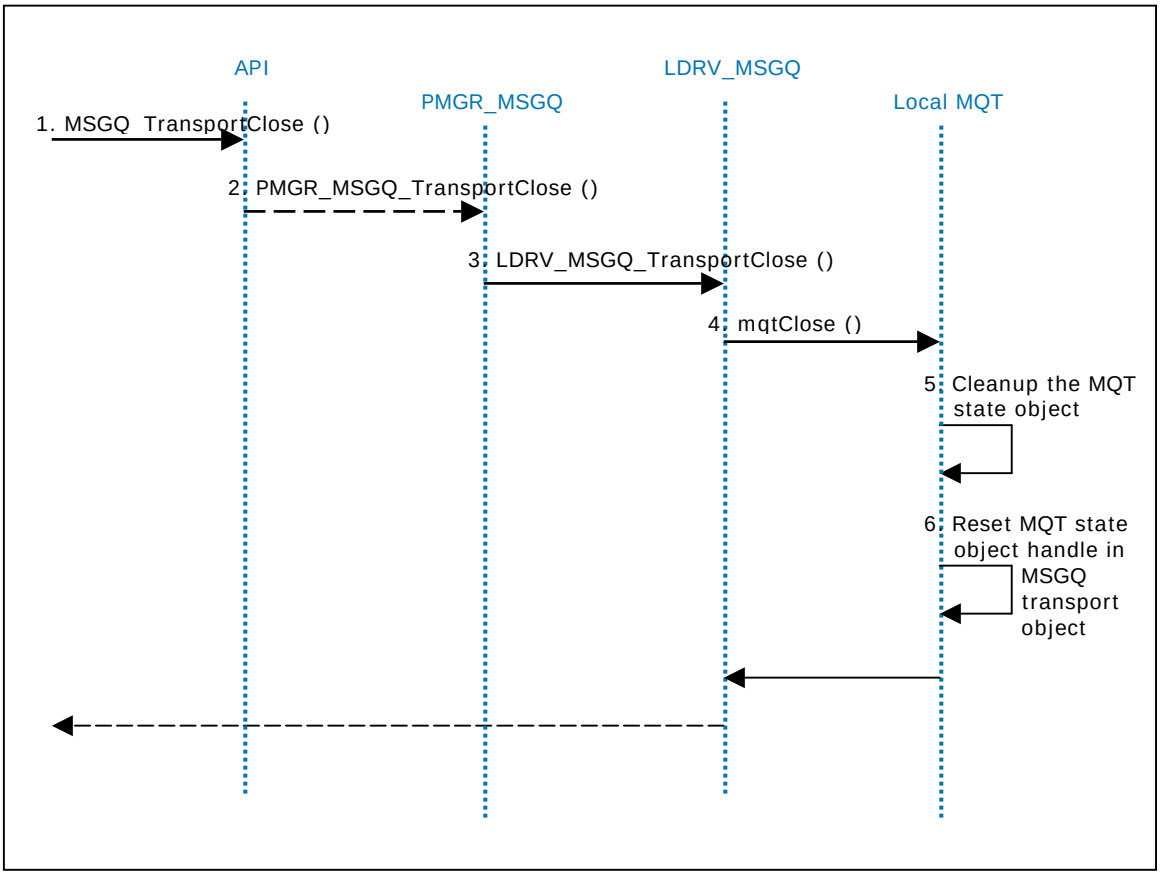

Figure 18. On the GPP: MSGQ_TransportClose () control flow - Local MQT

6.2.2.4 *Remote MQT*

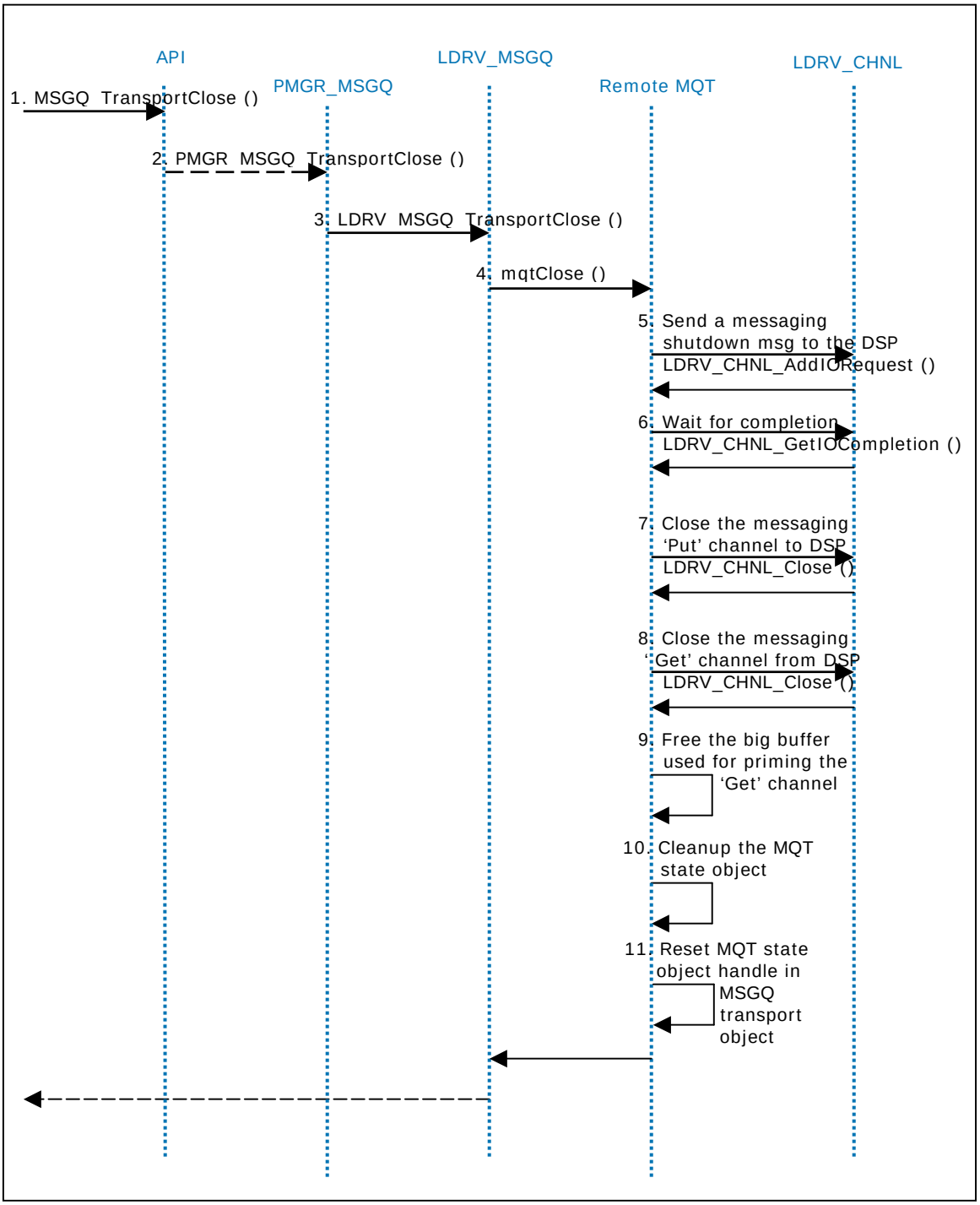

Figure 19. On the GPP: MSGQ_TransportClose () control flow - Remote MQT

6.2.3 MSGQ_Create ()

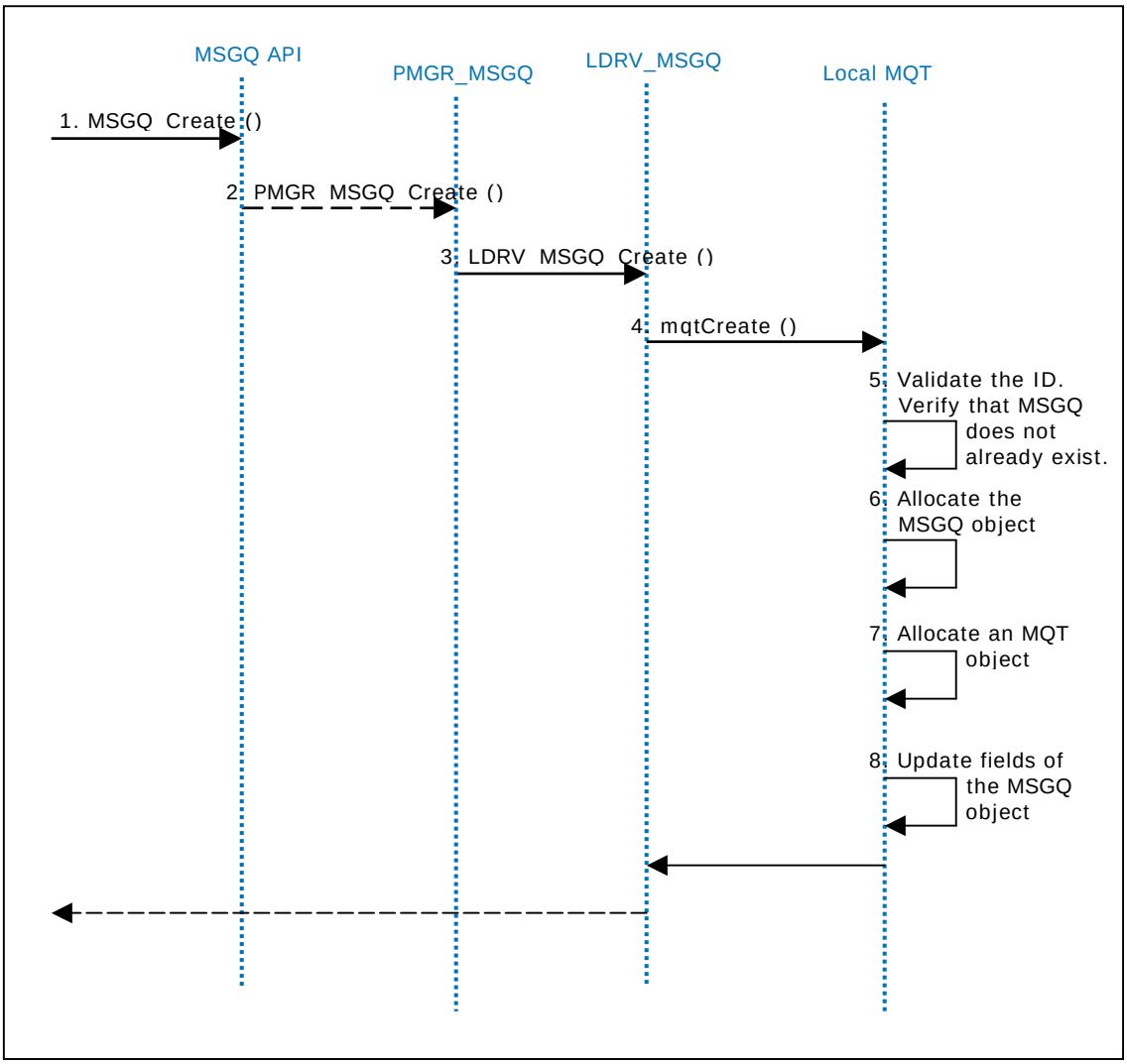

Figure 20. On the GPP: MSGQ_Create () control flow

- MSGQ_Create () is only called on local queues, and hence only the local MQT is involved in the control flow for this API.

6.2.4 MSGQ_Delete ()

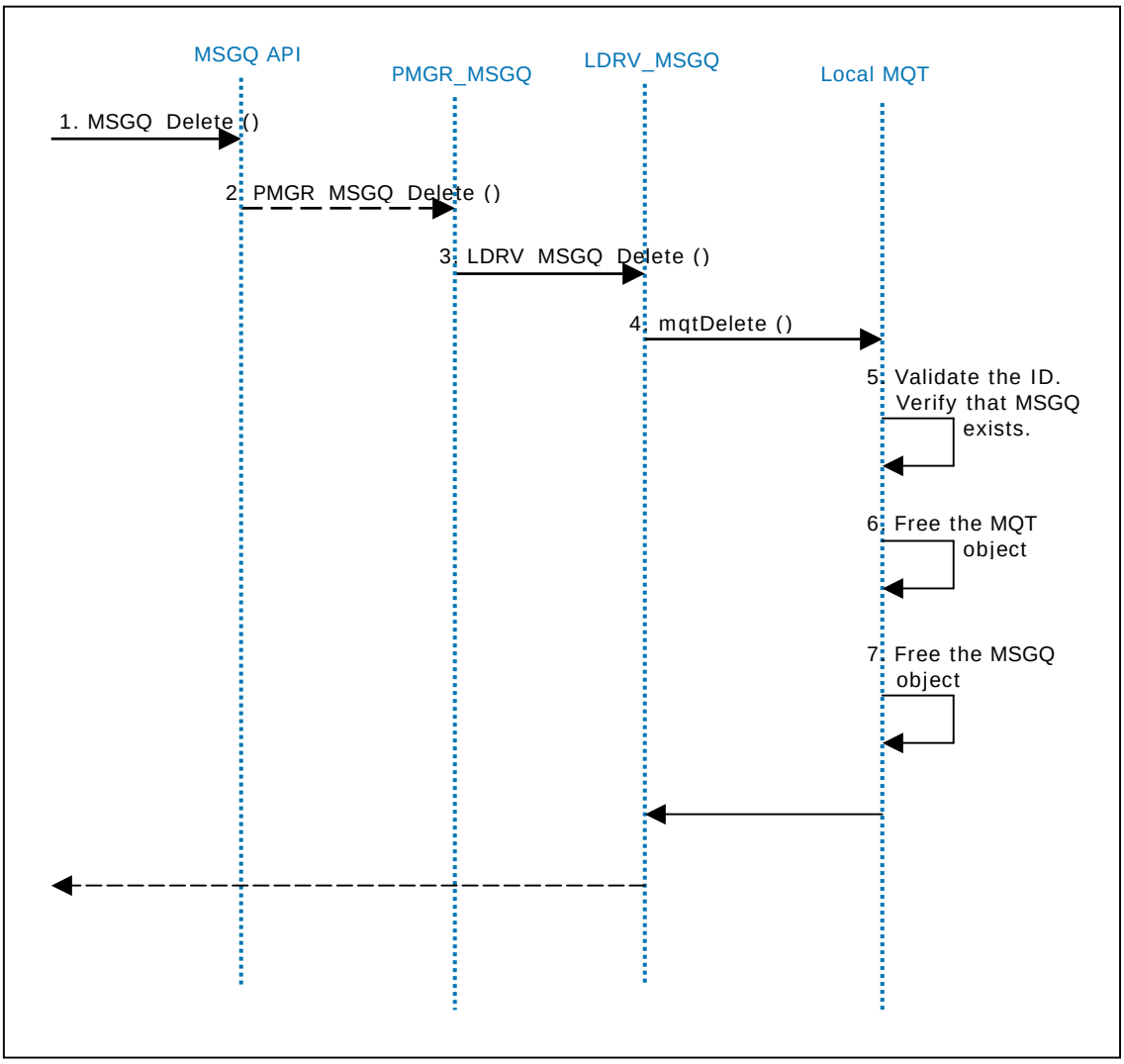

Figure 21. On the GPP: MSGQ_Delete () control flow

- MSGQ_Delete () is only called on local queues, and hence only the local MQT is involved in the control flow for this API.

6.2.5 MSGQ_Locate ()

6.2.5.1 Local MSGQ

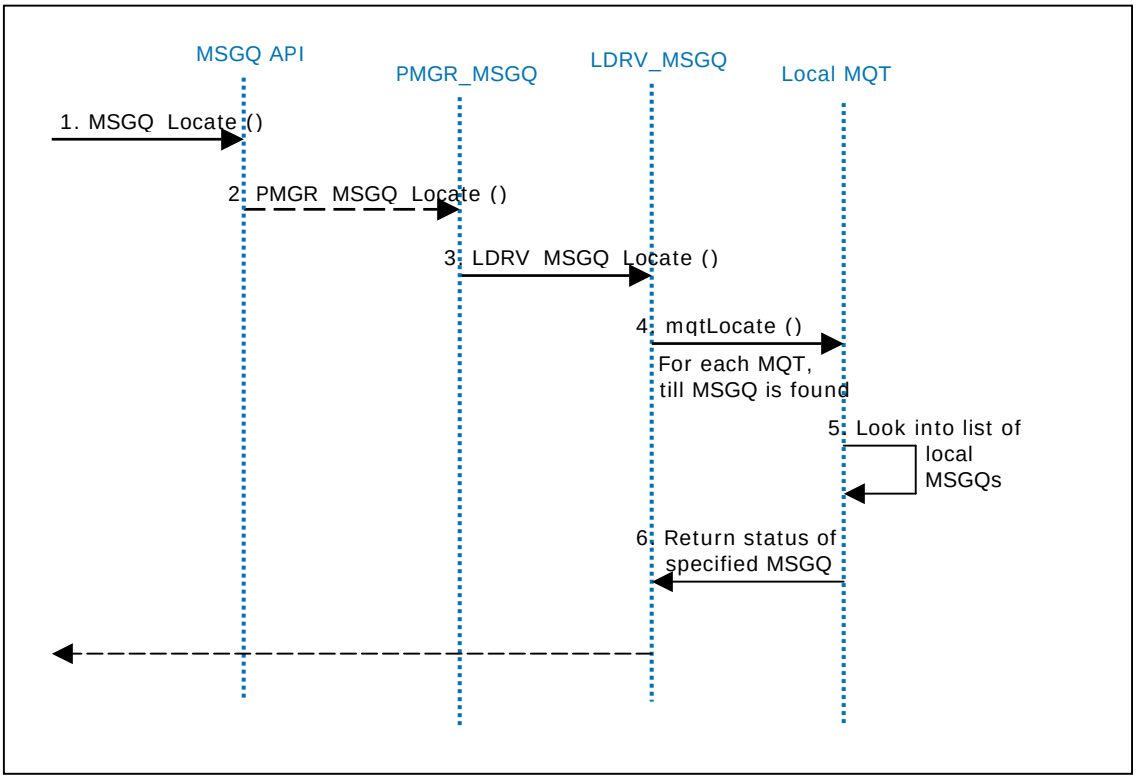

Figure 22. On the GPP: MSGQ_Locate () control flow – Local MSGQ

6.2.5.2 *Remote MSGQ*

Two cases need to be considered for `MSGQ_Locate ()` for a remote `MSGQ`: Caller (Calls `MSGQ_Locate ()` for remote MSGQ) & Receiver (Receives locate request from remote MQT on DSP).

Caller (Calls MSGQ_locate () for a remote MSGQ):

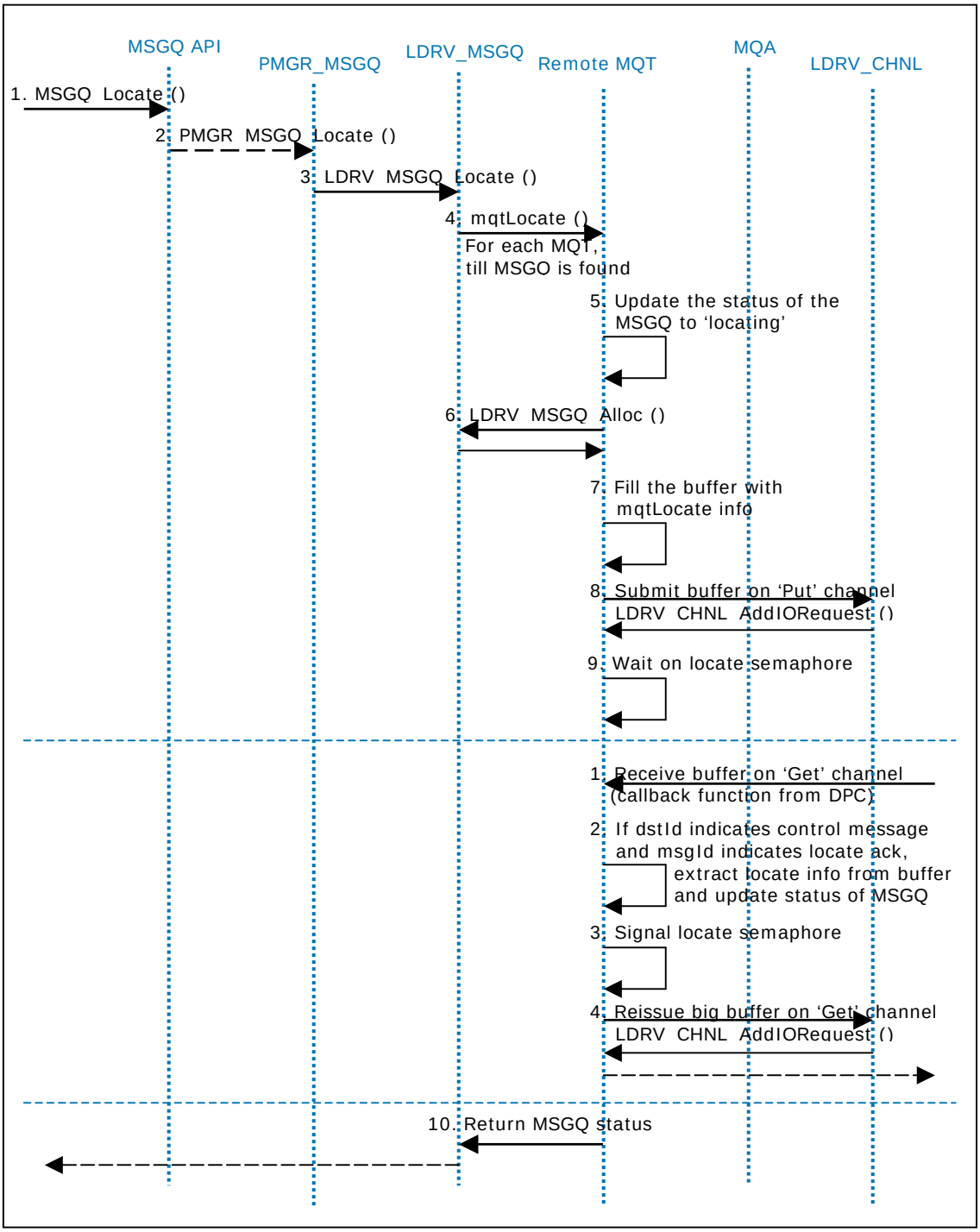

Figure 23. On the GPP: MSGQ_Locate () control flow – Remote MSGQ (Caller)

Receiver (Receives a locate request from remote MQT on the GPP):

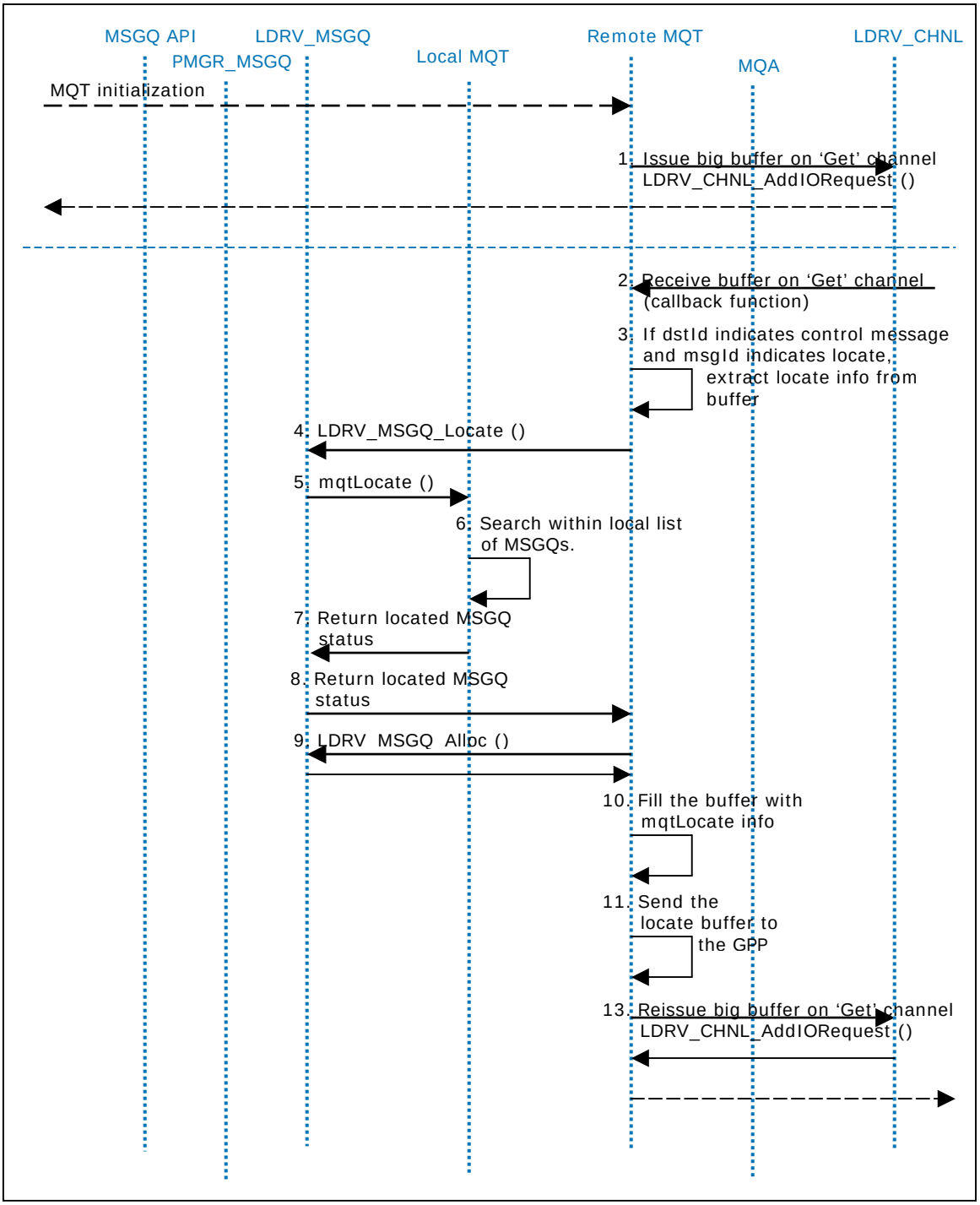

Figure 24. On the GPP: MSGQ_Locate () control flow – Remote MSGQ (Receiver)

- If the MSGQ cannot be located, DSP_ENOTFOUND shall be returned.
- The mqtLocate information packet is identified by an ID_RMQT_CTRL value in the dstId field of the message header to indicate a control message.

- If mqtLocate () is called on the same remote MSGQ more than once, it shall result in the MQT querying its counterpart on the remote processor each time. Existing information in the MSGQ object list of the remote MQT shall not be used. This shall enable detection of intermediate MSGQ deletion and always provide the latest run-time information on the MSGQs to the calling application.

- If multiple applications simultaneously make MSGQ_Locate () calls for the same MSGQ, all these applications shall wait on the same 'locate' semaphore. The locate acknowledgement messages received from the GPP unblock the applications in the order that their MSGQ_Locate () calls were made.

6.2.6 MSGQ_Put ()

6.2.6.1 Local MSGQ

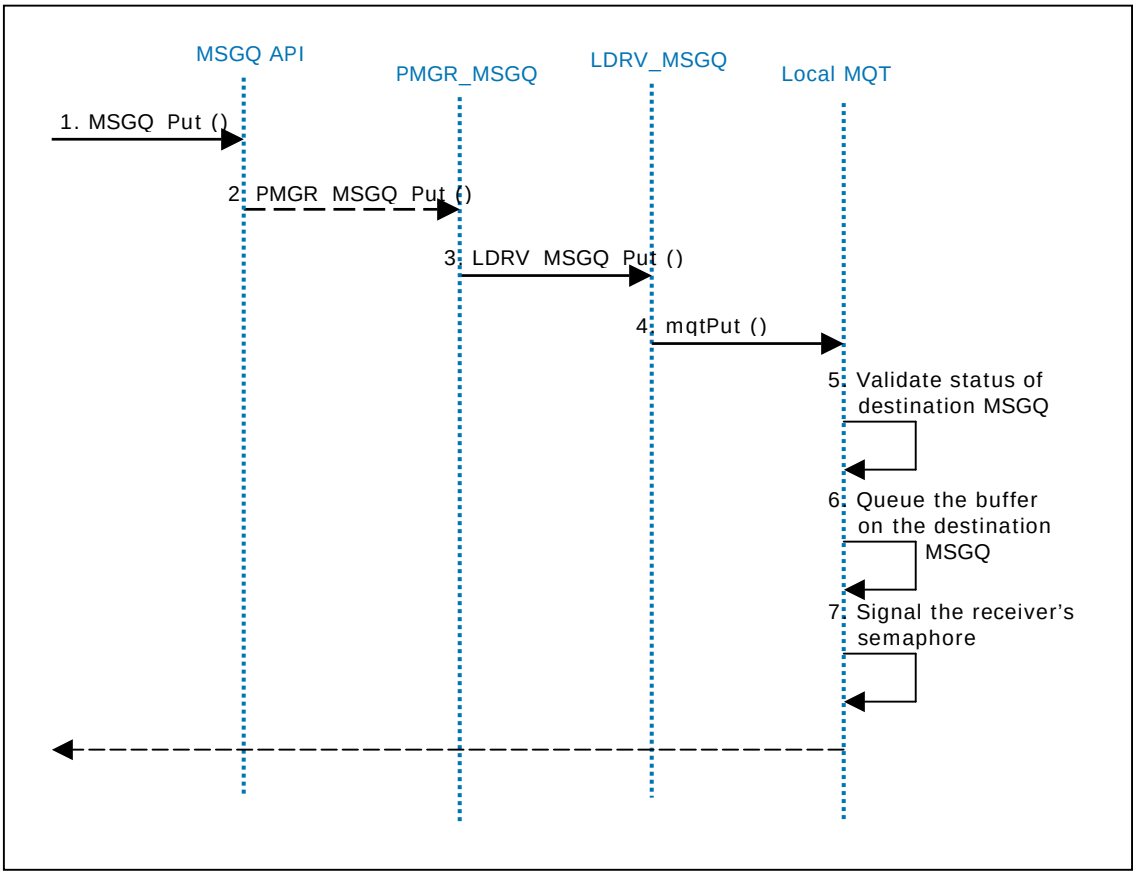

Figure 25. On the GPP: MSGQ_Put () control flow – Local MSGQ

6.2.6.2 Remote MSGQ

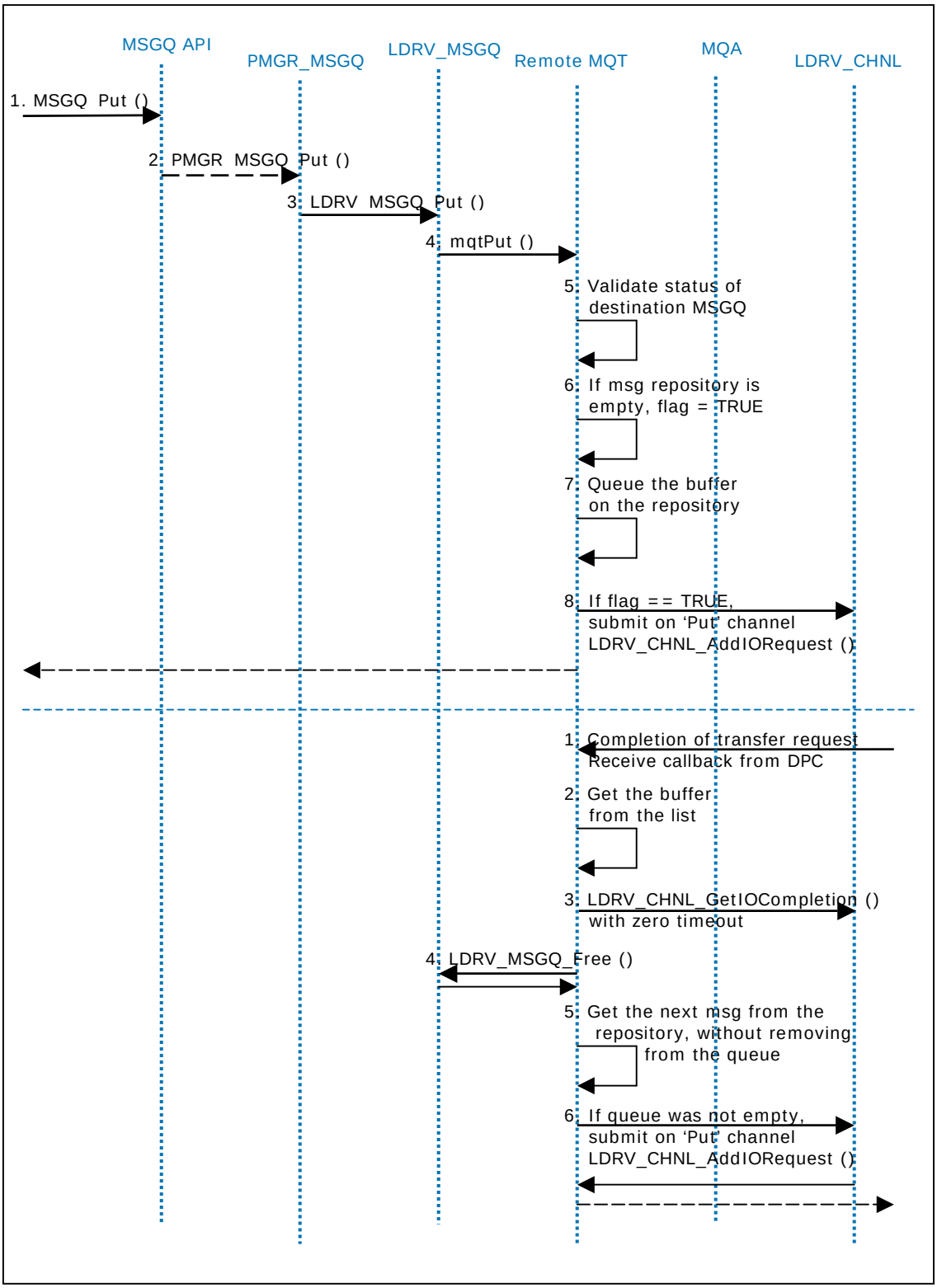

Figure 26. On the GPP: MSGQ_Put () control flow – Remote MSGQ

6.2.7 MSGQ_Get ()

6.2.7.1 Local MSGQ

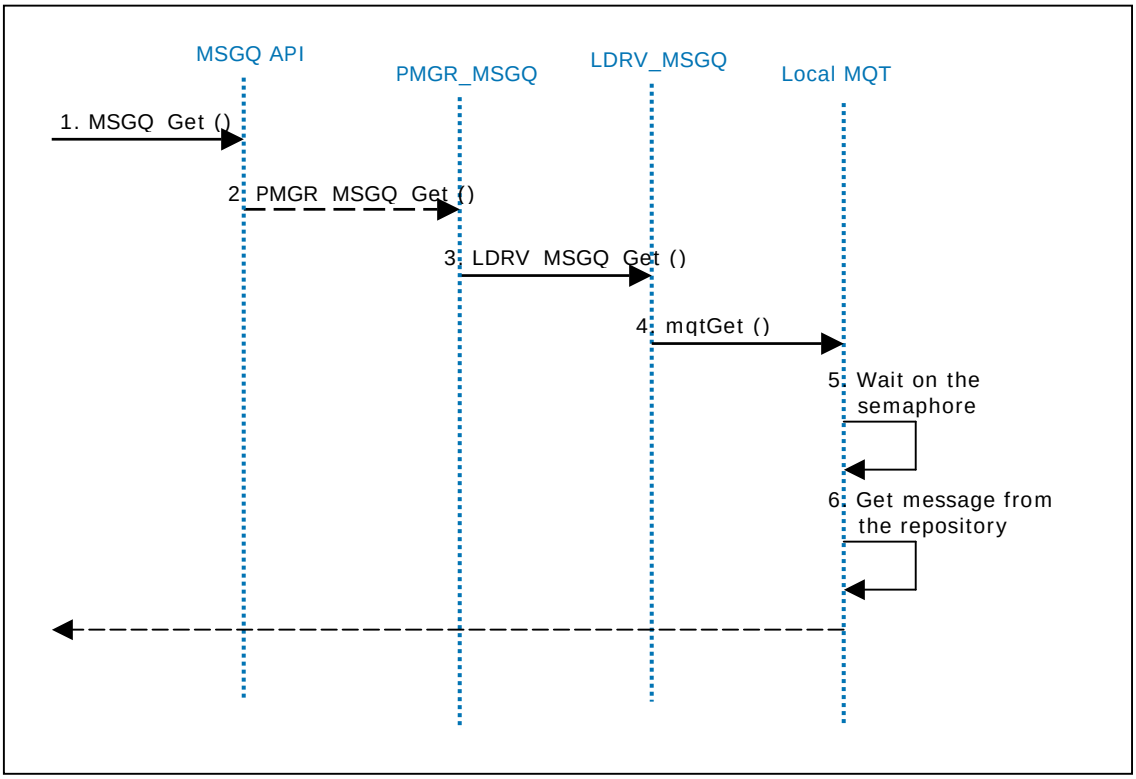

Figure 27. On the GPP: MSGQ_Get () control flow – Local MSGQ

6.2.7.2 Remote MSGQ

Two cases need to be considered for MSGQ_Get (), when the DSP-side sends messages to a GPP-side MSGQ.

Message received from DSP before the MSGQ_Get () call:

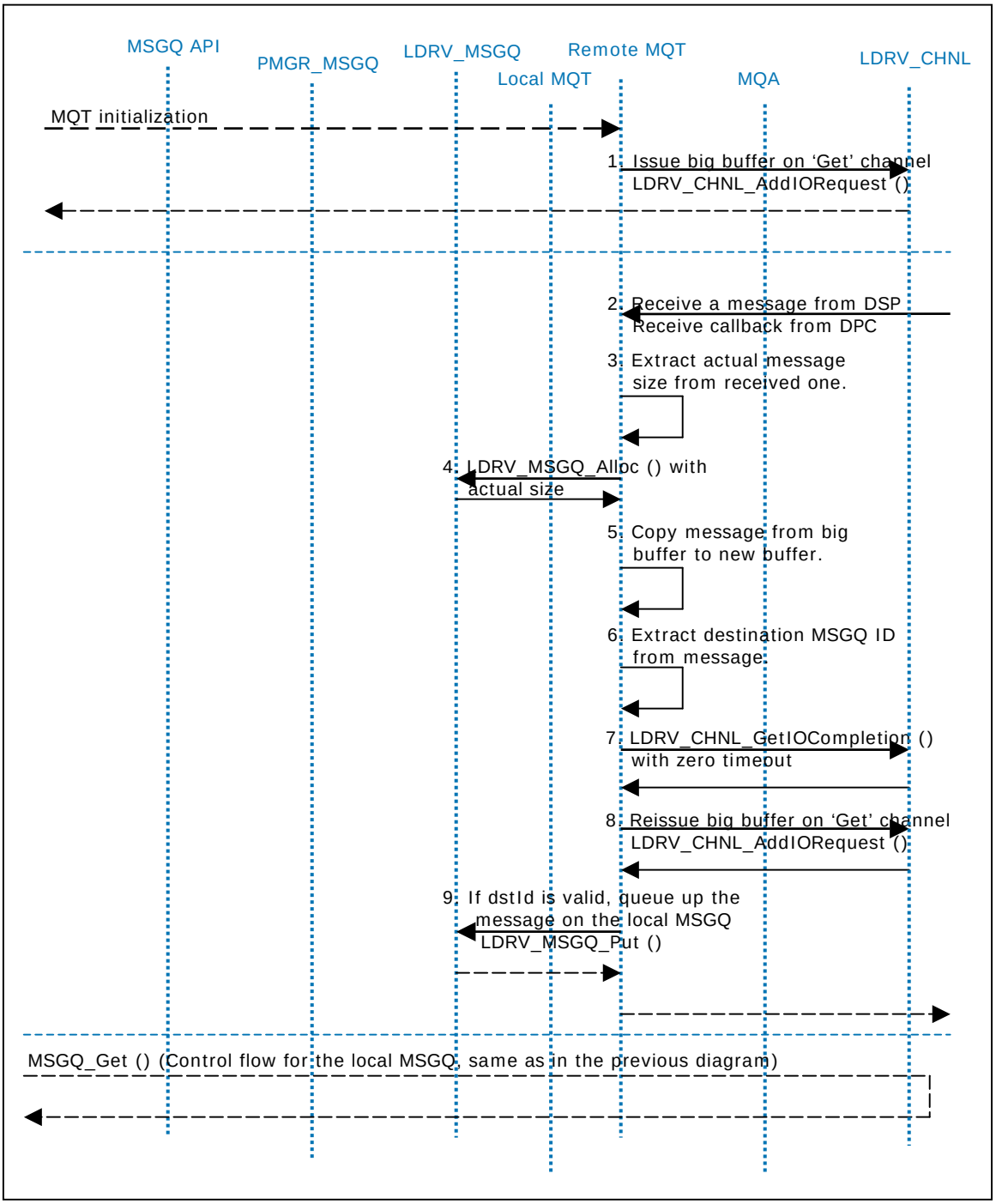

Figure 28. On the GPP: MSGQ_Get () control flow (Message received from DSP before the MSGQ_Get () call)

Message received from DSP after the MSGQ_Get () call:

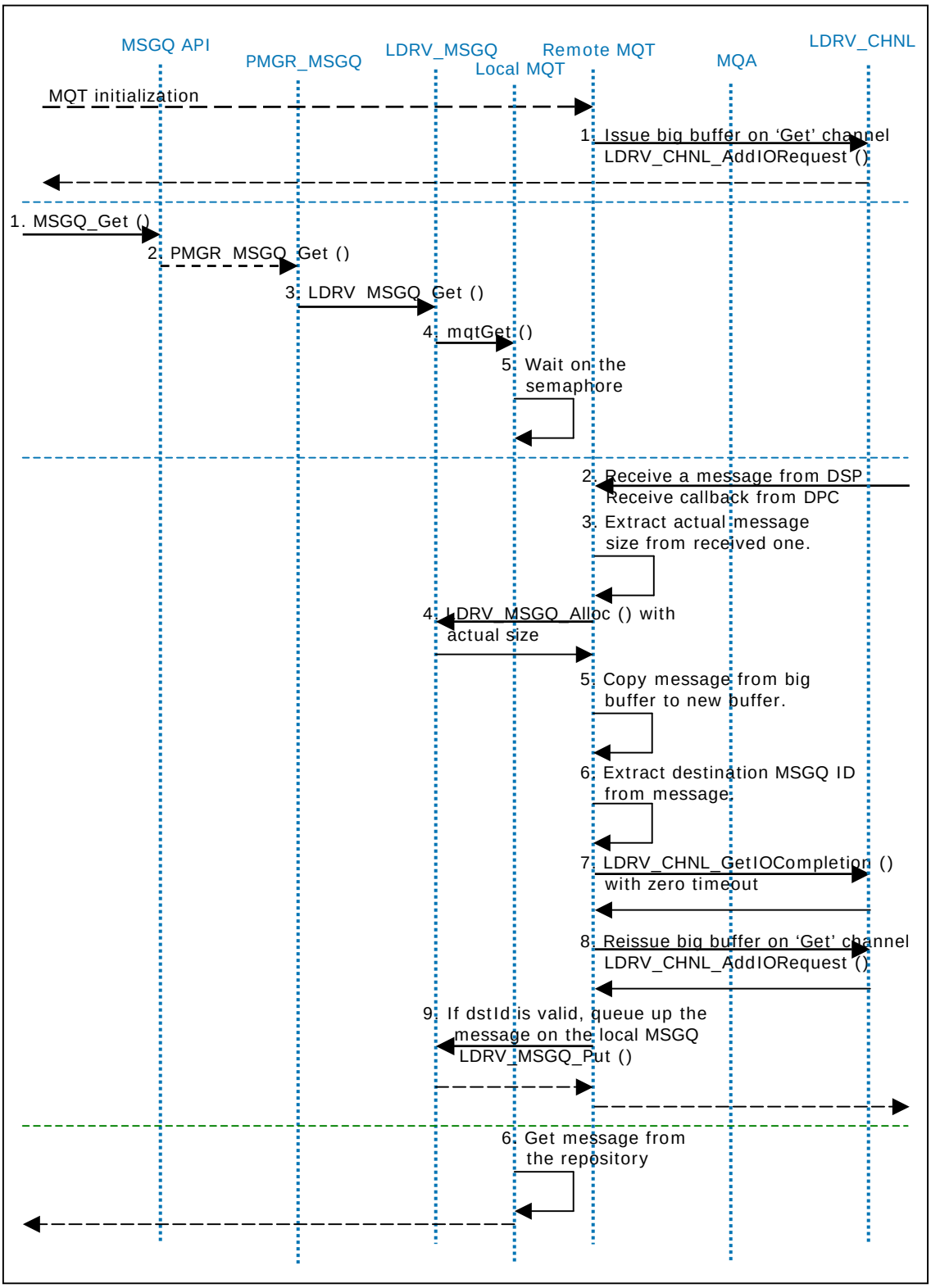

Figure 29. On the GPP: MSGQ_Get () control flow (Message received from GPP after the MSGQ_Get () call)

7 DSP side

7.1 MQA

7.1.1 Constants & Enumerations

7.1.2 Typedefs & Data Structures

7.1.2.1 MQABUF_Params

This structure defines the attributes required for initialization of the buffer MQA.

Definition

```
typedef struct MQABUF_Params_tag {
    Uint16    numBufPools ;
    Uint16 *  msgSize ;
    Uint16 *  numMsg ;
    Uint16    segId ;
} MQABUF_Params ;
```

Fields

`numBufPools`	Number of buffer pools to be configured in the MQA.
`msgSize`	Array of sizes of the messages in the buffer pools. The sizes are in MADUs. This array is of size `numBufPools`.
`numMsg`	Array of number of messages in all buffer pools. This array is of size `numBufPools`.
`segId`	Segment ID for allocation of the buffers.

Comments

These parameters are provided to the MQA once during its initialization, which takes place during the call to `MSGQ_init ()`.

The default parameters are provided to the user through the following structure:

```
extern MQABUF_Params MQABUF_PARAMS;
```

The definition is within the source file:

```
Uns MQABUF_MsgSize [4] = {16, 32, 64, sizeof (MSGQ_AsyncLocateMsg)} ;
Uns MQABUF_NumMsg  [4] = {10, 10, 10, 10} ;
MQABUF_Params MQABUF_PARAMS = {4,
                               (Uns *) MQABUF_MsgSize,
                               (Uns *) MQABUF_NumMsg,
                               0
                               } ;
```

Constraints

None.

See Also

`MQABUF_open ()`

7.1.2.2 MQABUF_State

This structure defines the allocator object, which represents the complete allocator.

Definition

```
typedef struct MQABUF_State_tag {
    Uint16        numBufPools ;
    MQABUF_Obj *  bufPools ;
} MQABUF_State ;
```

Fields

`numBufPools`	Number of buffer pools configured in the MQA.
`bufPools`	Array of buffer pools for various message sizes. The array is dynamically allocated of size equal to the one specified by the user.

Comments

An instance of this object is created and initialized during `MQABUF_open ()`, and the handle is returned to the caller. It contains all information required for maintaining the state of the MQA.

Constraints

None.

See Also

`MQABUF_Obj`
`MQABUF_open ()`

7.1.2.3 MQABUF_Obj

This structure defines the buffer object for the allocator, representing a single buffer pool.

Definition

```
typedef struct MQABUF_Obj_tag {
    Uint16        msgSize ;
    BUF_Handle    msgList ;
} MQABUF_Obj ;
```

Fields

`msgSize`	Size (in MADUs) of the messages in the buffer pool.
`msgList`	List of messages in the buffer pool.

Comments

The allocator contains multiple buffer pools for various message sizes. This structure defines the buffer object for a particular message size.

Constraints

None.

See Also

`MQABUF_State`
`MQABUF_alloc ()`
`MQABUF_free ()`

7.1.3 API Definition

The MQA APIs are exposed to MSGQ through a function table:

```
MSGQ_AllocatorFxns MQABUF_FXNS = {
    &MQABUF_open,
    &MQABUF_close,
    &MQABUF_alloc,
    &MQABUF_free
} ;
```

7.1.3.1 MQABUF_init

This function performs global initialization of the buffer MQA.

Syntax

```
Void MQABUF_init () ;
```

Arguments

None.

Return Value

None.

Comments

The `mqaInit ()` and `mqaExit ()` functions are directly exposed by the MQA to the application. The `MQABUF_init ()` function is called by the application, to perform any global initialization required for the MQA.

Constraints

None.

See Also

`MQABUF_exit ()`

7.1.3.2 MQABUF_exit

This function performs global finalization of the buffer MQA.

Syntax

```
Void MQABUF_exit () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called by the application to perform any global finalization required for the MQA.

Constraints

None.

See Also

`MQABUF_init ()`

7.1.3.3 MQABUF_open

This function opens the buffer MQA and configures it according to the user attributes.

Syntax

```
MSGQ_Status MQABUF_open (MSGQ_AllocatorHandle  mqaHandle) ;
```

Arguments

IN	MSGQ_AllocatorHandle	mqaHandle

Handle to the MSGQ allocator object.

Return Value

MSGQ_SUCCESS	This component has been successfully opened.
MSGQ_EMEMORY	Failure during memory operation.
MSGQ_EFAILURE	Failure in opening the component.

Comments

This API is called during MSGQ_init(). It creates an instance of the MQABUF_State object, initializes it, and sets the handle in the MSGQ allocator object. This handle to the MQA state object is passed to the allocator during its functions for allocating and freeing the messages, as part of the MSGQ allocator object.

A default allocator is implemented within DSPLINK, for usage during messaging. However, users may configure their own MQAs, with the following constraints, if they are used with MSGQs used for transferring messages through DSPLINK:

- The MQAs must not have dynamic allocation of memory.
- The MQA allocation and free functions must be callable from HWI and SWI context.

Constraints

This function cannot be called from SWI or HWI context.

See Also

```
MQABUF_Params
MQABUF_State
MQABUF_close ()
```

7.1.3.4 MQABUF_close

This function closes the MQA, and cleans up its state object.

Syntax

```
MSGQ_Status MQABUF_close (MSGQ_AllocatorHandle mqaHandle) ;
```

Arguments

`IN` `MSGQ_AllocatorHandle` `mqaHandle`

Handle to the MSGQ allocator object.

Return Value

`MSGQ_SUCCESS`	This component has been successfully closed.
`MSGQ_EMEMORY`	Failure during memory operation.
`MSGQ_EFAILURE`	Failure in closing the component.

Comments

This API is called during `MSGQ_exit()`. It frees up the instance of the `MQABUF_State` object, in addition to any other required actions for finalizing the MQA.

After successful completion of this function, no further calls can be made to the MQA for allocation or freeing of messages.

Constraints

This function cannot be called from SWI or HWI context.

See Also

`MQABUF_State`
`MQABUF_open ()`

7.1.3.5 MQABUF_alloc

This function allocates a message buffer of the specified size.

Syntax

```
MSGQ_Msg    MQABUF_alloc (MSGQ_AllocatorHandle mqaHandle,
                          Uint16 *             size) ;
```

Arguments

IN	MSGQ_AllocatorHandle	mqaHandle

Handle to the MSGQ allocator object.

IN OUT	Uint16 *	size

Pointer to the size (in MADUs) of the message to be allocated. On return, it stores the actual allocated size of the message, which, for the buffer MQA, is the same as the requested size on success, or zero on failure.

Return Value

Valid message handle	The message was successfully allocated.
NULL	Failure in message allocation.

Comments

This API is called during MSGQ_alloc (). This function attempts to allocate a buffer of the size required by the caller. If a buffer of the requested size is not found, or the buffer list corresponding to the specified size is empty, an error is returned. In addition, the size allocated is returned as zero, and the returned buffer pointer is NULL.

This function allocates the specified message by removing it from the appropriate buffer pool. The buffer pool is identified based on the size of the message to be allocated.

Constraints

None.

See Also

MQABUF_State
MQABUF_free ()

7.1.3.6 MQABUF_free

This function frees a message buffer of the specified size.

Syntax

```
Void MQABUF_free (MSGQ_AllocatorHandle mqaHandle,
                  MSGQ_Msg             msg,
                  Uint16               size) ;
```

Arguments

`IN` `MSGQ_AllocatorHandle` `mqaHandle`

Handle to the MSGQ allocator object.

`IN` `MSGQ_Msg` `msg`

Address of the message to be freed.

`IN` `Uint16` `size`

Size in MADUs of the message to be freed.

Return Value

None.

Comments

This API is called during `MSGQ_free ()`. This function frees the specified message by adding it to the appropriate buffer pool. The buffer pool is identified based on the size of the message to be freed.

Constraints

None.

See Also

`MQABUF_State`
`MQABUF_alloc ()`

7.2 Remote MQT

7.2.1 Constants & Enumerations

7.2.1.1 ID_MQTDSPLINK_CTRL

This macro defines the internal ID used to identify control messages.

Definition

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

Comments

This ID is used as the destination ID within the message header, when the message is intended for the MQT, and not a particular MSGQ.

Constraints

None.

See Also

None.

7.2.1.2 MQTDSPLINK_NAME

This macro defines the MQT name for the DSPLINK messaging component.

Definition

```
#define MQTDSPLINK_NAME    "MQTDSPLINK"
```

Comments

The MQT name is used for debugging purposes.

Constraints

None.

See Also

`MQTDSPLINK_open ()`

7.2.1.3 DSPLINK_DSPMSGQ_NAME

This macro defines the prefix to the names of all MSGQs created on the DSP for communication with the GPP.

Definition

```
#define DSPLINK_DSPMSGQ_NAME    "DSPLINK_DSP00MSGQ"
```

Comments

The message queues on the DSP used for inter-processor transfer through DSPLINK must be created with specific names expected by the DSPLINK MQT. The names must be of the following format:

DSPLINK_DSP<PROCESSORID>MSGQ<MSGQID>

<MSGQID> can have values from 00 to 'n-1' where 'n' is the maximum number of message queues on the processor.

<PROCESSORID> corresponds to the processor id of the DSP (used on GPP side to reference each DSP). The processor ID ranges in value from 00 to 'n-1', where 'n' is the maximum number of DSPs in the system

Constraints

None.

See Also

None.

7.2.1.4 DSPLINK_GPPMSGQ_NAME

This macro defines the prefix to the names of all MSGQs created on the GPP for communication with the DSP.

Definition

```
#define DSPLINK_GPPMSGQ_NAME     "DSPLINK_GPPMSGQ"
```

Comments

The message queues on the GPP used for inter-processor transfer through DSPLINK must be located by the DSP application with specific names expected by the DSPLINK MQT. The names must be of the following format:

`DSPLINK_GPPMSGQ<MSGQID>`

`<MSGQID>` can have values from `00` to '`n-1`' where '`n`' is the maximum number of message queues on the GPP.

Constraints

None.

See Also

None.

7.2.1.5 MQTDSPLINK_CTRLMSG_SIZE

This constant defines the size (in MADUs) of control messages used within the remote DSPLINK MQT.

Definition

```
#define MQTDSPLINK_CTRLMSG_SIZE 32
```

Comments

The remote MQT uses the default allocator 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 consider this requirement while configuring the default allocator.

Constraints

The user must always use this constant when configuring the default allocator for the remote MQT. The user must not hard-code the size within the application. This allows future compatibility with later versions of the remote MQT, which may have a different control message size and format.

The required size for control messages is larger than the actual size, to allow for future extensions, and any changes in structure size due to packing.

See Also

`MQTDSPLINK_CtrlMsg`

7.2.1.6 MQTDSPLINK_CtrlCmd

This enumeration defines the types of control commands that are sent between the MQTs on different processors.

Definition

```
typedef enum {
    MqtCmdLocate     = 0,
    MqtCmdLocateAck  = 1,
    MqtCmdExit       = 2
} MQTDSPLINK_CtrlCmd ;
```

Comments

A control message has the destination ID as `ID_MQTDSPLINK_CTRL`, indicating that the message is meant for the MQT, and not a particular MSGQ. In that case, the identification of the type of control message is made through the message ID field in the message header, which is one of the types defined by this enumeration. The actual message content differs depending on the control command.

Constraints

None.

See Also

`MQTDSPLINK_CtrlMsg`

7.2.2 Typedefs & Data Structures

7.2.2.1 MQTDSPLINK_Params

This structure defines the attributes required for initialization of the DSPLINK MQT.

Definition

```
typedef struct MQTDSPLINK_Params_tag {
    Uint16           maxNumMsgq ;
    Uint16           maxMsgSize ;
    Uint16           defaultMqaId ;
    DSPLINK_LinkType physicalLink ;
} MQTDSPLINK_Params ;
```

Fields

`maxNumMsgq`	Maximum number of MSGQs that can be created on the remote processor.
`maxMsgSize`	Maximum message size (in MADUs) supported by the MQT.
`defaultMqaId`	The default MQA to be used by the remote MQT, in case the MQA ID within the message received from the DSP is invalid. This case can occur in case of a mismatch between allocators configured on the GPP and the DSP.
`physicalLink`	The physical link to be used by this MQT.

Comments

These parameters are provided to the MQT once during its initialization, which takes place during the call to `MSGQ_init ()`.

The default parameters are provided to the user through the following structure:

```
extern MQTDSPLINK_Params MQTDSPLINK_PARAMS ;
```

The definition is within the source file:

```
MQTDSPLINK_Params MQTDSPLINK_PARAMS = { 16,
                                        64,
                                        0,
                                        DSPLINK_LinkShm};
```

Constraints

None.

See Also

`MQTDSPLINK_open ()`

7.2.2.2 MQTDSPLINK_State

This structure defines the transport state object, which exists as a single instance for the remote MQT.

Definition

```
typedef struct MQTDSPLINK_State_tag {
    Uint16            mqtId ;
    Uint16            numRemoteMsgq ;
    MSGQ_Handle       remoteMsgqs ;
    Uint16            maxMsgSize ;
    Uint16            numLocalMsgq ;
    MSGQ_Handle       localMsgqs ;
    Ptr               getBuffer ;
    GIO_Handle        inpChan ;
    GIO_Handle        outChan ;
    GIO_AppCallback   getCallback ;
    GIO_AppCallback   putCallback ;
    QUE_Obj           msgQueue ;
    Uint16            defaultMqaId ;
} MQTDSPLINK_State ;
```

Fields

`mqtId`	ID of this MQT.
`numRemoteMsgq`	Maximum number of MSGQs that can be created on the remote processor.
`remoteMsgqs`	Array of remote MSGQ objects.
`maxMsgSize`	Maximum message size (in MADUs) supported by the MQT.
`numLocalMsgq`	Number of local MSGQs.
`localMsgqs`	Array of local MSGQ objects.
`getBuffer`	The buffer to be used for priming the input channel.
`inpChan`	Handle to the input channel used for receiving messages from the GPP.
`outChan`	Handle to the output channel used for sending messages to the GPP.
`getCallback`	GIO callback object for submissions on the input channel.
`putCallback`	GIO callback object for submissions on the output channel.
`msgQueue`	Message repository to queue pending messages.
`defaultMqaId`	The default MQA to be used by the remote MQT.

Comments

An instance of this object is created and initialized during `MQTDSPLINK_open ()`, and its handle is returned to the caller. It contains all information required for maintaining the state of the MQT.

Constraints

None.

See Also

`MQTDSPLINK_open ()`

7.2.2.3 MQTDSPLINK_Obj

This structure defines the transport object, which has an instance for every MSGQ created on the processor.

Definition

```
typedef struct MQTDSPLINK_Obj_tag {
    SEM_Obj     locateSem ;
} MQTDSPLINK_Obj, *MQTDSPLINK_Handle ;
```

Fields

`locateSem`	Semaphore used during mqtLocate ().

Comments

One instance of the MQT object is created for every message queue. The MQT object corresponding to a receiver queue is created during `MSGQ_create ()`. For all remote queues (senders), the MQT objects are created during the `mqtOpen ()` function, along with all the remote MSGQ objects.

Constraints

None.

See Also

`MQTDSPLINK_open ()`

7.2.2.4 MQTDSPLINK_CtrlMsg

This structure defines the format of the control messages that are sent between the MQTs on different processors.

Definition

```
typedef struct MQTDSPLINK_CtrlMsg_tag {
    MSGQ_MsgHeader msgHeader ;
    union {
        struct {
            Uint16      msgqId ;
            Uint16      mqaId;
            Uint32      timeout;
            Uint32      replyHandle;
            Uint32      arg;
            Uint32      semHandle ;
        } locateMsg ;

        struct {
            Uint16      msgqId ;
            Uint16      msgqFound ;
            Uint16      mqaId;
            Uint32      timeout;
            Uint32      replyHandle;
            Uint32      arg;
            Uint32      semHandle ;
        } locateAckMsg ;
    } ctrlMsg ;
} MQTDSPLINK_CtrlMsg ;
```

Fields

`msgHeader`	Fixed message header required for all messages.

`ctrlMsg` Defines the format of the different control messages.

```
locateMsg:
    msgqId      -> ID of the MSGQ to be located on
                   the remote processor.
    mqaId       -> MQA ID to allocate async response
                   messages
    timeout     -> Timeout value for sync locate
    replyHandle -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    semHandle   -> Semaphore handle for sync locate
locateAckMsg:
    msgqId      -> ID of the MSGQ located on the
                   remote processor.
    msgqFound   -> Requested MSGQ was found on the
                   remote processor?
                   0 -> Not found,
                   1 -> Found
    mqaId       -> MQA ID to allocate async response
                   messages
    timeout     -> Timeout value for sync locate
    replyHandle -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    semHandle   -> Semaphore handle for sync locate
```

No control message is required when the command indicates exit of the remote MQT.

Comments

The control messages are used for communication between the MQTs. They contain information for MSGQ location and exit notification.

Constraints

None.

See Also

`MQTDSPLINK_CtrlCmd`

7.2.3 API Definition

The MQT APIs are exposed to MSGQ through a function table:

```
MSGQ_TransportFxns MQTDSPLINK_FXNS = {
    &MQTDSPLINK_open,
    &MQTDSPLINK_close,
    &MSGQ_MQTCREATENOTIMPL,
    &MQTDSPLINK_locate,
    &MSGQ_MQTDELETENOTIMPL,
    &MQTDSPLINK_release,
    &MSGQ_MQTGETNOTIMPL,
    &MQTDSPLINK_put,
    &MQTDSPLINK_getReplyHandle,
} ;
```

7.2.3.1 MQTDSPLINK_init

This function performs global initialization of the remote MQT.

Syntax

```
Void MQTDSPLINK_init () ;
```

Arguments

None.

Return Value

None.

Comments

The `mqtInit ()` and `mqtExit ()` functions are directly exposed by the MQT to the application. The `MQTDSPLINK_init ()` function is called by the application to perform any global initialization required for the remote MQT.

Constraints

None.

See Also

`MQTDSPLINK_exit ()`

7.2.3.2 MQTDSPLINK_exit

This function performs global finalization of the remote MQT.

Syntax

```
Void MQTDSPLINK_exit () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called by the application to perform any global finalization required for the remote MQT.

Constraints

None.

See Also

`MQTDSPLINK_init ()`

7.2.3.3 MQTDSPLINK_open

This function opens the remote MQT and configures it according to the user attributes.

Syntax

```
MSGQ_Status MQTDSPLINK_open (MSGQ_TransportHandle  mqtHandle) ;
```

Arguments

IN	MSGQ_TransportHandle	mqtHandle

Handle to the MSGQ transport object.

Return Value

MSGQ_SUCCESS	This component has been successfully opened.
MSGQ_EMEMORY	Failure during memory operation.
MSGQ_EFAILURE	Failure in opening the component.

Comments

This API is called during MSGQ_init (). It carries out all initialization required for the MQT. This function is called only once for the MQT before any of its other functions can be called.

It creates and initializes an instance of the state object MQTDSPLINK_State, and sets it in the MSGQ transport object.

This function expects certain attributes from the user, which are defined by the MQTDSPLINK_Params structure.

Constraints

This function cannot be called from SWI or HWI context.

See Also

MQTDSPLINK_State
MQTDSPLINK_Params
MQTDSPLINK_close ()

7.2.3.4 MQTDSPLINK_close

This function closes the remote MQT, and cleans up its state object.

Syntax

```
MSGQ_Status MQTDSPLINK_close (MSGQ_TransportHandle mqtHandle) ;
```

Arguments

`IN` `MSGQ_TransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

Return Value

`MSGQ_SUCCESS`	This component has been successfully closed.
`MSGQ_EMEMORY`	Failure during memory operation.
`MSGQ_EFAILURE`	Failure in closing the component.

Comments

This API is called during `MSGQ_transportClose ()`. It carries out any other required actions for finalizing the MQT.

After successful completion of this function, no further MQT services shall be available from the remote MQT.

Constraints

This function cannot be called from SWI or HWI context.

See Also

`MQTDSPLINK_State`
`MQTDSPLINK_open ()`

7.2.3.5 MQTDSPLINK_locate

This function attempts to locate the specified message queue on the remote processor.

Syntax

```
MSGQ_Status MQTDSPLINK_locate (MSGQ_TransportHandle mqtHandle,
                               String               queueName,
                               MSGQ_Handle *        msgqHandle,
                               MSGQ_LocateAttrs     attrs) ;
```

Arguments

IN	MSGQ_TransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	String	queueName

Name of the MSGQ to be located.

OUT	MSGQ_Handle *	msgqHandle

Location to receive the handle to the located message queue.

IN OPT	MSGQ_LocateAttrs	attrs

Optional attributes for location of the MSGQ.

Return Value

MSGQ_SUCCESS	The message queue has been successfully located.
MSGQ_ENOTFOUND	The message queue does not exist on the remote processor.
MSGQ_ETIMEOUT	Timeout during location of the MSGQ.
MSGQ_EMEMORY	Failure during memory operation.
MSGQ_EFAILURE	Failure during location of the MSGQ.

Comments

This API is called during MSGQ_locate (). After this API has been successfully called, the returned handle can be used for further actions on the MSGQ, including sending messages to it.

There are two ways of calling this API: Synchronous and asynchronous.

Synchronous: The API call is considered synchronous if the attrs parameter is NULL, or the replyHandle field in the attrs is NULL. When called synchronously, the msgqHandle parameter is used for returning the located MSGQ. The API blocks until the remote MSGQ has been located.

Asynchronous: The API call is considered asynchronous if the attrs parameter is non-NULL, and the replyHandle field in the attrs is also non-NULL. In this case, the msgqHandle parameter may be NULL. The API is non-blocking, and returns after

issuing a locate request to the remote processor. On receiving the locate acknowledgement, the MQT creates and fills an asyncLocator message, and sends it to the reply MSGQ specified by the user.

Constraints

The default allocator specified by the user for internal use by this MQT must be configured before this function can be called.

If called in the synchronous mode, this function cannot be called from the main () function, or SWI/HWI context.

See Also

MQTDSPLINK_Obj
MQTDSPLINK_release ()

7.2.3.6 MQTDSPLINK_release

This function releases the MSGQ located during an earlier `MSGQ_locate ()` or `MSGQ_getReplyHandle ()` call.

Syntax

```
MSGQ_Status MQTDSPLINK_release (MSGQ_Handle msgqHandle) ;
```

Arguments

IN	MSGQ_Handle	msgqHandle

Handle to the MSGQ to be released.

Return Value

MSGQ_SUCCESS	The message queue has been successfully released.
MSGQ_EFAILURE	Failure in releasing the message queue.

Comments

This API is called during `MSGQ_release ()`. After this API has been successfully called, the MSGQ needs to be located again before sending messages to it.

This function releases any resources allocated during the call to locate the remote MSGQ.

Constraints

None.

See Also

`MQTDSPLINK_locate ()`

7.2.3.7 MQTDSPLINK_put

This function sends a message to the specified remote MSGQ.

Syntax

```
MSGQ_Status MQTDSPLINK_put (MSGQ_Handle      msgqHandle,
                            MSGQ_Msg         msg) ;
```

Arguments

`IN` `MSGQ_Handle` `msgqHandle`

Handle to the MSGQ to which the message is to be sent.

`IN` `MSGQ_Msg` `msg`

Pointer to the message to be sent.

Return Value

`MSGQ_SUCCESS`	The message has been successfully sent.
`MSGQ_EFAILURE`	Failure in sending the message.

Comments

This API is called during `MSGQ_put ()`. This function is non-blocking and deterministic, so the message is queued up on the message repository for the required MSGQ. If the MSGQ was initially empty, an asynchronous `GIO_submit ()` is also called to issue the message to the IOM driver. On receiving the callback after the message has been transferred over the physical link by the MQT, the message is freed, and the next pending message on the MSGQ (if any) is submitted to the IOM driver.

Constraints

None.

See Also

`MQTDSPLINK_putCallback ()`

7.2.3.8 MQTDSPLINK_getReplyHandle

This function returns the MSGQ handle to be used for replying to a message received from a remote application.

Syntax

```
MSGQ_Handle MQTDSPLINK_getReplyHandle (MSGQ_TransportHandle mqtHandle,
                                       MSGQ_Msg             msg) ;
```

Arguments

IN	MSGQ_TransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MSGQ_Msg	msg

Message whose reply handle is to be obtained.

Return Value

Handle to the reply message queue	The source MSGQ ID has been successfully obtained for reply.
NULL	No source MSGQ ID has been specified, or an error occurred.

Comments

This API is called during MSGQ_getReplyHandle (). It is used when the application wishes to send a reply message back to the application that had sent the message. If an application expects a reply message, it must specify the MSGQ ID of a local MSGQ for receiving the reply message from the remote MSGQ.

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

In case the reply MSGQ was not previously located, this function also updates the state of the remote MSGQ object corresponding to the reply MSGQ. This remote MSGQ object can be released through a call to MSGQ_release ().

Constraints

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

See Also

MQTDSPLINK_release ()

7.2.3.9 MQTDSPLINK_putCallback

This function implements the callback that runs when the message to be sent to a remote MSGQ has been transferred across the physical link.

Syntax

```
Void MQTDSPLINK_putCallback (Ptr arg, Int status, Ptr bufp, Uns size) ;
```

Arguments

`IN` `Ptr` `arg`

Argument to the callback function.

`IN` `Int` `status`

Status of completion of the transfer packet.

`IN` `Ptr` `bufp`

Pointer to the submitted buffer.

`IN` `Uns` `size`

Size of the submitted buffer.

Return Value

None.

Comments

This callback function is used during `MQTDSPLINK_put ()` while submitting the message to the IOM driver. On receiving the callback after the message has been transferred over the physical link by the MQT, the message is freed, and the next pending message on the MSGQ (if any) is submitted to the IOM driver.

This function is not part of the standard MQT interface expected by the MSGQ module.

Constraints

None.

See Also

`MQTDSPLINK_put ()`

7.2.3.10 MQTDSPLINK_getCallback

This function implements the callback that runs when the message has been received from the GPP.

Syntax

```
Void MQTDSPLINK_getCallback (Ptr arg, Int status, Ptr bufp, Uns size) ;
```

Arguments

IN	Ptr	arg

Argument to the callback function. This is the MSGQ handle.

IN	Int	status

Status of completion of the transfer packet.

IN	Ptr	bufp

Pointer to the submitted buffer.

IN	Uns	size

Size of the submitted buffer.

Return Value

None.

Comments

During the initialization of the MQT in MQTDSPLINK_open (), a buffer of the maximum size supported by the MQT is submitted on the messaging channel reserved for receiving messages from the GPP. Whenever a message is received from the GPP, it is copied into the buffer, and this callback function is called. On receiving the callback, a new message of the actual message size is allocated, the message copied into it, and the same big buffer reissued on the channel.

This function is not part of the standard MQT interface expected by the MSGQ module.

Constraints

None.

See Also

MQTDSPLINK_open ()

7.3 Physical link

The physical link driver shall not provide separate commands for messaging. Messaging shall be supported within the link through the same commands provided for data transfer. All physical links that are used by the remote MQT for communicating with other processors shall ensure the following:

1. Two channels shall be reserved for messaging, one for messages from GPP to DSP, and the other for messages from DSP to GPP
2. Messaging shall be prioritized over data transfer. This implies that the link shall always check the messaging channels for any pending transfers before the data channels.

8 GPP side

8.1 API

8.1.1 Constants & Enumerations

8.1.1.1 ID_LOCAL_PROCESSOR

This constant defines the ID of the local processor (GPP).

Definition

```
#define ID_LOCAL_PROCESSOR    (ProcessorId) 0xFFFF
```

Comments

The local processor ID is used for specifying messaging operations for MSGQs on the GPP. This ID is passed to the `MSGQ_Locate ()` and `MSGQ_Put ()` APIs as the `processorId` to specify local messaging operations.

Constraints

None.

See Also

```
MSGQ_Locate ()
MSGQ_Put ()
```

8.1.1.2 MSGQ_INTERNAL_ID_START

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

Definition

```
#define MSGQ_INTERNAL_ID_START (Uint16) 0xFE00
```

Comments

None.

Constraints

None.

See Also

None.

8.1.1.3 MSGQ_ASYNC_ERROR_MSGID

This constant defines the asynchronous error message id.

Definition

```
#define MSGQ_ASYNC_ERROR_MSGID (Uint16) 0xFE01
```

Comments

None.

Constraints

None.

See Also

None.

8.1.1.4 MSGQ_INTERNAL_ID_END

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

Definition

```
#define MSGQ_INTERNAL_ID_END (Uint16) 0xFEFF
```

Comments

None.

Constraints

None.

See Also

None.

8.1.1.5 MSGQ_MQT_MSGID_START

This constant defines the start of transport message id range.

Definition

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

Comments

None.

Constraints

None.

See Also

None.

8.1.1.6 MSGQ_MQT_MSGID_END

This constant defines the end of transport message id range.

Definition

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

Comments

None.

Constraints

None.

See Also

None.

8.1.1.7 MSGQ_INVALID_ID

This constant defines the invalid ID for MSGQ, MQT, MQA and messages.

Definition

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

Comments

None.

Constraints

None.

See Also

None.

8.1.1.8 RMQT_CTRLMSG_SIZE

This constant defines the size (in bytes) of control messages used within the remote MQT.

Definition

```
#define RMQT_CTRLMSG_SIZE 64
```

Comments

The remote MQT uses the default allocator 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 consider this requirement while configuring the default allocator.

Constraints

The user must always use this constant when configuring the default allocator for the remote MQT. The user must not hard-code the size within the application. This allows future compatibility with later versions of the remote MQT, which may have a different control message size and format.

The required size for control messages is larger than the actual size, to allow for future extensions, and any changes in structure size due to packing.

See Also

`RmqtCtrlMsg`

8.1.1.9 MAX_MSGQS

Maximum number of message queues that can be created on the GPP.

Definition

```
#define MAX_MSGQS 32
```

Comments

None.

Constraints

None.

See Also

None.

8.1.1.10 MSG_HEADER_RESERVED_SIZE

Size of the reserved field in the message header.

Definition

```
#define MSG_HEADER_RESERVED_SIZE 2
```

Comments

None.

Constraints

None.

See Also

None.

8.1.1.11 MSGQ_GetMsgId

This macro returns the message ID of the specified message.

Definition

```
#define MSGQ_GetMsgId(msg) ((MsgqMsg) (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

`MsgqMsgHeader`
`MSGQ_GetMsgSize`

8.1.1.12 MSGQ_GetMsgSize

This macro returns the size of the specified message.

Definition

```
#define MSGQ_GetMsgSize(msg) ((MsgqMsg) (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

`MsgqMsgHeader`
`MSGQ_GetMsgId`

8.1.1.13 MsgqErrorType

This enumeration defines the possible types of asynchronous error messages.

Definition

```
typedef enum {
    MsgqErrorType_MqtExit   = 0,
    MsgqErrorType_PutFailed = 1
} MsgqErrorType ;
```

Comments

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 error types may have any one of the values from this enumeration.

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

Error Type	arg1	arg2	arg3
MsgqErrorType_MqtExit	ID of the transport that sent the exit message	Not used	Not used
MsgqErrorType_PutFailed	ID of the processor on which the destination message queue exists	ID of the destination message queue on which the put failed	Status of the MSGQ_Put call that failed

Constraints

None.

See Also

MsgqAsyncErrorMsg
MSGQ_SetErrorHandler

8.1.2 Typedefs & Data Structures

8.1.2.1 MsgQueueId

This type is used for identifying the different message queues used by DSPLINK.

Definition

```
typedef Uint16     MsgQueueId ;
```

Comments

All remote message queues shall be on different processors. To address a specific message queue, the processor ID shall be used.

Constraints

None.

See Also

```
MSGQ_Locate ()
MSGQ_Put ()
```

8.1.2.2 AllocatorId

This type is used for identifying the different allocators used by DSPLINK.

Definition

```
typedef Uint16     AllocatorId ;
```

Comments

None.

Constraints

None.

See Also

```
MSGQ_AllocatorOpen ()
MSGQ_AllocatorClose ()
MSGQ_Alloc ()
MSGQ_Free ()
```

8.1.2.3 TransportId

This type is used for identifying the different transports used by DSPLINK.

Definition

```
typedef Uint16     TransportId ;
```

Comments

None.

Constraints

None.

See Also

```
MSGQ_TransportOpen ()
MSGQ_TransportClose ()
```

8.1.2.4 MsgqMsgHeader

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

Definition

```
typedef struct MsgqMsgHeader_tag {
    Uint32    reserved [MSG_HEADER_RESERVED_SIZE] ;
    Uint16    mqtId ;
    Uint16    mqaId ;
    Uint16    size  ;
    Uint16    dstId ;
    Uint16    srcId ;
    Uint16    msgId ;
} MsgqMsgHeader, *MsgqMsg ;
```

Fields

`reserved`	Reserved for use by the MQT. The MQT typically uses them as a link for queuing the messages.
`mqtId`	ID of the MQT used for transporting this message.
`mqaId`	ID of the MQA 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 directly be modified by the user.

Constraints

None.

See Also

None

8.1.2.5 MsgqAttrs

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

Definition

```
typedef struct MsgqAttrs_tag {
    Uint16     dummy ;
} MsgqAttrs ;
```

Fields

dummy	Dummy placeholder field.

Comments

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

Constraints

None.

See Also

`MSGQ_Create ()`

8.1.2.6 MsgqLocateAttrs

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

Definition

```
typedef struct MsgqLocateAttrs_tag {
    Uint32     timeout ;
} MsgqLocateAttrs ;
```

Fields

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

Comments

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

Constraints

None.

See Also

`MSGQ_Locate ()`

8.1.2.7 MqaBufAttrs

This structure defines the attributes required for initialization of the buffer MQA.

Definition

```
typedef struct MqaBufAttrs_tag {
    Uint16      numBufPools ;
    Uint16 *    msgSize ;
    Uint16 *    numMsg ;
} MqaBufAttrs ;
```

Fields

`numBufPools`	Number of buffer pools to be configured in the MQA.
`msgSize`	Array of sizes (in bytes) of the messages in the buffer pools. This array is of size `numBufPools`.
`numMsg`	Array of number of messages in all buffer pools. This array is of size `numBufPools`.

Comments

These attributes are provided to the MQA once during its initialization, which takes place during the call to `MSGQ_AllocatorOpen ()`.

Constraints

None.

See Also

`MSGQ_AllocatorOpen ()`

8.1.2.8 LmqtAttrs

This structure defines the attributes for initialization of the local MQT.

Definition

```
typedef struct LmqtAttrs_tag {
    Uint16      maxNumMsgq ;
} LmqtAttrs ;
```

Fields

`maxNumMsgq`	Maximum number of MSGQs that can be created on the local processor.

Comments

These attributes are provided to the MQT once during its initialization, which takes place during the call to `MSGQ_TransportOpen ()`.

Constraints

None.

See Also

`MSGQ_TransportOpen ()`
`LMQT_Open ()`

8.1.2.9 RmqtAttrs

This structure defines the attributes for initialization of the remote MQT.

Definition

```
typedef struct RmqtAttrs_tag {
    Uint16         maxNumMsgq ;
    Uint16         maxMsgSize ;
    AllocatorId    defaultMqaId ;
} RmqtAttrs ;
```

Fields

`maxNumMsgq`	Maximum number of MSGQs that can be created on the remote processor.
`maxMsgSize`	Maximum message size (in bytes) supported by the MQT.
`defaultMqaId`	ID of the default MQA to be used by the remote MQT, in case the MQA ID within the message received from the DSP is invalid. This case can occur in case of a mismatch between allocators configured on the GPP and the DSP.

Comments

These attributes are provided to the MQT once during its initialization, which takes place during the call to `MSGQ_TransportOpen ()`.

Constraints

None.

See Also

`MSGQ_TransportOpen ()`
`RMQT_Open ()`

8.1.2.10 MsgqAsyncErrorMsg

This structure defines the asynchronous error message format.

Definition

```
typedef struct MsgqAsyncErrorMsg_tag {
    MsgqMsgHeader  header ;
    Uint16         errorType ;
    Pvoid          arg1 ;
    Pvoid          arg2 ;
    Pvoid          arg3 ;
} MsgqAsyncErrorMsg ;
```

Fields

`header`	Fixed message header required for all messages.
`errorType`	Type of error.
`arg1`	First argument dependent on the error type.
`arg2`	Second argument dependent on the error type.
`arg3`	Third argument dependent on the error type.

Comments

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

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

`MsgqErrorType`
`MSGQ_SetErrorHandler ()`

8.1.2.11 MsgqInstrument

This structure defines the instrumentation data for a message queue.

Definition

```
#if defined (DDSP_PROFILE)
typedef struct MsgqInstrument_tag {
    ProcessorId      procId ;
    MsgQueueId       msgqId ;
    Bool             isValid ;
    Uint32           transferred ;
    Uint32           queued ;
} MsgqInstrument ;
#endif /* if defined (DDSP_PROFILE) */
```

Fields

`procId`	Processor identifier.
`msgqId`	Message queue identifier.
`isValid`	Indicates if the message queue is valid.
`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 ()`

8.1.2.12 MsgqStats

This structure defines the instrumentation data for MSGQs on all processors in the system.

Definition

```
#if defined (DDSP_PROFILE)
typedef struct MsgqStats_tag {
    MsgqInstrument  localMsgqData [MAX_MSGQS] ;
    MsgqInstrument  msgqData [MAX_PROCESSORS] [MAX_MSGQS] ;
} MsgqStats ;
#endif /* if defined (DDSP_PROFILE) */
```

Fields

`localMsgqData`	Instrumentation data for the local MSGQs.
`msgqData`	Instrumentation data for the remote 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

`MsgqInstrument`
`MSGQ_Instrument ()`

8.1.3 API Definition

8.1.3.1 MSGQ_AllocatorOpen

This function initializes the allocator component.

Syntax

```
DSP_STATUS MSGQ_AllocatorOpen (AllocatorId mqaId, Pvoid mqaAttrs) ;
```

Arguments

IN	AllocatorId	mqaId

ID of the MQA to be opened.

IN	Pvoid	mqaAttrs

Attributes for initialization of the MQA component. The structure of the expected attributes is specific to an MQA.

Return Value

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

Comments

None.

Constraints

The static configuration of the MQA is done as part of the CFG. This includes configuration of the fixed attributes specific to each MQA, including its function table interface. This configuration also defines the IDs of the MQA. These IDs must be used while deciding the attributes required by each MQA.

A default allocator is implemented within DSPLINK, for usage during messaging. However, users may configure their own MQAs, with the following constraints, if they are used with MSGQs used for transferring messages through DSPLINK:

- The MQAs must not have dynamic allocation of memory.
- The MQA allocation and free functions must be callable from a DPC context.

See Also

```
MSGQ_AllocatorClose ()
```

8.1.3.2 MSGQ_AllocatorClose

This function finalizes the allocator component.

Syntax

```
DSP_STATUS MSGQ_AllocatorClose (AllocatorId mqaId) ;
```

Arguments

IN AllocatorId mqaId

ID of the MQA to be closed.

Return Value

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

Comments

None.

Constraints

None.

See Also

MSGQ_AllocatorOpen ()

8.1.3.3 MSGQ_TransportOpen

This function initializes the MQT component.

Syntax

```
DSP_STATUS MSGQ_TransportOpen (TransportId mqtId, Pvoid mqtAttrs) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT to be opened.

IN	Pvoid	mqtAttrs

Attributes for initialization of the MQT component. The structure of the expected attributes is specific to an MQT.

Return Value

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

Comments

None.

Constraints

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. 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 ()

8.1.3.4 MSGQ_TransportClose

This function finalizes the MQT component.

Syntax

```
DSP_STATUS MSGQ_TransportClose (TransportId mqtId) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT to be closed.

Return Value

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

Comments

None.

Constraints

The default allocator 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_TransportOpen ()

8.1.3.5 MSGQ_Create

This function creates the message queue to be used for receiving messages, identified through the specified MSGQ ID.

Syntax

```
DSP_STATUS MSGQ_Create (MsgQueueId msgqId, MsgqAttrs * msgqAttrs) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the message queue to be created.

IN OPT	MsgqAttrs *	msgqAttrs

Optional attributes for creation of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully created.
DSP_EINVALIDARG	Invalid argument.
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

MsgQueueId
MsgqAttrs
MSGQ_Delete ()
MSGQ_Locate ()

8.1.3.6 MSGQ_Delete

This function deletes the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS MSGQ_Delete (MsgQueueId msgqId) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the message queue to be deleted.

Return Value

DSP_SOK	The message queue has been successfully deleted.
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

MsgQueueId
MSGQ_Create ()

8.1.3.7 MSGQ_Locate

This function verifies the existence and status of the message queue identified by the specified MSGQ ID, on the specified processor.

Syntax

```
DSP_STATUS MSGQ_Locate (ProcessorId       processorId,
                        MsgQueueId        msgqId,
                        MsgqLocateAttrs * attrs) ;
```

Arguments

`IN` `ProcessorId` `processorId`

ID of the processor on which the MSGQ is to be located.

`IN` `MsgQueueId` `msgqId`

ID of the message queue to be located.

`IN` `MsgqLocateAttrs *` `attrs`

Attributes for location of the MSGQ.

Return Value

`DSP_SOK`	The message queue has been successfully located.
`DSP_ENOTFOUND`	The message queue does not exist on the specified processor.
`DSP_ETIMEOUT`	Timeout occurred while locating the MSGQ.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.
`DSP_EINVALIDARG`	Invalid Parameter passed.

Comments

This API is called to get the status of the MSGQs that may exist on a remote processor. Before sending a message to the remote MSGQ, its existence must be verified through this call.

For locating a local MSGQ, `ID_LOCAL_PROCESSOR` is used to specify the `processorId`.

Constraints

The default allocator 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

`MsgQueueId`

```
MsgqLocateAttrs
MSGQ_Put ()
MSGQ_Release ()
```

8.1.3.8 MSGQ_Release

This function releases the MSGQ located through an earlier MSGQ_Locate () or MSGQ_GetReplyId () call.

Syntax

```
DSP_STATUS MSGQ_Release (ProcessorId processorId, MsgQueueId msgqId) ;
```

Arguments

IN	ProcessorId	processorId

ID of the processor on which the MSGQ is to be released.

IN	MsgQueueId	msgqId

ID of 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.
DSP_EINVALIDARG	Invalid Parameter passed.

Comments

This API is the counterpart to the MSGQ_Locate () API, and reverses the activities performed during MSGQ_Locate (), including releasing any resources allocated during the call. It is also used to release any resources allocated during MSGQ_GetReplyId () for a remote MSGQ. Once the MSGQ has been released, it needs to be located once again before sending a message to it.

For releasing a local MSGQ, ID_LOCAL_PROCESSOR is used to specify the processorId.

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

Constraints

None.

See Also

ID_LOCAL_PROCESSOR
MsgQueueId
MSGQ_Locate ()

8.1.3.9 MSGQ_Alloc

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

Syntax

```
DSP_STATUS MSGQ_Alloc (AllocatorId mqaId, Uint16 size, MsgqMsg * msg) ;
```

Arguments

IN	AllocatorId	mqaId

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

IN	Uint16	size

Size of the message to be allocated.

OUT	MsgqMsg *	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

MsgqMsgHeader
MSGQ_Put ()

8.1.3.10 MSGQ_Free

This function frees a message.

Syntax

```
DSP_STATUS MSGQ_Free (MsgqMsg msg) ;
```

Arguments

IN	MsgqMsg	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. 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

MsgqMsgHeader
MSGQ_Get ()

8.1.3.11 MSGQ_Put

This function sends a message to the specified MSGQ on a particular processor.

Syntax

```
DSP_STATUS MSGQ_Put (ProcessorId processorId,
                     MsgQueueId  destMsgqId,
                     MsgqMsg     msg,
                     Uint16      msgId,
                     MsgQueueId  srcMsgqId) ;
```

Arguments

IN	ProcessorId	processorId

ID of the processor on which the destination MSGQ exists.

IN	MsgQueueId	destMsgqId

ID of the destination MSGQ.

IN	MsgqMsg	msg

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

IN OPT	Uint16	msgId

Optional message ID to be associated with the message.

IN OPT	MsgQueueId	srcMsgqId

Optional ID of the source MSGQ to receive reply messages.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	The message queue does not exist. This implies that the MSGQ has not been located before this call was made.
DSP_EFAIL	General failure.

Comments

This function must be non-blocking and deterministic, so the message is queued up on the message repository for the required MSGQ. For sending messages to a local MSGQ, ID_LOCAL_PROCESSOR is used to specify the processorId.

Constraints

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

See Also

ID_LOCAL_PROCESSOR

```
MsgqMsgHeader
MSGQ_Get ()
```

8.1.3.12 MSGQ_Get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS MSGQ_Get (MsgQueueId msgqId,
                     Uint32     timeout,
                     MsgqMsg *  msg) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the MSGQ on which the message is to be received.

IN	timeout	timeout

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

OUT	MsgqMsg *	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_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

MsgqMsgHeader
MSGQ_Put ()

8.1.3.13 MSGQ_GetReplyId

This function extracts the MSGQ ID and processor ID to be used for replying to a received message.

Syntax

```
DSP_STATUS MSGQ_GetReplyId (MsgqMsg       msg,
                            ProcessorId * procId,
                            MsgQueueId *  msgqId) ;
```

Arguments

IN	MsgqMsg	msg

Message, whose reply MSGQ ID is to be extracted.

OUT	ProcessorId *	procId

Location to retrieve the ID of the processor where the reply MSGQ resides.

OUT	MsgQueueId	msgqId

Location to retrieve the ID of the reply MSGQ.

Return Value

DSP_SOK	The reply information has been successfully retrieved.
DSP_ENOTFOUND	Reply information has not been provided by the sender.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EINVALIDARG	Invalid argument.
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 MSGQ ID of a local MSGQ for receiving the reply message from the remote MSGQ.

After getting the reply MSGQ and processor IDs, 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 ID.

See Also

MsgqMsgHeader

8.1.3.14 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 (MsgQueueId msgqId, Uint16 mqaId) ;
```

Arguments

IN	MsgQueueId	msgqId

Message queue to receive the error messages.

IN	Uint16	mqaId

ID indicating the allocator 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

MsgqErrorType
MsgqAsyncErrorMsg

8.1.3.15 MSGQ_Instrument

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

Syntax

```
DSP_STATUS MSGQ_Instrument (ProcessorId      procId,
                            MsgQueueId       msgqId,
                            MsgqInstrument * retVal) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

OUT	MsgqInstrument *	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

MsgqInstrument

8.1.3.16 MSGQ_Debug

This function prints the status of the MSGQ subcomponent.

Syntax

```
Void MSGQ_Debug (ProcessorId    procId,
                 MsgQueueId     msgqId) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

Return Value

None.

Comments

None.

Constraints

This function is defined only for debug builds.

See Also

None.

8.2 PMGR

8.2.1 API Definition

8.2.1.1 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 owner process. It passes down the call into the Link Driver layer.

Constraints

None.

See Also

LDRV_MSGQ_Setup ()

8.2.1.2 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 owner process. It passes down the call into the Link Driver layer.

Constraints

None.

See Also

LDRV_MSGQ_Destroy ()

8.2.1.3 PMGR_MSGQ_AllocatorOpen

This function initializes the MQA component.

Syntax

```
DSP_STATUS PMGR_MSGQ_AllocatorOpen (AllocatorId mqaId,
                                    Pvoid       mqaAttrs,
                                    Pvoid *     mqaInfo);
```

Arguments

IN	AllocatorId	mqaId

ID of the MQA to be opened.

IN	Pvoid	mqaAttrs

Attributes for initialization of the MQA component. The structure of the expected attributes is specific to an MQA.

OUT	Pvoid *	mqaInfo

Location to receive the handle of the initialized MQA state object.

Return Value

DSP_SOK	The MQA component has been successfully opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EINVALIDARG	Invalid Parameter passed.
DSP_EFAIL	General failure.

Comments

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

Constraints

None.

See Also

MSGQ_AllocatorOpen ()
LDRV_MSGQ_AllocatorOpen ()

8.2.1.4 PMGR_MSGQ_AllocatorClose

This function finalizes the MQA component.

Syntax

```
DSP_STATUS PMGR_MSGQ_AllocatorClose (AllocatorId mqaId) ;
```

Arguments

`IN` `AllocatorId` `mqaId`

ID of the MQA to be closed.

Return Value

`DSP_SOK`	The MQA 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

None.

See Also

`MSGQ_AllocatorClose ()`
`LDRV_MSGQ_AllocatorClose ()`

8.2.1.5 PMGR_MSGQ_TransportOpen

This function initializes the MQT component.

Syntax

```
DSP_STATUS PMGR_MSGQ_TransportOpen (TransportId mqtId,
                                    Pvoid       mqtAttrs) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT to be opened.

IN	Pvoid	mqtAttrs

Attributes for initialization of the MQT component. The structure of the expected attributes is specific to an MQT.

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

None.

See Also

MSGQ_TransportOpen ()
LDRV_MSGQ_TransportOpen ()

8.2.1.6 PMGR_MSGQ_TransportClose

This function finalizes the MQT component.

Syntax

```
DSP_STATUS PMGR_MSGQ_TransportClose (TransportId mqtId) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT 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

None.

See Also

```
MSGQ_TransportClose ()
LDRV_MSGQ_TransportClose ()
```

8.2.1.7 PMGR_MSGQ_Create

This function creates the message queue to be used for receiving messages, identified through the specified MSGQ ID.

Syntax

```
DSP_STATUS PMGR_MSGQ_Create (MsgQueueId  msgqId,
                             MsgqAttrs * msgqAttrs) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the message queue to be created.

IN OPT	MsgqAttrs *	msgqAttrs

Optional attributes for creation of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully created.
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

None.

See Also

MSGQ_Create ()
LDRV_MSGQ_Create ()

8.2.1.8 PMGR_MSGQ_Delete

This function deletes the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS PMGR_MSGQ_Delete (MsgQueueId msgqId) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the message queue to be deleted.

Return Value

DSP_SOK	The message queue has been successfully deleted.
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

None.

See Also

MSGQ_Delete ()
LDRV_MSGQ_Delete ()

8.2.1.9 PMGR_MSGQ_Locate

This function verifies the existence and status of the message queue identified by the specified MSGQ ID, on the specified processor.

Syntax

```
DSP_STATUS PMGR_MSGQ_Locate (ProcessorId       processorId,
                             MsgQueueId        msgqId,
                             MsgqLocateAttrs * attrs) ;
```

Arguments

IN	ProcessorId	processorId

ID of the processor on which the MSGQ is to be located.

IN	MsgQueueId	msgqId

ID of the message queue to be located.

IN	MsgqLocateAttrs *	attrs

Attributes for location of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully located.
DSP_ENOTFOUND	The message queue does not exist on the specified processor.
DSP_ETIMEOUT	Timeout occurred while locating the MSGQ.
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

None.

See Also

MSGQ_Locate ()
LDRV_MSGQ_Locate ()

8.2.1.10 PMGR_MSGQ_Release

This function releases the MSGQ located earlier.

Syntax

```
DSP_STATUS PMGR_MSGQ_Release (ProcessorId processorId,
                              MsgQueueId  msgqId) ;
```

Arguments

IN	ProcessorId	processorId

ID of the processor on which the MSGQ is to be released.

IN	MsgQueueId	msgqId

ID of 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_EFAIL	General failure.

Comments

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

Constraints

None.

See Also

MSGQ_Release ()
LDRV_MSGQ_Release ()

8.2.1.11 PMGR_MSGQ_Alloc

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

Syntax

```
DSP_STATUS PMGR_MSGQ_Alloc (AllocatorId mqaId,
                            Uint16      size,
                            MsgqMsg *   msg) ;
```

Arguments

IN	AllocatorId	mqaId

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

IN	Uint16	size

Size of the message to be allocated.

OUT	MsgqMsg *	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

None.

See Also

MSGQ_Alloc ()
LDRV_MSGQ_Alloc ()

8.2.1.12 PMGR_MSGQ_Free

This function frees a message.

Syntax

```
DSP_STATUS PMGR_MSGQ_Free (MsgqMsg msg) ;
```

Arguments

IN	MsgqMsg	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

None.

See Also

MSGQ_Free ()
LDRV_MSGQ_Free ()

8.2.1.13 PMGR_MSGQ_Put

This function sends a message to the specified MSGQ on a particular processor.

Syntax

```
DSP_STATUS PMGR_MSGQ_Put (ProcessorId processorId,
                          MsgQueueId  destMsgqId,
                          MsgqMsg     msg,
                          Uint16      msgId,
                          MsgQueueId  srcMsgqId) ;
```

Arguments

IN	ProcessorId	processorId

ID of the processor on which the MSGQ to which the message is to be sent, exists.

IN	MsgQueueId	destMsgqId

ID of the destination MSGQ.

IN	MsgqMsg	msg

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

IN OPT	Uint16	msgId

Optional message ID to be associated with the message.

IN OPT	MsgQueueId	srcMsgqId

Optional ID of the source MSGQ to receive reply messages.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_ENOTFOUND	The message queue does not exist. This implies that the MSGQ has not been located before this call was made.
DSP_EFAIL	General failure.

Comments

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

Constraints

None.

See Also

MSGQ_Put ()
LDRV_MSGQ_Put ()

8.2.1.14 PMGR_MSGQ_Get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS PMGR_MSGQ_Get (MsgQueueId msgqId,
                          Uint32     timeout,
                          MsgqMsg *  msg) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the MSGQ on which the message is to be received.

IN	timeout	timeout

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

OUT	MsgqMsg *	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_EFAIL	General failure.

Comments

After validating the MSGQ ownership, this function passes on the call from the API layer to the Link Driver layer.

Constraints

None.

See Also

MSGQ_Get ()
LDRV_MSGQ_Get ()

8.2.1.15 PMGR_MSGQ_GetReplyId

This function extracts the MSGQ ID and processor ID to be used for replying to a received message.

Syntax

```
DSP_STATUS PMGR_MSGQ_GetReplyId (MsgqMsg       msg,
                                 ProcessorId * procId,
                                 MsgQueueId *  msgqId) ;
```

Arguments

IN	MsgqMsg	msg

Message, whose reply MSGQ ID is to be extracted.

OUT	ProcessorId *	procId

Location to retrieve the ID of the processor where the reply MSGQ resides.

OUT	MsgQueueId	msgqId

Location to retrieve the ID of the reply MSGQ.

Return Value

DSP_SOK	The reply information has been successfully retrieved.
DSP_ENOTFOUND	Reply information has not been provided by the sender.
DSP_EFAIL	General failure.
DSP_EMEMORY	Operation failed due to a memory error.

Comments

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

Constraints

None.

See Also

MSGQ_GetReplyId ()
LDRV_MSGQ_GetReplyId ()

8.2.1.16 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 (MsgQueueId msgqId,
                                      Uint16     mqaId) ;
```

Arguments

IN	MsgQueueId	msgqId

Message queue to receive the error messages.

IN	Uint16	mqaId

ID indicating the allocator 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

None.

See Also

MSGQ_SetErrorHandler
LDRV_MSGQ_SetErrorHandler

8.2.1.17 PMGR_MSGQ_Instrument

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

Syntax

```
DSP_STATUS PMGR_MSGQ_Instrument (ProcessorId      procId,
                                 MsgQueueId       msgqId,
                                 MsgqInstrument * retVal) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

OUT	MsgqInstrument *	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.

See Also

MsgqInstrument
MSGQ_Instrument
LDRV_MSGQ_Instrument

8.2.1.18 PMGR_MSGQ_Debug

This function prints the status of the MSGQ subcomponent.

Syntax

```
Void PMGR_MSGQ_Debug (ProcessorId procId, MsgQueueId msgqId) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

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.

See Also

MSGQ_Debug
LDRV_MSGQ_Debug

8.3 LDRV – MSGQ

8.3.1 Constants & Enumerations

8.3.1.1 ID_MSGCHNL_TO_DSP

This macro defines the ID of the messaging channel to the DSP.

Definition

```
#define ID_MSGCHNL_TO_DSP MAX_CHANNELS
```

Comments

None.

Constraints

None.

See Also

`ID_MSGCHNL_FM_DSP`

8.3.1.2 ID_MSGCHNL_FM_DSP

This macro defines the ID of the messaging channel from the DSP.

Definition

```
#define ID_MSGCHNL_FM_DSP (MAX_CHANNELS + 1)
```

Comments

None.

Constraints

None.

See Also

ID_MSGCHNL_TO_DSP

8.3.1.3 DSPLINK_DSPMSGQ_NAME

This macro defines the prefix to the names of all MSGQs created on the DSP for communication with the GPP.

Definition

```
#define DSPLINK_DSPMSGQ_NAME    "DSPLINK_DSP00MSGQ"
```

Comments

The message queues on the DSP used for inter-processor transfer through DSPLINK must be created with specific names expected by the DSPLINK MQT. The names must be of the following format:

DSPLINK_DSP<PROCESSORID>MSGQ<MSGQID>

<MSGQID> can have values from 00 to 'n-1' where 'n' is the maximum number of message queues on the processor.

<PROCESSORID> corresponds to the processor id of the DSP (used on GPP side to reference each DSP). The processor ID ranges in value from 00 to 'n-1', where 'n' is the maximum number of DSPs in the system

Constraints

This macro is defined only for debug build.

See Also

None.

8.3.1.4 DSPLINK_GPPMSGQ_NAME

This macro defines the prefix to the names of all MSGQs created on the GPP for communication with the DSP.

Definition

```
#define DSPLINK_GPPMSGQ_NAME     "DSPLINK_GPPMSGQ"
```

Comments

The message queues on the GPP used for inter-processor transfer through DSPLINK must be located by the DSP application with specific names expected by the DSPLINK MQT. The names must be of the following format:

DSPLINK_GPPMSGQ<MSGQID>

<MSGQID> can have values from 00 to 'n-1' where 'n' is the maximum number of message queues on the GPP.

Constraints

This macro is defined only for debug build.

See Also

None.

8.3.1.5 LdrvMsgqStatus

This enumeration defines the possible states of the MSGQ object.

Definition

```
typedef enum {
    LdrvMsgqStatus_Empty         = 0,
    LdrvMsgqStatus_Inuse         = 1,
    LdrvMsgqStatus_LocatePending = 2
} LdrvMsgqStatus ;
```

Comments

None.

Constraints

None.

See Also

LdrvMsgqObject

8.3.2 Typedefs & Data Structures

8.3.2.1 FnMqaInit

This type defines the MQA initialization function.

Definition

```
typedef Void (*FnMqaInit) () ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

`MqaInterface`

8.3.2.2 FnMqaExit

This type defines the MQA finalization function.

Definition

```
typedef Void (*FnMqaExit) () ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

`MqaInterface`

8.3.2.3 FnMqaOpen

This type defines the MQA open function.

Definition

```
typedef DSP_STATUS (*FnMqaOpen) (LdrvMsgqAllocatorHandle mqaHandle,
                                 Pvoid                   mqaAttrs) ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

MqaInterface

8.3.2.4 FnMqaClose

This type defines the MQA close function.

Definition

```
typedef DSP_STATUS (*FnMqaClose) (LdrvMsgqAllocatorHandle mqaHandle) ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

`MqaInterface`

8.3.2.5 FnMqaAlloc

This type defines the MQA function for allocating a message.

Definition

```
typedef DSP_STATUS (*FnMqaAlloc) (LdrvMsgqAllocatorHandle mqaHandle,
                                  Uint16 *                size,
                                  MsgqMsg *               addr) ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

`MqaInterface`

8.3.2.6 FnMqaFree

This type defines the MQA function for freeing a message.

Definition

```
typedef DSP_STATUS (*FnMqaFree) (LdrvMsgqAllocatorHandle mqaHandle,
                                 MsgqMsg                 addr,
                                 Uint16                  size) ;
```

Comments

This function type is part of the MQA interface table.

Constraints

None.

See Also

`MqaInterface`

8.3.2.7 FnMqtInit

This type defines the MQT initialization function.

Definition

```
typedef Void (*FnMqtInit) () ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.8 FnMqtExit

This type defines the MQT finalization function.

Definition

```
typedef Void (*FnMqtExit) () ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.9 FnMqtOpen

This type defines the MQT open function.

Definition

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

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.10 FnMqtClose

This type defines the MQT close function.

Definition

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

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.11 FnMqtCreate

This type defines the MQT function for creating a MSGQ.

Definition

```
typedef DSP_STATUS (*FnMqtCreate) (LdrvMsgqTransportHandle mqtHandle,
                                   MsgQueueId              msgqId,
                                   LdrvMsgqHandle *        msgqHandle,
                                   MsgqAttrs *             attrs) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.12 FnMqtLocate

This type defines the MQT function for locating a MSGQ.

Definition

```
typedef DSP_STATUS (*FnMqtLocate) (LdrvMsgqTransportHandle mqtHandle,
                                   MsgQueueId              msgqId,
                                   MsgqLocateAttrs *       attrs) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.13 FnMqtDelete

This type defines the MQT function for deleting a MSGQ.

Definition

```
typedef DSP_STATUS (*FnMqtDelete) (LdrvMsgqTransportHandle mqtHandle,
                                   MsgQueueId              msgqId) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

MqtInterface

8.3.2.14 FnMqtRelease

This type defines the MQT function for releasing a MSGQ.

Definition

```
typedef DSP_STATUS (*FnMqtRelease) (LdrvMsgqTransportHandle mqtHandle,
                                    MsgQueueId              msgqId) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

MqtInterface

8.3.2.15 FnMqtGet

This type defines the MQT function for receiving a message.

Definition

```
typedef DSP_STATUS (*FnMqtGet) (LdrvMsgqTransportHandle mqtHandle,
                                MsgQueueId              msgqId,
                                Uint32                  timeout,
                                MsgqMsg *               msg) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

MqtInterface

8.3.2.16 FnMqtPut

This type defines the MQT function for sending a message.

Definition

```
typedef DSP_STATUS (*FnMqtPut) (LdrvMsgqTransportHandle mqtHandle,
                                MsgQueueId              msgqId,
                                MsgqMsg                 msg) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.17 FnMqtGetReplyId

This type defines the MQT function for getting the reply MSGQ ID for a particular message.

Definition

```
typedef
DSP_STATUS (*FnMqtGetReplyId) (LdrvMsgqTransportHandle mqtHandle,
                               MsgqMsg                 msg,
                               MsgQueueId *            msgqId) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.18 FnMqtGetById

This type defines the MQT function for receiving a message having a particular MSG ID.

Definition

```
typedef DSP_STATUS (*FnMqtGetById) (MsgQueueId  msgqId,
                                    Uint16 *    msgIds,
                                    Uint16      numIds,
                                    Uint32      timeout,
                                    MsgqMsg *   msg) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MqtInterface`

8.3.2.19 FnMqtInstrument

This type defines the MQT function for receiving instrumentation statistics.

Definition

```
typedef
DSP_STATUS (*FnMqtInstrument) (LdrvMsgqTransportHandle mqtHandle,
                               MsgQueueId              msgqId,
                               MsgqInstrument *        retVal) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

`MsgqInstrument`

8.3.2.20 FnMqtDebug

This type defines the MQT function for printing debug information.

Definition

```
typedef DSP_STATUS (*FnMqtDebug) (LdrvMsgqTransportHandle mqtHandle,
                                  MsgQueueId              msgqId) ;
```

Comments

This function type is part of the MQT interface table.

Constraints

None.

See Also

MqtInterface

8.3.2.21 LdrvMsgqState

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

Definition

```
typedef struct LdrvMsgqState_tag {
    LdrvMsgqAllocatorObj * allocators ;
    LdrvMsgqTransportObj * transports ;
    Uint16                 numAllocators ;
    Uint16                 numTransports ;
    Uint16                 localTransportId ;
    Uint16                 mqtMap [MAX_PROCESSORS] ;
    MsgQueueId             errorHandlerMsgq ;
    AllocatorId            errorMqaId ;
} LdrvMsgqState ;
```

Fields

`allocators`	Array of allocator objects.
`transports`	Array of transport objects, one for every processor in the system. This includes the local processor, as well as any other processors to which the local processor is connected.
`numAllocators`	Number of allocators configured in the system.
`numTransports`	Number of transports configured in the system.
`localTransportId`	ID of the local transport. The local MQT ID is used to index into the MQT objects table whenever the local MQT needs to be accessed.
`mqtMap`	Mapping of the processor ID to the MQT ID. This information is obtained through the CFG, and used for converting the processor ID into the MQT ID during MSGQ API calls.
`errorHandlerMsgq`	ID of 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_INVALID_ID`.
`errorMqaId`	ID of the allocator 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 `MSGQ_INVALID_ID`.

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 ()

8.3.2.22 LdrvMsgqObject

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

Definition

```
typedef struct LdrvMsgqObject_tag {
    Uint16                  msgqId ;
    SyncSemObject *         getSem ;
    LdrvMsgqTransportHandle mqtHandle ;
    Pvoid                   mqtRepository ;
    FnMqtGet                mqtGet ;
    FnMqtPut                mqtPut ;
    LdrvMsgqStatus          msgqStatus ;
} LdrvMsgqObject, *LdrvMsgqHandle ;
```

Fields

`msgqId`	ID of the MSGQ.
`getSem`	Pointer to the semaphore to be used for waiting for messages on this MSGQ.
`mqtHandle`	Handle to the MQT object that manages this MSGQ. There is a single instance of the MQT object for every MQT in the system.
`mqtRepository`	Handle to the MQT instance for this MSGQ. This object is specific to each MQT, and contains all information needed by it for interaction with this MSGQ. There is one instance of this object for every MSGQ.
`mqtGet`	Pointer to the `mqtGet ()` function of the transport. This pointer is replicated within this structure for faster access in time critical `MSGQ_Get ()` function.
`mqtPut`	Pointer to the `mqtPut ()` function of the transport. This pointer is replicated within this structure for faster access in time critical `MSGQ_Put ()` function.
`msgqStatus`	State of the MSGQ.

Comments

The MSGQ object is created during the create function of the local MQT. The MSGQ objects for remote MSGQs are created during calls to the locate functions of the corresponding MQTs.

The semaphore `getSem` within the MSGQ object is a binary semaphore.

Constraints

None.

See Also

`LdrvMsgqTransportObj`

```
MqtInterface
MsgqState
LMQT_Create ()
RMQT_Locate ()
```

8.3.2.23 LdrvMsgqTransportObj

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

Definition

```
typedef struct LdrvMsgqTransportObj_tag {
#if defined (DDSP_DEBUG)
    Char8                mqtName [DSP_MAX_STRLEN] ;
#endif /* if defined (DDSP_DEBUG) */
    MqtInterface *       mqtInterface ;
    Pvoid                mqtInfo ;
    Uint16               mqtId ;
    ProcessorId          procId ;
} LdrvMsgqTransportObj, *LdrvMsgqTransportHandle ;
```

Fields

`mqtName`	Name of the MQT. Used for debugging purposes only.
`mqtInterface`	Pointer to the function table of the MQT represented by the transport object.
`mqtInfo`	State information needed by the transport. The contents of this are transport-specific.
`mqtId`	ID of the MQT represented by the transport object.
`procId`	Processor Id associated with this MQT.

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 ()`

8.3.2.24 LdrvMsgqAllocatorObj

This structure defines the allocator object. There is one instance of the allocator object per MQA in the system.

Definition

```
typedef struct LdrvMsgqAllocatorObj_tag {
#if defined (DDSP_DEBUG)
    Char8                   mqaName [DSP_MAX_STRLEN] ;
#endif /* if defined (DDSP_DEBUG) */
    MqaInterface *          mqaInterface ;
    Pvoid                   mqaInfo ;
    Uint16                  mqaId ;
} LdrvMsgqAllocatorObj, *LdrvMsgqAllocatorHandle ;
```

Fields

`mqaName`	Name of the MQA. Used for debugging purposes only.
`mqaInterface`	Pointer to the function table of the MQA represented by the allocator object.
`mqaInfo`	State information needed by the allocator. The contents of this are allocator-specific.
`mqaId`	ID of the MQA represented by the allocator object.

Comments

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

The allocator objects are initialized during `LDRV_MSGQ_Setup ()` through configuration information obtained from the CFG. The MQA state information is filled in during `LDRV_MSGQ_AllocatorOpen ()`.

Constraints

None.

See Also

`LDRV_MSGQ_Setup ()`
`LDRV_MSGQ_AllocatorOpen ()`

8.3.2.25 MqaInterface

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

Definition

```
typedef struct MqaInterface_tag {
    FnMqaInit      mqaInit ;
    FnMqaExit      mqaExit ;
    FnMqaOpen      mqaOpen ;
    FnMqaClose     mqaClose ;
    FnMqaAlloc     mqaAlloc ;
    FnMqaFree      mqaFree ;
} MqaInterface;
```

Fields

`*mqaInit`	Pointer to MQA initialization function.
`*mqaExit`	Pointer to MQA finalization function.
`*mqaOpen`	Pointer to MQA open function.
`*mqaClose`	Pointer to MQA close function.
`*mqaAlloc`	Pointer to MQA function for allocating a message.
`*mqaFree`	Pointer to MQA function for freeing a message.

Comments

Each MQA 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 `MqaInterface`.

Constraints

None

See Also

`MqtInterface`

8.3.2.26 MqtInterface

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

Definition

```
typedef struct MqtInterface_tag {
    FnMqtInit         mqtInit ;
    FnMqtExit         mqtExit ;
    FnMqtOpen         mqtOpen ;
    FnMqtClose        mqtClose ;
    FnMqtCreate       mqtCreate ;
    FnMqtLocate       mqtLocate ;
    FnMqtDelete       mqtDelete ;
    FnMqtRelease      mqtRelease ;
    FnMqtGet          mqtGet ;
    FnMqtPut          mqtPut ;
    FnMqtGetReplyId   mqtGetReplyId ;
    FnMqtGetById      mqtGetById ;
#if defined (DDSP_PROFILE)
    FnMqtInstrument   mqtInstrument ;
#endif /* defined (DDSP_PROFILE) */
#if defined (DDSP_DEBUG)
    FnMqtDebug        mqtDebug ;
#endif /* defined (DDSP_DEBUG) */
} MqtInterface ;
```

Fields

*mqtInit	Pointer to MQT initialization function.
*mqtExit	Pointer to MQT finalization function.
*mqtOpen	Pointer to MQT open function.
*mqtClose	Pointer to MQT close function.
*mqtCreate	Pointer to MQT function for creating a MSGQ.
*mqtLocate	Pointer to MQT function for locating a MSGQ.
*mqtDelete	Pointer to MQT function for deleting a MSGQ.
*mqtRelease	Pointer to MQT function for releasing a MSGQ.
*mqtGet	Pointer to MQT function for receiving a message.
*mqtPut	Pointer to MQT function for sending a message.
*mqtGetReplyId	Pointer to MQT function for getting the reply MSGQ ID for a particular message.
*mqtGetById	Pointer to MQT function for receiving a message having a particular MSG ID.
*mqtInstrument	Pointer to MQT Instrumentation function.

*mqtDebug	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 MqtInterface.

Constraints

Remote MQTs are not required to implement mqtCreate (), mqtDelete (), mqtGet () and mqtGetById () functions.

The mqtGetById () function shall not be initially supported for the local MQT also.

See Also

MqaInterface

8.3.3 API Definition

8.3.3.1 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.

Constraints

None.

See Also

LDRV_MSGQ_Destroy ()

8.3.3.2 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 interacts with the MQAs and the MQTs for cleanup, in case they have not been finalized before.

Constraints

None.

See Also

LDRV_MSGQ_Setup ()

8.3.3.3 LDRV_MSGQ_AllocatorOpen

This function opens the MQA component.

Syntax

```
DSP_STATUS LDRV_MSGQ_AllocatorOpen (AllocatorId mqaId,
                                    Pvoid       mqaAttrs,
                                    Pvoid *     mqaInfo);
```

Arguments

`IN` `AllocatorId` `mqaId`

ID of the MQA to be opened.

`IN` `Pvoid` `mqaAttrs`

Attributes for initialization of the MQA component. The structure of the expected attributes is specific to an MQA.

`OUT` `Pvoid *` `mqaInfo`

Location to receive the handle of the initialized MQA state object.

Return Value

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

Comments

This function calls the `mqaOpen ()` function of the MQA identified through the MQA ID. It initializes the MQA using the provided attributes.

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

Constraints

None.

See Also

None

8.3.3.4 LDRV_MSGQ_AllocatorClose

This function closes the MQA component.

Syntax

```
DSP_STATUS LDRV_MSGQ_AllocatorClose (AllocatorId mqaId) ;
```

Arguments

IN AllocatorId mqaId

ID of the MQA to be closed.

Return Value

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

Comments

This function calls the mqaClose () function of the MQA identified through the MQA ID.

Constraints

None.

See Also

LDRV_MSGQ_AllocatorOpen ()

8.3.3.5 LDRV_MSGQ_TransportOpen

This function opens the MQT component.

Syntax

```
DSP_STATUS LDRV_MSGQ_TransportOpen (TransportId mqtId,
                                    Pvoid       mqtAttrs) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT to be opened.

IN	Pvoid	mqtAttrs

Attributes for initialization of the MQT component. The requirement and structure of the expected attributes is specific to an MQT.

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 mqtOpen () function of the MQT identified through the MQT 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

None.

See Also

None

8.3.3.6 LDRV_MSGQ_TransportClose

This function closes the MQT component.

Syntax

```
DSP_STATUS LDRV_MSGQ_TransportClose (TransportId mqtId) ;
```

Arguments

IN	TransportId	mqtId

ID of the MQT 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 `mqtClose ()` function of the MQT identified through the MQT ID.

Constraints

None.

See Also

`LDRV_MSGQ_TransportOpen ()`

8.3.3.7 LDRV_MSGQ_Create

This function creates the message queue to be used for receiving messages, identified through the specified MSGQ ID.

Syntax

```
DSP_STATUS LDRV_MSGQ_Create (MsgQueueId  msgqId,
                             MsgqAttrs * msgqAttrs) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the message queue to be created.

IN OPT	MsgqAttrs *	msgqAttrs

Optional attributes for creation of the MSGQ.

Return Value

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

Comments

This function interacts with the local MQT to create a MSGQ as specified by the user.

Constraints

None.

See Also

LMQT_Create ()

8.3.3.8 LDRV_MSGQ_Delete

This function deletes the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS LDRV_MSGQ_Delete (MsgQueueId msgqId) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of 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 interacts with the local MQT to delete the MSGQ as specified by the user.

Constraints

None.

See Also

LMQT_Delete ()

8.3.3.9 LDRV_MSGQ_Locate

This function verifies the existence and status of the message queue identified by the specified MSGQ ID, on the specified processor.

Syntax

```
DSP_STATUS LDRV_MSGQ_Locate (ProcessorId       procId,
                             MsgQueueId        msgqId,
                             MsgqLocateAttrs * attrs) ;
```

Arguments

IN	ProcessorId	procId

ID of the processor on which the MSGQ is to be located.

IN	MsgQueueId	msgqId

ID of the message queue to be located.

IN	MsgqLocateAttrs *	attrs

Attributes for location of the MSGQ.

Return Value

DSP_SOK	The message queue has been successfully located.
DSP_ENOTFOUND	The message queue does not exist on the specified processor.
DSP_ETIMEOUT	Timeout occurred while locating the MSGQ.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function interacts with the local and remote MQTs to locate the MSGQ as specified by the user. If the procId specified is the ID_LOCAL_PROCESSOR, the corresponding locate call is made to the local MQT. For all other processors, the call passes down to the corresponding remote MQT for the specified processor.

Constraints

None.

See Also

ID_LOCAL_PROCESSOR
LMQT_Locate ()
RMQT_Locate ()

8.3.3.10 LDRV_MSGQ_Release

This function releases the MSGQ located earlier.

Syntax

```
DSP_STATUS LDRV_MSGQ_Release (ProcessorId procId,
                              MsgQueueId  msgqId) ;
```

Arguments

IN	ProcessorId	procId

ID of the processor on which the MSGQ is to be released.

IN	MsgQueueId	msgqId

ID of 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_EFAIL	General failure.

Comments

This function interacts with the local and remote MQTs to release the MSGQ as specified by the user. If the procId specified is the ID_LOCAL_PROCESSOR, the corresponding locate call is made to the local MQT. For all other processors, the call passes down to the corresponding remote MQT for the specified processor.

Constraints

None.

See Also

ID_LOCAL_PROCESSOR
LMQT_Release ()
RMQT_Release ()

8.3.3.11 LDRV_MSGQ_Alloc

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

Syntax

```
DSP_STATUS LDRV_MSGQ_Alloc (AllocatorId mqaId,
                            Uint16      size,
                            MsgqMsg *   msg) ;
```

Arguments

IN	AllocatorId	mqaId

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

IN	Uint16	size

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

OUT	MsgqMsg *	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 MQA to allocate a message of specified size.

Constraints

None.

See Also

MQABUF_Alloc ()

8.3.3.12 LDRV_MSGQ_Free

This function frees a message.

Syntax

```
DSP_STATUS LDRV_MSGQ_Free (MsgqMsg msg) ;
```

Arguments

IN	MsgqMsg	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

None.

See Also

MQABUF_Free ()

8.3.3.13 LDRV_MSGQ_Put

This function sends a message to the specified MSGQ on the specified processor.

Syntax

```
DSP_STATUS LDRV_MSGQ_Put (ProcessorId procId,
                          MsgQueueId  destMsgqId,
                          MsgqMsg     msg,
                          Uint16      msgId,
                          MsgQueueId  srcMsgqId) ;
```

Arguments

IN	ProcessorId	procId

ID of the processor on which the MSGQ to which the message is to be sent, exists.

IN	MsgQueueId	destMsgqId

ID of the destination MSGQ.

IN	MsgqMsg	msg

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

IN OPT	Uint16	msgId

Optional message ID to be associated with the message.

IN OPT	MsgQueueId	srcMsgqId

Optional ID of the source MSGQ to receive reply messages.

Return Value

DSP_SOK	The message has been successfully sent.
DSP_ENOTFOUND	The message queue does not exist. This implies that the MSGQ has not been located before this call was made.
DSP_EFAIL	General failure.

Comments

This function interacts with the local or remote MQT to send the message to the destination MSGQ. If the procId specified is the ID_LOCAL_PROCESSOR, the corresponding put call is made to the local MQT. For all other processors, the call passes down to the corresponding remote MQT for the specified processor.

Constraints

None.

See Also

ID_LOCAL_PROCESSOR

```
LMQT_Put ()
RMQT_Put ()
```

8.3.3.14 LDRV_MSGQ_Get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS LDRV_MSGQ_Get (MsgQueueId msgqId,
                          Uint32     timeout,
                          MsgqMsg *  msg) ;
```

Arguments

IN	MsgQueueId	msgqId

ID of the MSGQ on which the message is to be received.

IN	timeout	timeout

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

OUT	MsgqMsg *	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_EFAIL	General failure.

Comments

This function interacts with the local MQT to receive a message on the specified MSGQ.

Constraints

None.

See Also

LMQT_Get ()

8.3.3.15 LDRV_MSGQ_GetReplyId

This function extracts the MSGQ ID and processor ID to be used for replying to a received message.

Syntax

```
DSP_STATUS LDRV_MSGQ_GetReplyId (MsgqMsg       msg,
                                 ProcessorId * procId,
                                 MsgQueueId *  msgqId) ;
```

Arguments

IN	MsgqMsg	msg

Message, whose reply MSGQ ID is to be extracted.

OUT	ProcessorId *	procId

Location to retrieve the ID of the processor where the reply MSGQ resides.

OUT	MsgQueueId *	msgqId

Location to retrieve the ID of the reply MSGQ.

Return Value

DSP_SOK	The reply information has been successfully retrieved.
DSP_ENOTFOUND	Reply information has not been provided by the sender.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function extracts the MSGQ ID and processor ID to be used for replying to a message. The processor ID is extracted from the map of the processor IDs to the MQT IDs maintained within the MSGQ state information.

It also interacts with the MQT to get the reply MSGQ ID for the specified message.

Constraints

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

See Also

None.

8.3.3.16 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 (MsgQueueId msgqId,
                                      Uint16     mqaId) ;
```

Arguments

IN MsgQueueId msgqId

Message queue to receive the error messages.

IN Uint16 mqaId

ID indicating the allocator 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.

See Also

MsgqAsyncErrorMsg
LDRV_MSGQ_SendErrorMsg ()

8.3.3.17 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 (MsgqErrorType errorType,
                                   Pvoid         arg1,
                                   Pvoid         arg2,
                                   Pvoid         arg3) ;
```

Arguments

IN	MsgqErrorType	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.

IN	Pvoid	arg3

Third 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.

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

MsgqErrorType
MsgqAsyncErrorMsg
MSGQ_SetErrorHandler ()
LDRV_MSGQ_SetErrorHandler ()

8.3.3.18 LDRV_MSGQ_Instrument

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

Syntax

```
DSP_STATUS LDRV_MSGQ_Instrument (ProcessorId      procId,
                                 MsgQueueId       msgqId,
                                 MsgqInstrument * retVal) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

OUT	MsgqInstrument *	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.

See Also

MsgqInstrument

8.3.3.19 LDRV_MSGQ_Debug

This function prints the status of the MSGQ subcomponent.

Syntax

```
Void LDRV_MSGQ_Debug (ProcessorId procId, MsgQueueId msgqId) ;
```

Arguments

IN	ProcessorId	procId

Processor identifier.

IN	MsgQueueId	msgqId

Message queue identifier.

Return Value

None.

Comments

None.

Constraints

This function is defined only for debug builds.

See Also

None.

8.3.4 Other Updates

1. LdrvMsgqStateObj

The LDRV MSGQ maintains its state within a global state variable.

```
static LdrvMsgqState LdrvMsgqStateObj ;
```

8.4 BUF

BUF is a generic component for creation and management of fixed buffer pools. Buffer pools of various sizes can be configured, and buffers of requested sizes allocated and freed.

This component is used by the buffer allocator required for messaging.

8.4.1 Typedefs & Data Structures

8.4.1.1 BufObj

This structure defines the buffer pool object. It maintains the pool of buffers of a particular fixed size.

Definition

```
typedef struct BufObj_tag {
    Uint32        startAddress ;
    Uint16        size ;
    Uint32        nextFree ;
    Uint16        totalBuffers ;
    Uint16        freeBuffers ;
    Bool          freePool ;
} BufObj, *BufHandle ;
```

Fields

`startAddress`	Starting address of buffer pool.
`size`	Size of the buffers in this pool.
`nextFree`	Pointer to next free buffer.
`totalBuffers`	Total number of buffers in pool.
`freeBuffers`	Number of free buffers in pool.
`freePool`	Indicates whether the buffer pool was allocated within the BUF component, and should be freed during `BUF_Delete ()`

Comments

None.

Constraints

None.

See Also

`BUF_Create ()`

8.4.1.2 BufHeader

This structure defines the buffer header. This structure maintains information required to link the buffers within each buffer pool.

Definition

```
typedef struct BufHeader_tag {
    struct BufHeader_tag * next ;
#if defined (DDSP_PROFILE)
    Uint16                 usedOnce ;
#endif /* if defined (DDSP_PROFILE) */
} BufHeader ;
```

Fields

`next`	Pointer to next buffer header.
`usedOnce`	Indicates if the buffer has been used at least once. When profiling is enabled, this field is used for getting statistics regarding the usage of the buffers within the buffer pools.

Comments

Each free buffer within the buffer pool maintains the linking information through this structure. No extra space is required within the buffer for maintaining this information, since it is only required for the free buffers, and can be overwritten by the users after the buffer has been allocated.

Constraints

The size of the buffer pools configured during `BUF_Create ()` must be greater than or equal to the size of the buffer header.

See Also

`BUF_Alloc ()`
`BUF_Free ()`
`BUF_Create ()`

8.4.1.3 BufStats

This structure defines instrumentation data for the buffer pools.

Definition

```
#if defined (DDSP_PROFILE)
typedef struct BufStats_tag {
    Uint16        size ;
    Uint16        totalBuffers ;
    Uint16        freeBuffers ;
    Uint16        maxUsed ;
} BufStats ;
#endif /* if defined (DDSP_PROFILE) */
```

Fields

`size`	Size of the buffers in this pool.
`totalBuffers`	Total number of buffers in pool.
`freeBuffers`	Number of free buffers in pool.
`maxUsed`	Maximum number of buffers that have been used at least once since creation of the buffer pool.

Comments

This structure holds the instrumentation data to be returned for a particular buffer pool. The information about maximum number of buffers that have been used at least once is useful to find out the optimal use of the buffer pool.

Constraints

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

See Also

`BUF_GetStats ()`

8.4.2 API Definition

8.4.2.1 BUF_Initialize

This function initializes the buffer component.

Syntax

```
DSP_STATUS BUF_Initialize () ;
```

Arguments

None.

Return Value

DSP_SOK	This component has been successfully initialized.
DSP_EFAIL	General failure.

Comments

This function performs all global initialization of this component.

Constraints

None.

See Also

BUF_Finalize ()

8.4.2.2 BUF_Finalize

This function finalizes the buffer component.

Syntax

```
DSP_STATUS BUF_Finalize () ;
```

Arguments

None.

Return Value

DSP_SOK	This component has been successfully finalized.
DSP_EFAIL	General failure.

Comments

This function performs all global finalization of this component.

Constraints

None.

See Also

BUF_Initialize ()

8.4.2.3 BUF_Create

This function creates and initializes a fixed buffer pool and returns a handle to the corresponding buffer pool object.

Syntax

```
DSP_STATUS BUF_Create (Uint16      numBufs,
                       Uint16      size,
                       BufHandle * bufHandle,
                       Uint32      bufAddress) ;
```

Arguments

`IN` `Uint16` `numBufs`

Number of buffers to be created in the pool.

`IN` `Uint16` `size`

Size of the buffers within the pool.

`OUT` `BufHandle *` `bufHandle`

Location to receive the handle to the created buffer pool object.

`IN OPT` `Uint32` `bufAddress`

Address of the memory reserved for this buffer pool. If a valid address is specified, no memory is allocated within this function, and it can be called from within DPC context. The size of the memory for the buffer pool allocated by the user must be equal to (`size * numBufs`).

If the address specified is NULL, this function internally allocates the memory required for the buffer pool. In this case, this function cannot be called from DPC context.

Return Value

`DSP_SOK`	The buffer pool has been successfully created.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	General failure.

Comments

This function initializes the buffer object and makes the buffer pool ready for access.

Constraints

The size of the buffer pools must be greater than or equal to the size of the buffer header.

See Also

`BufHeader`

```
BufObj
BUF_Delete ()
```

8.4.2.4 BUF_Delete

This function deletes the buffer pool specified by the user.

Syntax

```
DSP_STATUS BUF_Delete (BufHandle bufHandle) ;
```

Arguments

IN	BufHandle	bufHandle

Handle to the buffer pool object.

Return Value

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

Comments

This function deletes the buffer pool object. If the memory for the buffer pool was allocated within the BUF component, it is freed during this function. After this function has been called, no further calls can be made to allocate or free memory from this buffer pool.

Constraints

None.

See Also

BufObj
BUF_Create ()

8.4.2.5 BUF_Alloc

This function allocates a free buffer from the specified buffer pool and returns it to the user.

Syntax

```
DSP_STATUS BUF_Alloc (BufHandle bufHandle, Pvoid * buffer) ;
```

Arguments

IN	BufHandle	bufHandle

Handle to the buffer pool object.

OUT	Pvoid *	buffer

Location to receive the allocated buffer.

Return Value

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

Comments

The size of the buffer allocated through this function is always fixed for a specific buffer pool, hence no size is specified for allocation. The allocation is deterministic.

Constraints

None.

See Also

BufObj
BUF_Free ()

8.4.2.6 BUF_Free

This function frees the buffer specified by the user, and returns it to the buffer pool.

Syntax

```
DSP_STATUS BUF_Free (BufHandle bufHandle, Pvoid buffer) ;
```

Arguments

IN	BufHandle	bufHandle

Handle to the buffer pool object.

IN	Pvoid	buffer

Pointer to the buffer to be freed.

Return Value

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

Comments

The size of the buffer allocated through this function is always fixed for a specific buffer pool, hence no size is specified for freeing the buffer.

Constraints

None.

See Also

BufObj
BUF_Alloc ()

8.4.2.7 BUF_GetStats

This function gets instrumentation information about the specified buffer pool.

Syntax

```
DSP_STATUS BUF_GetStats (BufHandle bufHandle, BufStats * bufStats) ;
```

Arguments

IN	BufHandle	bufHandle

Handle to the buffer pool object.

OUT	BufStats *	bufStats

Location to receive the instrumentation information.

Return Value

DSP_SOK	The buffer has been successfully freed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This function returns information useful for profiling of the buffer pool usage. This also includes information about the maximum number of buffers that have been used at least once, to indicate the level of usage of the buffers in the pool. This information is achieved by traversing the buffer pool until a buffer is found which has never been used so far.

Constraints

This function is only defined when profiling is enabled within DSPLINK.

See Also

BufStats

8.5 LDRV - MQA

8.5.1 Typedefs & Data Structures

8.5.1.1 MqaBufObj

This structure defines the buffer object for the buffer allocator.

Definition

```
typedef struct MqaBufObj_tag {
    Uint16       msgSize ;
    BufHandle    msgList ;
} MqaBufObj ;
```

Fields

`msgSize`	Size of the messages in the buffer pool.
`msgList`	List of messages in the buffer pool.

Comments

The allocator contains multiple buffer pools for various message sizes. This structure defines the buffer object for a particular message size.

Constraints

None.

See Also

`BufObj`
`MqaBufState`
`MQABUF_Alloc ()`
`MQABUF_Free ()`

8.5.1.2 MqaBufState

This structure defines the allocator state object, which exists as a single instance in the system, and represents the complete allocator.

Definition

```
typedef struct MqaBufState_tag {
    Uint16        numBufPools ;
    MqaBufObj *   bufPools ;
    Uint32        phyAddr ;
    Uint32        virtAddr ;
    Uint32        size ;
} MqaBufState ;
```

Fields

`numBufPools`	Number of buffer pools configured in the MQA.
`bufPools`	Array of buffer pools for various message sizes. The array is dynamically allocated of size equal to the one specified by the user.
`phyAddr`	Physical address of the contiguous memory for the buffer pools.
`virtAddr`	Virtual address of the contiguous memory for the buffer pools.
`size`	Size of the contiguous memory for the buffer pool.

Comments

An instance of this object is created and initialized during `MQABUF_Open ()`, and the handle is returned to the caller. It contains all information required for maintaining the state of the MQA.

Constraints

None.

See Also

`MqaBufObj`
`MQABUF_Open ()`

8.5.2 API Definition

The MQA APIs are exposed to MSGQ through a function table:

```
MqaInterface MQABUF_Interface = {
    &MQABUF_Init,
    &MQABUF_Exit,
    &MQABUF_Open,
    &MQABUF_Close,
    &MQABUF_Alloc,
    &MQABUF_Free
} ;
```

8.5.2.1 MQABUF_Init

This function performs global initialization of the buffer MQA.

Syntax

```
Void MQABUF_Init () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during the initialization of the MSGQ component, to perform any global initialization required for the MQA.

Constraints

None.

See Also

```
LDRV_MSGQ_Setup ()
MQABUF_Exit ()
```

8.5.2.2 MQABUF_Exit

This function performs global finalization of the buffer MQA.

Syntax

```
Void MQABUF_Exit () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during finalization of the MSGQ component, to perform any global finalization required for the MQA.

Constraints

None.

See Also

```
LDRV_MSGQ_Destroy ()
MQABUF_Init ()
```

8.5.2.3 MQABUF_Open

This function opens the buffer MQA and configures it according to the user attributes.

Syntax

```
DSP_STATUS MQABUF_Open (LdrvMsgqAllocatorHandle mqaHandle,
                        Pvoid                   mqaAttrs) ;
```

Arguments

IN LdrvMsgqAllocatorHandle mqaHandle

Handle to the MSGQ allocator object.

IN Pvoid mqaAttrs

Attributes for initialization of the MQA component.

Return Value

DSP_SOK	This component has been successfully opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

This function is called once during LDRV_MSGQ_AllocatorOpen ().

It creates an instance of the MqaBufState object, initializes it, and sets the handle in the MSGQ allocator object. This handle to the MQA state object is passed to the allocator during its functions for allocating and freeing the messages, as part of the MSGQ allocator object.

The steps involved in initialization of the MQA are:

1. Calculate the total size of buffer required for the MQA. This is equivalent to:

   The sum of (For each pool: (numMsgs * msgSize))

2. Allocate a contiguous buffer of calculated size.
3. Create and initialize each buffer pool through calls to BUF_Create (), passing the kernel address of the contiguous buffer reserved for each pool through step (2).

Constraints

None.

See Also

MqaBufAttrs
MqaBufState
MQABUF_Close ()
BUF_Create ()

8.5.2.4 MQABUF_Close

This function closes the MQA and cleans up its state object.

Syntax

```
DSP_STATUS MQABUF_Close (LdrvMsgqAllocatorHandle mqaHandle) ;
```

Arguments

`IN` `LdrvMsgqAllocatorHandle` `mqaHandle`

Handle to the MSGQ allocator object.

Return Value

`DSP_SOK`	This component has been successfully closed.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	General failure.

Comments

This function is called once during `LDRV_MSGQ_AllocatorClose ()`.

It cleans up the instance of the `MqaBufState` object, in addition to any other required actions for finalizing the MQA.

After successful completion of this function, no further calls can be made to the MQA for allocation or freeing of messages.

Constraints

None.

See Also

```
MqaBufState
MQABUF_Open ()
BUF_Delete ()
```

8.5.2.5 MQABUF_Alloc

This function allocates a message buffer of the specified size.

Syntax

```
DSP_STATUS MQABUF_Alloc (LdrvMsgqAllocatorHandle mqaHandle,
                         Uint16 *                size,
                         MsgqMsg *               addr) ;
```

Arguments

`IN` `LdrvMsgqAllocatorHandle` `mqaHandle`

Handle to the MSGQ allocator object.

`IN OUT` `Uint16 *` `size`

Size of the message to be allocated. On return, it stores the actual allocated size of the message, which, for the buffer MQA is the same as the requested size on success, or zero on failure.

`OUT` `MsgqMsg *` `addr`

Location to receive the allocated message.

Return Value

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

Comments

This function attempts to allocate a buffer of the size required by the caller. If a buffer of the requested size is not found, or the buffer list corresponding to the specified size is empty, an error is returned. In addition, the size allocated is returned as zero, and the returned buffer pointer is NULL.

This function allocates the specified message by removing it from the appropriate buffer pool. The buffer pool is identified based on the size of the message to be allocated.

Constraints

None.

See Also

`MqaBufState`
`MQABUF_Free ()`
`BUF_Alloc ()`

8.5.2.6 MQABUF_Free

This function frees a message buffer of the specified size.

Syntax

```
DSP_STATUS MQABUF_Free (LdrvMsgqAllocatorHandle mqaHandle,
                        MsgqMsg                 addr,
                        Uint16                  size) ;
```

Arguments

`IN` `LdrvMsgqAllocatorHandle` `mqaHandle`

Handle to the MSGQ allocator object.

`IN` `MsgqMsg` `addr`

Address of the message to be freed.

`IN` `Uint16` `size`

Size of 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_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	General failure.

Comments

This function frees the specified message by adding it to the appropriate buffer pool. The buffer pool is identified based on the size of the message to be freed.

Constraints

None.

See Also

`MqaBufState`
`MQABUF_Alloc ()`
`BUF_Free ()`

8.6 LDRV - Local MQT

8.6.1 Typedefs & Data Structures

8.6.1.1 LmqtState

This structure defines the transport state object, which exists as a single instance for the local MQT.

Definition

```
typedef struct LmqtState_tag {
    Uint16            maxNumMsgq ;
    LdrvMsgqHandle *  msgqHandles ;
} LmqtState ;
```

Fields

`maxNumMsgq`	Maximum number of MSGQs that can be created on the local processor.
`msgqHandles`	Array of handles to the MSGQ objects for the local MSGQs.

Comments

An instance of this object is created and initialized during `LMQT_Open ()`, and its handle is returned to the caller. It contains all information required for maintaining the state of the MQT. This includes the maximum number of local MSGQs and the array of MSGQ handles for MSGQs created on the local processor.

Constraints

None.

See Also

`LMQT_Open ()`

8.6.1.2 LmqtObj

This structure defines the transport object, which has an instance for every MSGQ created on the processor.

Definition

```
typedef struct LmqtObj_tag {
    List *    msgQueue ;
} LmqtObj ;
```

Fields

`msgQueue`	Message repository to queue pending messages.

Comments

One instance of the MQT object is created for every message queue. The MQT object corresponding to a receiver queue is created during `LDRV_MSGQ_Create ()`. The MQT object for a remote queue (sender) is created during `LDRV_MSGQ_Locate ()`.

Constraints

None.

See Also

`LMQT_Create ()`
`LDRV_MSGQ_Create ()`
`LDRV_MSGQ_Locate ()`

8.6.2 API Definition

The MQT APIs are exposed to MSGQ through a function table:

```
MqtInterface LMQT_Interface = {
    &LMQT_Init,
    &LMQT_Exit,
    &LMQT_Open,
    &LMQT_Close,
    &LMQT_Create,
    &LMQT_Locate,
    &LMQT_Delete,
    &LMQT_Release,
    &LMQT_Get,
    &LMQT_Put,
    &LMQT_GetReplyId,
    NULL
} ;
```

8.6.2.1 LMQT_Init

This function performs global initialization of the local MQT.

Syntax

```
Void LMQT_Init () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during the initialization of the MSGQ component, to perform any global initialization required for the local MQT.

Constraints

None.

See Also

```
LDRV_MSGQ_Setup ()
LMQT_Exit ()
```

8.6.2.2 LMQT_Exit

This function performs global finalization of the local MQT.

Syntax

```
Void LMQT_Exit () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during finalization of the MSGQ component, to perform any global finalization required for the local MQT.

Constraints

None.

See Also

```
LDRV_MSGQ_Destroy ()
LMQT_Init ()
```

8.6.2.3 LMQT_Open

This function opens the local MQT and configures it according to the user attributes.

Syntax

```
DSP_STATUS LMQT_Open (LdrvMsgqTransportHandle mqtHandle,
                      Pvoid                   mqtAttrs) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the transport object.

IN	Pvoid	mqtAttrs

Attributes required for initialization of the MQT component.

Return Value

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

Comments

This function is called during MSGQ_TransportOpen (), when the MQT ID passed is the one indicating the local MQT. It carries out all initialization required for the MQT. This function is called only once for the MQT before any of its other functions can be called.

It creates and initializes an instance of the state object LmqtState, and returns it to the LDRV MSGQ component.

This function expects certain attributes from the user, which are defined by the LmqtAttrs structure.

Constraints

None.

See Also

LmqtState
LmqtAttrs
LMQT_Close ()

8.6.2.4 LMQT_Close

This function closes the local MQT, and cleans up its state object.

Syntax

```
DSP_STATUS LMQT_Close (LdrvMsgqTransportHandle mqtHandle) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle mqtHandle`

Handle to the MSGQ transport object.

Return Value

`DSP_SOK`	This component has been successfully closed.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	General failure.

Comments

This function is called during `MSGQ_TransportClose ()`, when the MQT ID passed is the one indicating the local MQT. After successful completion of this function, no further MQT services shall be available from the local MQT.

Constraints

None.

See Also

`LmqtState`
`LMQT_Open ()`

8.6.2.5 LMQT_Create

This function creates the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS LMQT_Create (LdrvMsgqTransportHandle mqtHandle,
                        MsgQueueId              msgqId,
                        LdrvMsgqHandle *        msgqHandle,
                        MsgqAttrs *             attrs) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MsgQueueId	msgqId

ID of the message queue to be created.

OUT	LdrvMsgqHandle *	msgqHandle

Location to store the handle to the created MSGQ.

IN OPT	MsgqAttrs *	attrs

Optional attributes for creation of the MSGQ.

Return Value

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

Comments

This function is called during LDRV_MSGQ_Create (). It creates an instance of the message queue object, initializes it, and returns the handle to the caller.

An instance of the MQT object is also created during this function. This MQT object stores all data that is specific to every MSGQ, including the message repository for receiving messages.

Constraints

None.

See Also

LmqtObj ()
LMQT_Delete ()

8.6.2.6 LMQT_Delete

This function deletes the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS LMQT_Delete (LdrvMsgqTransportHandle mqtHandle,
                        MsgQueueId              msgqId) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MsgQueueId	msgqId

ID of the message queue to be deleted.

Return Value

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

Comments

This function is called during LDRV_MSGQ_Delete (). It deletes the specified message queue object. After this API has been successfully called, no further calls can be made to this MSGQ, including receiving messages.

The instance of the MQT object created during LMQT_Create () is also deleted during this function.

Constraints

None.

See Also

LmqtObj ()
LMQT_Create ()

8.6.2.7 LMQT_Locate

This function verifies the existence and status of the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS LMQT_Locate (LdrvMsgqTransportHandle mqtHandle,
                        MsgQueueId              msgqId,
                        MsgqLocateAttrs *       attrs) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of the message queue to be located.

`IN` `MsgqLocateAttrs *` `attrs`

Attributes for location of the MSGQ.

Return Value

`DSP_SOK`	The message queue has been successfully located.
`DSP_ENOTFOUND`	The message queue does not exist among the MSQs managed by this MQT.
`DSP_EFAIL`	General failure

Comments

This function is called during `LDRV_MSGQ_Locate ()`.

If the `processorId` specified for location of the MSGQ during `LDRV_MSGQ_Locate ()` is the `ID_LOCAL_PROCESSOR`, the corresponding locate call passes down into the local MQT.

Constraints

None.

See Also

`ID_LOCAL_PROCESSOR`
`LMQT_Release ()`

8.6.2.8 LMQT_Release

This function releases the MSGQ located through an earlier `LMQT_Locate ()` call.

Syntax

```
DSP_STATUS LMQT_Release (LdrvMsgqTransportHandle mqtHandle,
                         MsgQueueId              msgqId) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of 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_EFAIL`	General failure

Comments

This function is called during `LDRV_MSGQ_Release ()`.

If the `processorId` specified for location of the MSGQ during `LDRV_MSGQ_Release ()` is the `ID_LOCAL_PROCESSOR`, the corresponding locate call passes down into the local MQT.

This function does not need to implement any specific functionality for releasing the MSGQ.

Constraints

None.

See Also

`ID_LOCAL_PROCESSOR`
`LMQT_Locate ()`

8.6.2.9 LMQT_Get

This function receives a message on the specified MSGQ.

Syntax

```
DSP_STATUS LMQT_Get (LdrvMsgqTransportHandle mqtHandle,
                     MsgQueueId              msgqId,
                     Uint32                  timeout,
                     MsgqMsg *               msg) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of the MSGQ on which the message is to be received.

`IN` `timeout` `timeout`

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

`OUT` `MsgqMsg *` `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_EFAIL`	General failure.

Comments

This function is called during `LDRV_MSGQ_Get ()`.

The function waits for arrival of a message on the specified MSGQ with the timeout as specified by the user. It then returns the message to the user.

Constraints

None.

See Also

`ID_LOCAL_PROCESSOR`

8.6.2.10 LMQT_Put

This function sends a message to the specified local MSGQ.

Syntax

```
DSP_STATUS LMQT_Put (LdrvMsgqTransportHandle mqtHandle,
                     MsgQueueId              msgqId,
                     MsgqMsg                 msg) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of the destination MSGQ.

`IN` `MsgqMsg` `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. This implies that the MSGQ has not been located before this call was made.
`DSP_EFAIL`	General failure.

Comments

This function is called during `LDRV_MSGQ_Put ()`.

If the `processorId` specified is the `ID_LOCAL_PROCESSOR`, the corresponding put call passes down into the local MQT. This function queues up the message onto the destination MSGQ and signals it about the arrival of the message.

Constraints

None.

See Also

`ID_LOCAL_PROCESSOR`

8.6.2.11 LMQT_GetReplyId

This function extracts the MSGQ ID to be used for replying to a received message.

Syntax

```
DSP_STATUS LMQT_GetReplyId (LdrvMsgqTransportHandle mqtHandle,
                            MsgqMsg                 msg,
                            MsgQueueId *            msgqId) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MsgqMsg	msg

Message, whose reply MSGQ ID is to be extracted.

OUT	MsgQueueId *	msgqId

Location to receive the ID of the reply MSGQ.

Return Value

DSP_SOK	The reply MSGQ ID has been successfully retrieved.
DSP_ENOTFOUND	Reply information has not been provided by the sender.
DSP_EFAIL	General failure.

Comments

This function extracts the MSGQ ID to be used for replying to a message.

Constraints

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

See Also

None.

8.7 LDRV - Remote MQT

8.7.1 Constants & Enumerations

8.7.1.1 ID_RMQT_CTRL

This macro defines the internal ID used to identify control messages.

Definition

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

Comments

This ID is used as the destination ID within the message header, when the message is intended for the MQT, and not a particular MSGQ.

Constraints

None.

See Also

None.

8.7.1.2 RmqtCtrlCmd

This enumeration defines the types of control commands that are sent between the MQTs on different processors.

Definition

```
typedef enum {
    RmqtCtrlCmd_Locate     = 0,
    RmqtCtrlCmd_LocateAck  = 1,
    RmqtCtrlCmd_Exit       = 2
} RmqtCtrlCmd ;
```

Comments

A control message has the destination ID as `ID_RMQT_CTRL`, indicating that the message is meant for the MQT, and not a particular MSGQ. In that case, the identification of the type of control message is made through the message ID field in the message header, which is one of the types defined by this enumeration. The actual message content differs depending on the control command.

Constraints

None.

See Also

`RmqtCtrlMsg`

8.7.2 Typedefs & Data Structures

8.7.2.1 RmqtState

This structure defines the transport state object, which exists as a single instance for the remote MQT.

Definition

```
typedef struct RmqtState_tag {
    LdrvMsgqTransportHandle mqtHandle ;
    Uint16                  maxNumMsgq ;
    Uint16                  maxMsgSize ;
    LdrvMsgqHandle *        msgqHandles ;
    MsgqMsg                 getBuffer ;
    List *                  msgQueue ;
    Uint16                  defaultMqaId ;
    ProcessorId             procId ;
} RmqtState ;
```

Fields

`mqtHandle`	Handle to the LDRV MSGQ Transport object.
`maxNumMsgq`	Maximum number of MSGQs that can be created on the remote processor.
`maxMsgSize`	Maximum message size (in bytes) supported by the MQT.
`msgqHandles`	Array of handles to the MSGQ objects for the remote MSGQs.
`getBuffer`	The buffer to be used for priming the input channel.
`msgQueue`	Message repository to queue pending messages.
`defaultMqaId`	The default MQA to be used by the remote MQT.
`procId`	Processor Id associated with this MQT.

Comments

An instance of this object is created and initialized during `RMQT_Open ()`, and its handle is returned to the caller. It contains all information required for maintaining the state of the MQT. This includes the maximum number of MSGQs on the remote processor, and the array of MSGQ handles for MSGQs created on the remote processor.

Constraints

None.

See Also

`RMQT_Open ()`

8.7.2.2 RmqtObj

This structure defines the transport object, which has an instance for every MSGQ created on the processor.

Definition

```
typedef struct RmqtObj_tag {
    SyncSemObject * locateSem;
} RmqtObj ;
```

Fields

`locateSem`	Semaphore used during `mqtLocate ()`.

Comments

One instance of the MQT object is created for every message queue. The MQT object corresponding to a receiver queue is created during `LDRV_MSGQ_Create ()`. The MQT object for a remote queue (sender) is created during `LDRV_MSGQ_Locate ()`.

The semaphore `locateSem` is a counting semaphore.

Constraints

None.

See Also

`RMQT_Locate ()`
`LDRV_MSGQ_Create ()`
`LDRV_MSGQ_Locate ()`

8.7.2.3 RmqtCtrlMsg

This structure defines the format of the control messages that are sent between the MQTs on different processors.

Definition

```
typedef struct RmqtCtrlMsg_tag {
    MsgqMsgHeader msgHeader ;
    union {
        struct {
            Uint16      msgqId ;
            Uint16      mqaId;
            Uint32      timeout;
            Uint32      replyHandle;
            Uint32      arg;
            Uint32      semHandle ;
        } locateMsg ;

        struct {
            Uint16      msgqId ;
            Uint16      msgqFound ;
            Uint16      mqaId;
            Uint32      timeout;
            Uint32      replyHandle;
            Uint32      arg;
            Uint32      semHandle ;
        } locateAckMsg ;
    } ctrlMsg ;
} RmqtCtrlMsg ;
```

Fields

`msgHeader`	Fixed message header required for all messages.

`ctrlMsg` Defines the format of the different control messages.

```
locateMsg:
    msgqId      -> ID of the MSGQ to be located on
                   the remote processor.
    mqaId       -> MQA ID to allocate async response
                   messages
    timeout     -> Timeout value for sync locate
    replyHandle -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    semHandle   -> Semaphore handle for sync locate
locateAckMsg:
    msgqId      -> ID of the MSGQ located on the
                   remote processor.
    msgqFound   -> Requested MSGQ was found on the
                   remote processor?
                   0 -> Not found,
                   1 -> Found
    mqaId       -> MQA ID to allocate async response
                   messages
    timeout     -> Timeout value for sync locate
    replyHandle -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    semHandle   -> Semaphore handle for sync locate
```

No control message is required when the command indicates exit of the remote MQT.

Comments

The control messages are used for communication between the MQTs. They contain information for MSGQ location and exit notification.

Constraints

None.

See Also

`RmqtCtrlCmd`

8.7.3 API Definition

The MQT APIs are exposed to MSGQ through a function table:

```
MqtInterface RMQT_Interface = {
    &RMQT_Init,
    &RMQT_Exit,
    &RMQT_Open,
    &RMQT_Close,
    NULL,
    &RMQT_Locate,
    NULL,
    &RMQT_Release,
    NULL,
    &RMQT_Put,
    &RMQT_GetReplyId,
    NULL
};
```

8.7.3.1 RMQT_Init

This function performs global initialization of the remote MQT.

Syntax

```
Void RMQT_Init () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during the initialization of the MSGQ component, to perform any global initialization required for the remote MQT.

Constraints

None.

See Also

```
LDRV_MSGQ_Setup ()
RMQT_Exit ()
```

8.7.3.2 RMQT_Exit

This function performs global finalization of the remote MQT.

Syntax

```
Void RMQT_Exit () ;
```

Arguments

None.

Return Value

None.

Comments

This function is called during finalization of the MSGQ component, to perform any global finalization required for the remote MQT.

Constraints

None.

See Also

```
LDRV_MSGQ_Destroy ()
RMQT_Init ()
```

8.7.3.3 RMQT_Open

This function opens the remote MQT and configures it according to the user attributes.

Syntax

```
DSP_STATUS RMQT_Open (LdrvMsgqTransportHandle mqtHandle,
                      Pvoid                   mqtAttrs) ;
```

Arguments

IN LdrvMsgqTransportHandle mqtHandle

Handle to the MSGQ transport object.

IN Pvoid mqtAttrs

Attributes required for initialization of the MQT component.

Return Value

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

Comments

This function is called during MSGQ_TransportOpen (), when the MQT ID passed is the one indicating the remote MQT. It carries out all initialization required for the MQT. This function is called only once for the MQT before any of its other functions can be called.

It creates and initializes an instance of the state object RmqtState, and returns it to the LDRV MSGQ component.

This function expects certain attributes from the user, which are defined by the RmqtAttrs structure.

Constraints

None.

See Also

RmqtState
RmqtAttrs
RMQT_Close ()

8.7.3.4 RMQT_Close

This function closes the remote MQT, and cleans up its state object.

Syntax

```
DSP_STATUS RMQT_Close (LdrvMsgqTransportHandle mqtHandle) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle mqtHandle`

Handle to the MSGQ transport object.

Return Value

`DSP_SOK`	This component has been successfully closed.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EFAIL`	General failure.

Comments

This function is called during `MSGQ_TransportClose ()`, when the MQT ID passed is the one indicating the remote MQT. After successful completion of this function, no further MQT services shall be available from the remote MQT.

Constraints

None.

See Also

`RmqtState`
`RMQT_Open ()`

8.7.3.5 RMQT_Locate

This function verifies the existence and status of the message queue identified by the specified MSGQ ID.

Syntax

```
DSP_STATUS RMQT_Locate (LdrvMsgqTransportHandle mqtHandle,
                        MsgQueueId              msgqId,
                        MsgqLocateAttrs *       attrs) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of the message queue to be located.

`IN` `MsgqLocateAttrs *` `attrs`

Attributes for location of the MSGQ.

Return Value

`DSP_SOK`	The message queue has been successfully located.
`DSP_ENOTFOUND`	The message queue does not exist among the MSQs managed by this MQT.
`DSP_ETIMEOUT`	Timeout occurred while locating the MSGQ.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EFAIL`	General failure

Comments

This function is called during `LDRV_MSGQ_Locate ()`.

If the `processorId` specified for location of the MSGQ during `LDRV_MSGQ_Locate ()` is the one corresponding to this remote MQT, the corresponding locate call passes down into it.

Constraints

None.

See Also

`RMQT_Release ()`

8.7.3.6 RMQT_Release

This function releases the MSGQ located through an earlier RMQT_Locate () or RMQT_GetReplyId () call.

Syntax

```
DSP_STATUS RMQT_Release (LdrvMsgqTransportHandle mqtHandle,
                         MsgQueueId              msgqId) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MsgQueueId	msgqId

ID of 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 was not previously located.
DSP_EFAIL	General failure

Comments

This function is called during LDRV_MSGQ_Release ().

If the processorId specified for releasing the MSGQ during LDRV_MSGQ_Release () is the one corresponding to this remote MQT, the corresponding release call passes down into it.

This function releases all resources allocated during the call to locate the remote MSGQ, including the remote MSGQ object maintained by the remote MQT.

Constraints

None.

See Also

RMQT_Locate ()

8.7.3.7 RMQT_Put

This function sends a message to the specified remote MSGQ.

Syntax

```
DSP_STATUS RMQT_Put (LdrvMsgqTransportHandle mqtHandle,
                     MsgQueueId              msgqId,
                     MsgqMsg                 msg) ;
```

Arguments

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgQueueId` `msgqId`

ID of the destination MSGQ.

`IN` `MsgqMsg` `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_ENOTFOUND`	The message queue does not exist. This implies that the MSGQ has not been located before this call was made.
`DSP_EFAIL`	General failure.

Comments

This function is called during `LDRV_MSGQ_Put ()`.

If the `processorId` specified is the one corresponding to this remote MQT, the corresponding put call passes down into it. This function queues up the message onto the destination MSGQ and initiates the transfer across the physical link. The completion of actual transfer over the physical link is indicated by the callback function being called.

Constraints

None.

See Also

`RMQT_PutCallback ()`

8.7.3.8 RMQT_GetReplyId

This function extracts the MSGQ ID to be used for replying to a received message.

Syntax

```
DSP_STATUS RMQT_GetReplyId (LdrvMsgqTransportHandle mqtHandle,
                            MsgqMsg                 msg,
                            MsgQueueId *            msgqId) ;
```

Arguments

IN	LdrvMsgqTransportHandle	mqtHandle

Handle to the MSGQ transport object.

IN	MsgqMsg	msg

Message, whose reply MSGQ ID is to be extracted.

OUT	MsgQueueId *	msgqId

Location to receive the ID of the reply MSGQ.

Return Value

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

Comments

This function extracts the MSGQ ID to be used for replying to a message.

It also checks whether the corresponding MSGQ has been previously located. If not, it creates and initializes the remote MSGQ object corresponding to the remote MSGQ.

Constraints

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

See Also

None.

8.7.3.9 RMQT_PutCallback

This function implements the callback that runs when the message to be sent to a remote MSGQ has been transferred across the physical link.

Syntax

```
DSP_STATUS RMQT_PutCallback (ProcessorId   procId,
                             Uint8  *      buffer,
                             Uint32        size,
                             Pvoid         arg) ;
```

Arguments

IN	ProcessorId	procId

Processor Identifier.

IN	Uint8 *	buffer

Pointer to the message buffer.

IN	Uint32	size

Size of the message buffer.

IN	Pvoid	arg

Argument associated with the IO request.

Return Value

DSP_SOK	The callback completed successfully
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

On receiving the callback after the message has been transferred over the physical link by the MQT, the message is freed, and the next pending message on the MSGQ (if any) is submitted to the driver.

Constraints

None.

See Also

RMQT_Put ()

8.7.3.10 RMQT_GetCallback

This function implements the callback that runs when the message has been received from the DSP.

Syntax

```
DSP_STATUS RMQT_GetCallback (ProcessorId   procId,
                             Uint8  *      buffer,
                             Uint32        size,
                             Pvoid         arg) ;
```

Arguments

IN ProcessorId procId

Processor Identifier.

IN Uint8 * buffer

Pointer to the message buffer.

IN Uint32 size

Size of the message buffer.

IN Pvoid arg

Argument associated with the IO request.

Return Value

DSP_SOK	The callback completed successfully
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

When the MQT is opened in RMQT_Open (), a buffer of the maximum size supported by the MQT is submitted on the messaging channel reserved for receiving messages from the DSP. Whenever a message is received from the DSP, it is copied into the buffer, and this callback function is called. On receiving the callback, a new message of the actual message size is allocated, the message copied into it, and the same big buffer reissued on the channel.

Constraints

None.

See Also

RMQT_Open ()

8.8 LDRV – physical link

The physical link driver shall not provide separate commands for messaging. Messaging shall be supported within the link through the same commands provided for data transfer. All physical links that are used by the remote MQT for communicating with other processors shall ensure the following:

1. Two channels shall be reserved for messaging, one for messages from GPP to DSP, and the other for messages from DSP to GPP
2. Messaging shall be prioritized over data transfer. This implies that the link shall always check the messaging channels for any pending transfers before the data channels.

8.9 Other updates

8.9.1 Constants & Enumerations

8.9.1.1 IS_VALID_MSGCHNLID

This macro checks for whether a channel ID provided is a valid messaging channel.

Definition

```
#define IS_VALID_MSGCHNLID(chnlId)                          \
                (    (chnlId == MAX_CHANNELS)               \
                 ||  (chnlId == (MAX_CHANNELS + 1)))
```

Comments

This macro is used within the LDRV CHNL component for validation of the channel IDs.

Constraints

None.

See Also

None.

8.9.2 Typedefs & Data Structures

8.9.2.1 CFG_Mqa

This structure defines the MQA configuration structure.

Definition

```
typedef struct CFG_Mqa_tag {
    Char8  mqaName    [CFG_MAX_STRLEN] ;
    Pvoid  interface                   ;
} CFG_Mqa ;
```

Fields

`mqaName`	Name of the MQA. For debugging purposes only.
`interface`	Function pointer interface to access the functions for this MQA.

Comments

These objects are generated from the CFG through cfg2c.pl.

Constraints

None.

See Also

None.

8.9.2.2 CFG_Mqt

This structure defines the MQT configuration structure.

Definition

```
typedef struct CFG_Mqt_tag {
    Char8  mqtName    [CFG_MAX_STRLEN] ;
    Pvoid  interface                   ;
    Uint32 linkId                      ;
} CFG_Mqt ;
```

Fields

`mqtName`	Name of the MQT. For debugging purposes only.
`interface`	Function pointer interface to access the functions for this MQT.
`linkId`	ID of the link used by this MQT.

Comments

These objects are generated from the CFG through cfg2c.pl.

Constraints

None.

See Also

None.

8.9.2.3 MqaObject

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

Definition

```
typedef struct MqaObject_tag {
    Char8  mqaName    [DSP_MAX_STRLEN] ;
    Pvoid  interface                   ;
} MqaObject ;
```

Fields

`mqaName`	Name of the MQA. For debugging purposes only.
`interface`	Function pointer interface to access the functions for this MQA.

Comments

An array of MQA objects is maintained within the LDRV object. These hold all MQA information obtained through the CFG.

Constraints

None.

See Also

None.

8.9.2.4 MqtObject

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

Definition

```
typedef struct MqtObject_tag {
    Char8  mqtName    [DSP_MAX_STRLEN] ;
    Pvoid  interface                   ;
    Uint32 linkId                      ;
} MqtObject ;
```

Fields

`mqtName`	Name of the MQT. For debugging purposes only.
`interface`	Function pointer interface to access the functions for this MQT.
`linkId`	ID of the link used by this MQT.

Comments

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

Constraints

None.

See Also

None.