# DSP/BIOS™ LINK

# ZERO COPY LINK DRIVER

# LNK 041 DES

## Version 0.90

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 zero copy link driver for DSP/BIOS™ LINK.

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

## 1.2 Terms & Abbreviations

| | |
|---|---|
| *DSPLINK* | DSP/BIOS™ LINK |
| ZCPY | Zero Copy |
| SHM | Shared Memory |
| SMA | Shared Memory Allocator |
| CHIRP | Channel I/O Request Packet |
| ⚐ | This bullet indicates important information.<br>Please read such text carefully. |
| ❑ | This bullet indicates additional information. |

## 1.3 References

| | | |
|---|---|---|
| 1. | LNK 012 DES | DSP/BIOS™ LINK<br>Link Driver |
| 2. | LNK 031 DES | DSP/BIOS™ LINK<br>Messaging Component |
| 3. | LNK 076 DES | DSP/BIOS™ LINK<br>Buffer Pools |

## 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 zero-copy driver provides a fast physical link between the GPP and the DSP, based on the concept of pointer exchange between the GPP and DSP applications.

This document provides a detailed description of the Zero Copy (ZCPY) driver design.

# 2 Requirements

R29 The physical link driver shall not perform any data copies between GPP memory, shared memory, and DSP memory areas.

R30 In addition to the configurations supported with the previous releases, this release shall support the following configurations

| Device | GPP Operating System | Features Supported |
|---|---|---|
| OMAP5912 | Montavista Linux Professional Edition 3.1 | DSP Bootloading<br>Zero copy data streaming.<br>Zero copy messaging<br>Processor copy data streaming<br>Processor copy messaging |
| VP-Hibari | PrOS | DSP Bootloading<br>Zero copy messaging |

R31 The build system shall allow the capability to directly expose the LDRV level APIs to reduce code. The APIs exposed shall however be consistent with the APIs exposed by DSP/BIOS™ LINK in the full configuration.

R32 The following table provides footprint expected in different configurations on **Hibari** running **PrOS on GPP.**

| # | Configuration Description | Components Included | | Footprint (bytes) | |
|---|---|---|---|---|---|
| | | GPP | DSP | GPP | DSP |
| 1 | Basic DSP bootloading only | DSP,<br>PROC (shim over DSP),<br>OSAL for PrOS | None | 1024 | None |
| 2 | Message transfer between GPP and DSP with static configuration for MSGQ objects and memory pools | DSP,<br>PROC (shim over DSP),<br>MSGQ (shim over LDRV_MSGQ)<br>POOL (shim over SMA),<br>ZCPYMQT,<br>OSAL for PrOS | MSGQ,<br>POOL,<br>ZCPYMQT | 1024 | 1024 |
| 3 | Both mentioned above | All of above | All of above | 2048 | 1024 |

R33 The above footprint numbers also assume static configuration for DSP/BIOS™LINK where no objects are created at runtime and are created statically prior to the application commences execution.

R34 Additionally, the following table provides the code footprint expected indifferent configurations on **OMAP5912** running **Montavista Linux Professional Edition 3.1 on GPP**.

| # | Configuration Description | Components Included | | Footprint (bytes) | |
|---|---|---|---|---|---|
| | | GPP | DSP | GPP | DSP |
| 1 | Basic DSP bootloading only | DSP,<br>PROC,<br>OSAL for MVLinux Pro 3.1 | None | 45K | 0 |
| 2 | Message transfer between GPP and DSP | DSP,<br>PROC,<br>MSGQ,<br>SMA,<br>ZCPYMQT,<br>OSAL for MVLinux Pro 3.1 | MSGQ,<br>POOL,<br>ZCPYMQT | 55K | 1500 |
| 3 | Bootloading,<br>Message transfer | All of above | All of above | 55K | 1500 |
| 4 | Bootloading,<br>Basic data streaming | DSP,<br>PROC,<br>CHNL,<br>SMA | IOM Driver for Streaming | 60K | 1500 |
| 5 | Bootloading,<br>Basic data streaming,<br>Message Transfer | All of above | All of above | 70K | 3000 |

R35 The performance measures for transferring data and message buffers between GPP and DSP are dependent on the H/W configuration and its clock frequency. Therefore it is best to categorize this data in terms of actual CPU cycles used while transferring data and message.

The following table specifies the CPU cycles used by LINK for transferring data and message buffers using the zero copy link drivers.

| H/W + S/W Configuration | Operation | CPU Cycles (average) |
|---|---|---|

| | | GPP | DSP |
|---|---|---|---|
| OMAP5912 running Montavista Linux Professional Edition 3.1 | Data Buffer Transfer using standard LINK APIs | 1000 | 1000 |
| | Message Transfer using standard LINK APIs | 1000 | 1000 |
| Hibari running PrOS | Message Transfer using shim LINK APIs and static configuration. | 500 | 500 |

In addition, the zero copy driver must meet the following generic requirement:

1. The link driver shall transfer messages at a higher priority than data buffers.
2. The physical link driver shall use a single interrupt for transferring messages over shared memory.
3. The code used for messaging shall be independent of the code used for data streaming to enable efficient method for message passing.
4. The existing APIs exposed by PROC and CHNL components shall remain unchanged.
5. The MSGQ APIs provided on GPP shall be identical to the APIs provided on DSP by the implementation of MSGQ in DSP/BIOS v5.10 (within the constraints of different capabilities provided by the OSes running on GPP and DSP).

# 3 Assumptions

The ZCPY driver design makes the following assumptions:

1. The hardware provides a shared memory area, to which both the GPP and the DSP have access.
2. The GPP OS provides a facility for translation of the physical address into the virtual address space of the user application.

# 4 Constraints

None.

# 5 High Level Design

In a multiprocessor system having shared access to a memory region, an efficient mode of data and message transfer can be implemented, which does not require copying of data across the processors. This mode of communication is called ZCPY transfer mode.

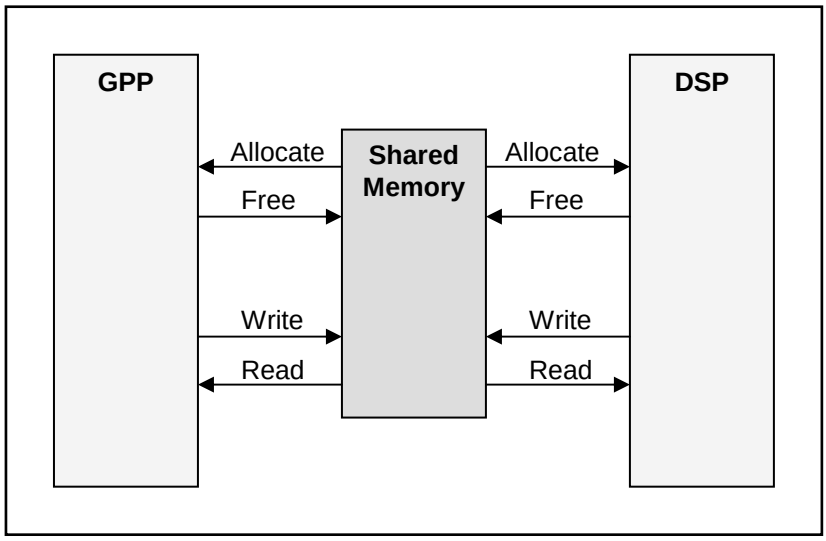


**Figure 1.** Basic architecture of system supporting ZCPY transfer mode

The ZCPY component on each processor shall provide the ability to allocate/free buffers from the shared memory (SHM) area. It shall also provide the client with means of writing into and reading from the shared memory region, so that the client can treat SHM buffers as user-space buffers.

## 5.1 Architecture overview

The ZCPY driver consists of three major sub-components:

1. Shared memory allocator (SMA)
2. Address translator
3. Shared Memory Inter Processor Signaling (SHMIPS)

The GPP-side ZCPY driver component shall need to implement all of the above three sub-components. The DSP component does not need the address translator sub-component, since the address conversion between the GPP virtual address and DSP physical address is completely done on the GPP-side. In addition, the DSP address space is uniform, without any kernel-user space division; hence, address translation is also not required from kernel to user space and vice versa.

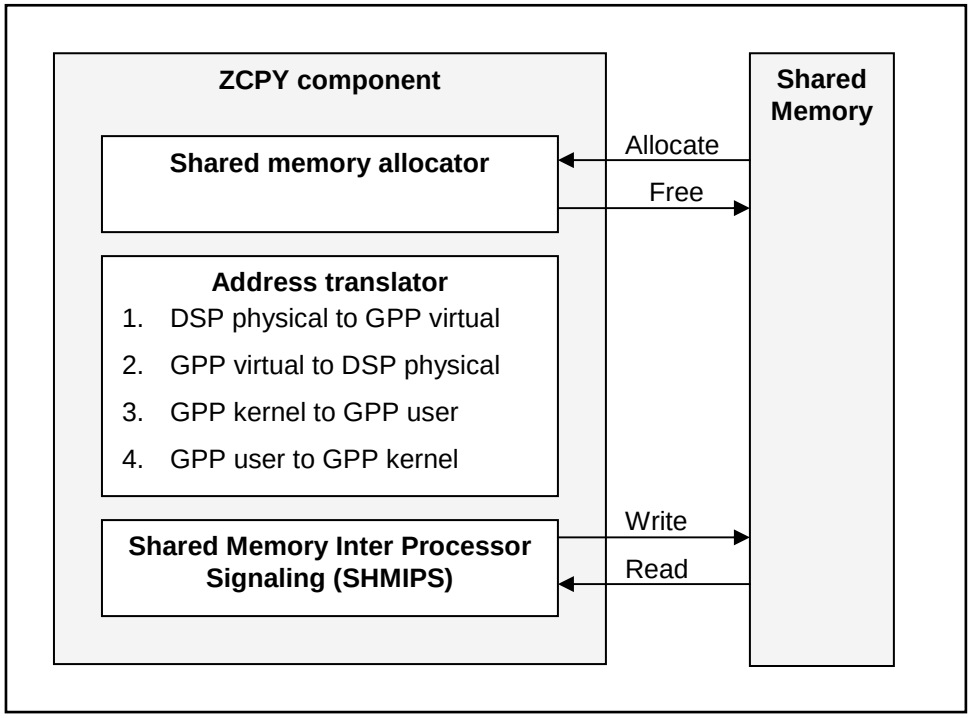


**Figure 2.** ZCPY sub-components

### 5.1.1 Shared memory allocator

To ensure that data is transferred between the processors without any copy, the application buffers must be directly allocated from shared memory. This requires a memory manager that allows the user to allocate/free buffers from the predefined shared memory region.

For the detailed design of the shared memory allocator, please refer to the Shared Memory Allocator (SMA) design document [Ref. 5].

### 5.1.2 Address translator

The address translator performs four different types of address translation:

1. DSP physical to GPP virtual
2. GPP virtual to DSP physical
3. GPP kernel to GPP user
4. GPP user to GPP kernel.

The conversion between DSP & GPP addresses (1 & 2) is required irrespective of the OS on the GPP. This address translation is completely performed on the GPP-side, through predefined mappings between the addresses of the shared memory on the GPP & DSP sides.

The conversion between user and kernel address space (3 & 4) is required only for GPP OSes like Linux, having a user/kernel-space division. The implementation of this part of the address translation is OS-specific.

The functionality of the address translator may be divided across components in DSP/BIOS™ LINK. For example, while the address translation between the DSP & GPP addresses may be performed at the link driver level, address translation

between the kernel and user spaces (if required) shall be performed at the user/kernel-space boundary.

### 5.1.3 Shared Memory Inter Processor Signaling

For data transfer, the link driver manages a configurable number of logical channels. The SHMIPS component manages the transfer of data and messages across the two processors. For this, it uses the shared memory control structure and interrupts between the processors to inform about any changes in status of buffer/message availability on the channels.

## 5.2 Control flow

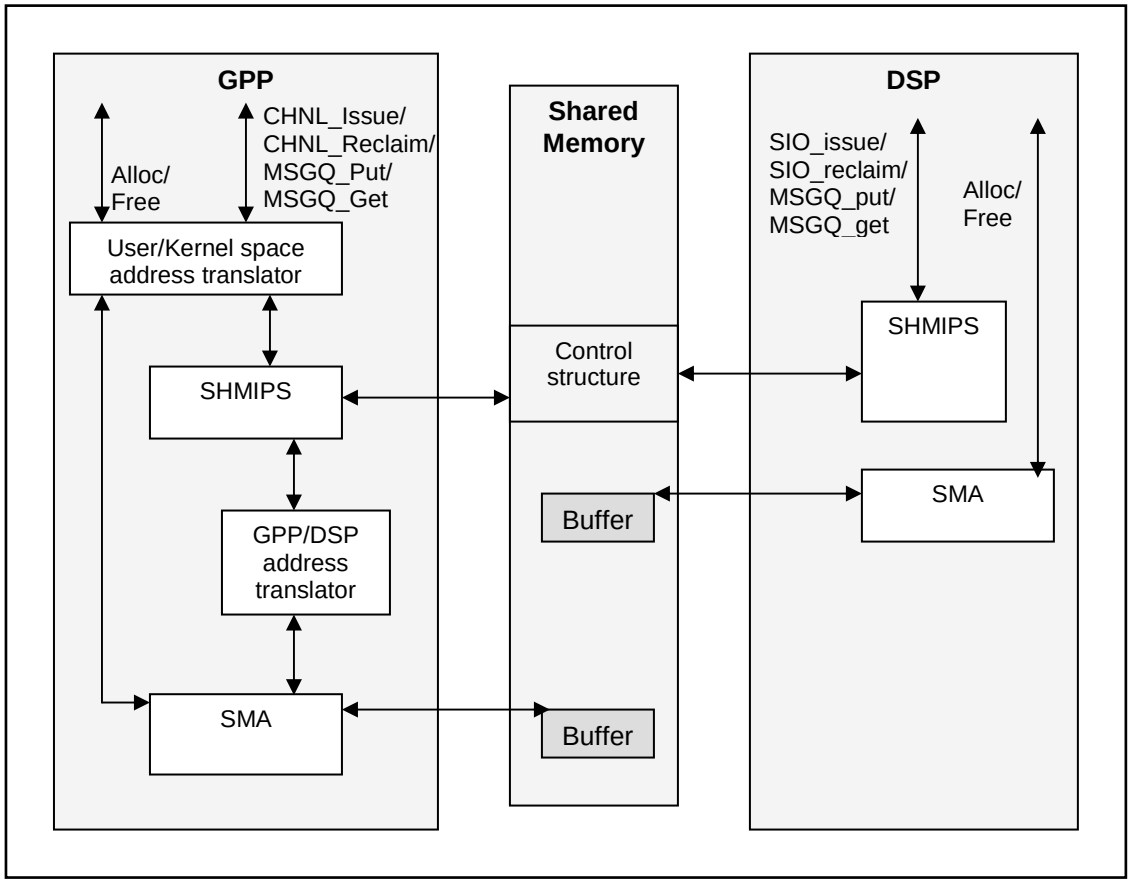


**Figure 3.** ZCPY sub-component interaction

**ZCPY message/buffer allocation**

A request for allocation of a zero-copy message or buffer on the GPP is serviced by the GPP-side shared memory manager. It allocates a buffer of the requested size from the shared memory region reserved for usage by the shared memory allocator. After translation of the address (if required) from the kernel space into the user space, this buffer is returned to the user application that had requested the buffer allocation.

On the DSP-side, the message or buffer allocation proceeds in a similar manner.

**ZCPY message/buffer freeing**

On a GPP-side call to free a zero copy message or buffer, the user address is translated (if required) into its equivalent address in the kernel space. The request is

then passed to the shared memory allocator, which frees the buffer and returns it to the available memory pool.

On the DSP-side, the message or buffer is freed in a similar manner.

**ZCPY buffer exchange for data transfer**

When the GPP and DSP are ready to transfer data on a specific channel, they both issue a buffer to the channel in the ZCPY link driver. Data transfer does not take place until both sides have issued a buffer on the same channel. On issuing a buffer to a channel, the corresponding driver updates the SHM control structure as required. When a buffer is available on both sides, the pointers are exchanged. The buffer allocated by the DSP-side application is sent to the GPP-side user application, whereas the one on the GPP-side is sent to the DSP-side application.

**ZCPY message transfer**

When either the GPP or DSP is ready to send a message to the other processor, it sends the message to the SHMIPS component. On receiving a message from the other processor, the SHMIPS component makes a callback to the ZCPY MQT component, which places the received message onto the appropriate local message queue. Zero copy message transfer occurs in a similar manner to the message transfer for processor copy mode. In the case of zero-copy messaging, however, the pointer to the message allocated from shared memory is directly transferred to the remote processor, and does not require intermediate allocation/freeing of the messages at the driver level.

## 5.3 Zero copy mechanism

The data transfer mechanism between the GPP and DSP for zero-copy mode is based on pointer-exchange.

The typical data transfer control flow during zero copy buffer exchange is illustrated in the following figure. The mechanism is the same for both input and output channel modes.

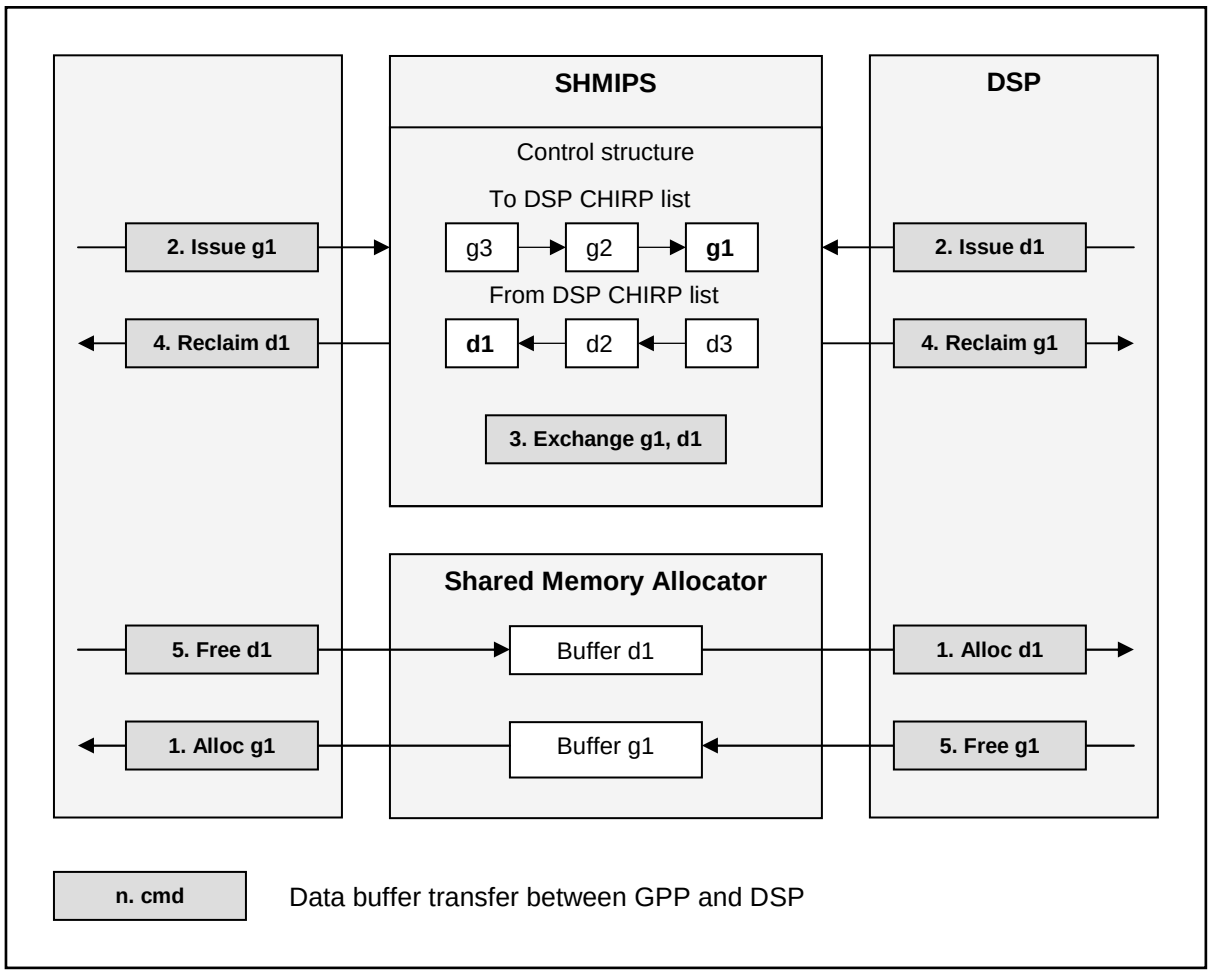


**Figure 4.** Zero copy buffer exchange during data transfer

The message transfer mechanism between the GPP and DSP for zero-copy mode is based on pointer-passing. The typical message transfer control flow during zero copy pointer passing is illustrated in the following figure.

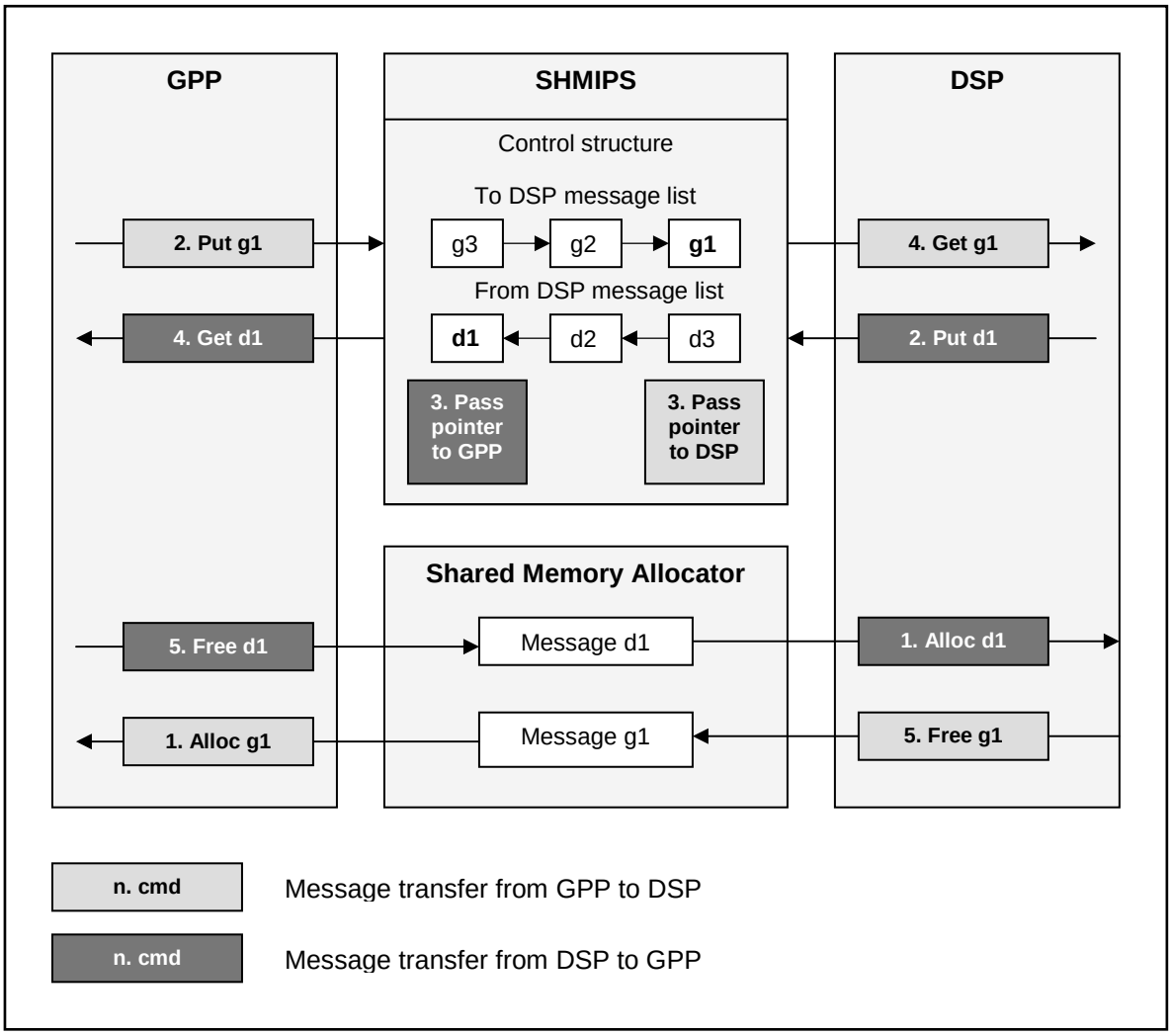


**Figure 5.** Zero copy pointer passing during message transfer.

The following basic steps are followed during data or message transfer between the GPP and the DSP:

1. **`Alloc buffer:`** The GPP and DSP allocate data buffers/messages using the Shared Memory Allocator.
2. **`Issue/Put buffer:`** The GPP and DSP send the data buffer/message to the SHMIPS component. The SHMIPS component sends the command across to the other processor. In case of data transfer, the CHIRP is added to the "`dataToDsp`" list within the SHM control structure maintained by the SHMIPS component. In case of message transfer, the message is added to the "`msgToDsp`" list within the SHM control structure.
3. **`Exchange/pass pointers:`** In case of data transfer, the channel driver on both processors exchanges the data buffers present within the shared memory. In case of message transfer, the message pointer is passed to the other processor.

4. **`Reclaim/Get buffer:`** The SHMIPS component sends the received data buffer/message to the channel/message driver.
5. **`Free buffer:`** Once all processing is complete, the application can free the buffer it received from the other processor.

## 5.4 ZCPY driver

The ZCPY driver is implemented as a physical link on the GPP-side and an IOM driver with supporting libraries on the DSP-side.

The ZCPY driver utilizes the shared memory between the GPP and DSP for implementing the data transfer and message transfer protocols.

The shared memory shall have the following basic areas for both ZCPY and PCPY (order may be different):

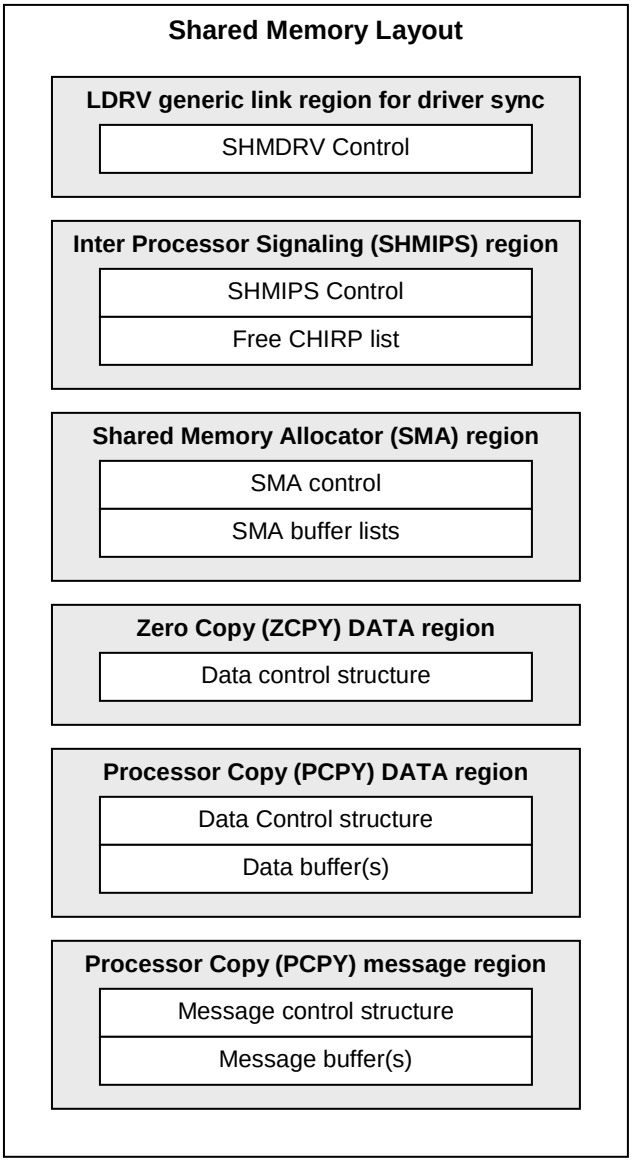


**Figure 6.** Shared memory layout

The shared memory area used by the link driver is divided up into several regions used by the various sub-components of the ZCPY and PCPY link drivers.

The different regions are:

1. **LDRV generic link region for driver init (SHMDRV):** The LDRV generic link region contains the control structure for synchronization of the *DSPLINK* drivers on the GPP and the DSP.
2. **Shared Memory Inter Processor Signaling (SHMIPS) region:** The SHMIPS region contains the control structure and Free CHIRP list used for communicating events between the two processors.
3. **Shared Memory Allocator (SMA) region:** The SMA SHM region contains the control structures and buffer lists for allocating and freeing buffers shared between the GPP and DSP.
4. **Zero Copy (ZCPY) DATA region:** The ZCPY Data region contains the control structure needed by the ZCPY channel driver for implementing the data transfer protocol between the GPP and the DSP.
5. **Processor Copy (PCPY) DATA region:** The PCPY Data region contains the control structure and data buffer(s) needed by the PCPY channel driver for implementing the data transfer protocol between the GPP and the DSP.
6. **Processor Copy (PCPY) message region:** The PCPY message region contains the control structure needed by the PCPY MQT for implementing the message transfer protocol between the GPP and the DSP.

### 5.4.1 GPP-side

#### *5.4.1.1 Component interaction*

The component interaction diagram indicates the placement of the various ZCPY driver components within *DSPLINK*.

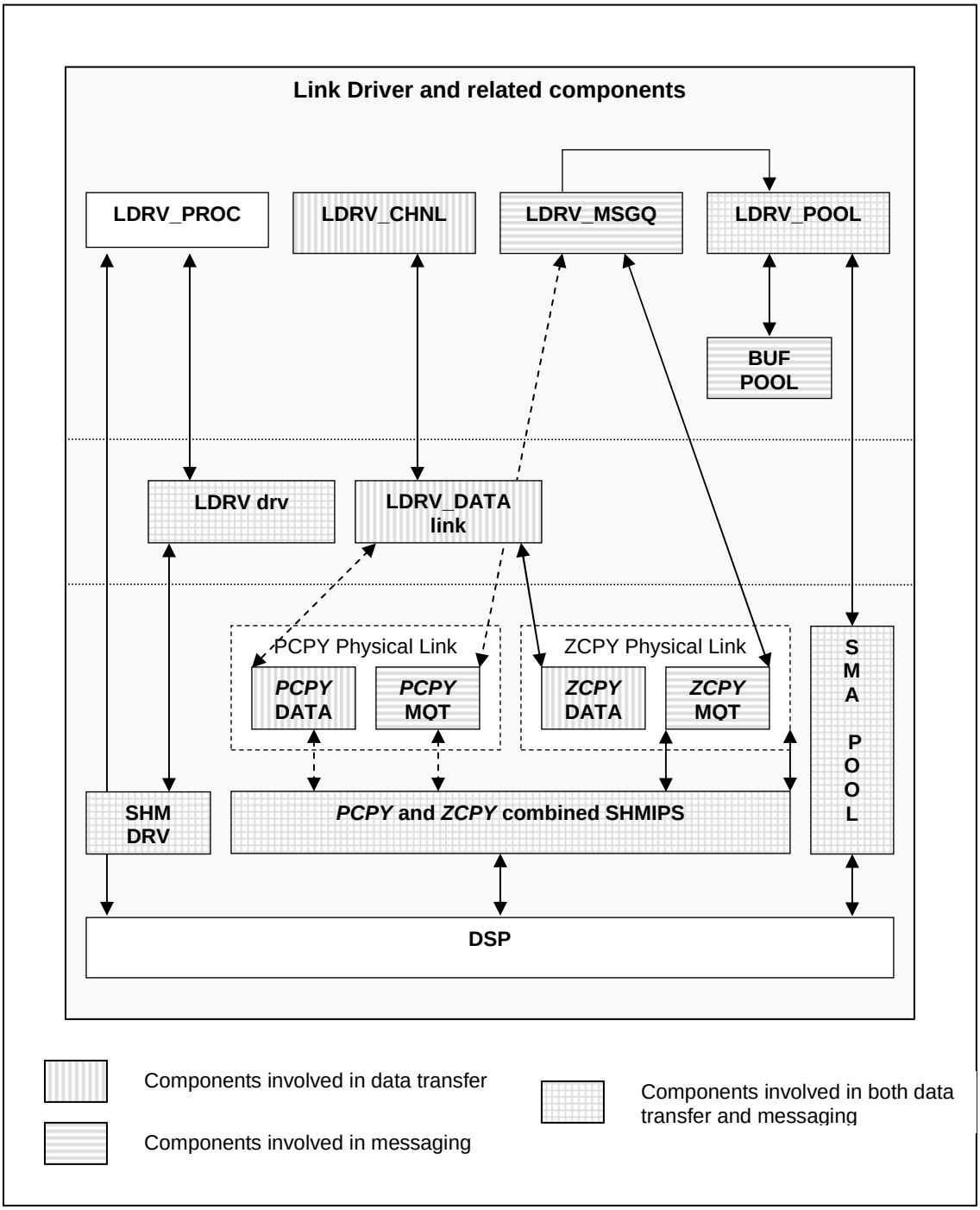


**Figure 7.** GPP-side component interaction

### 5.4.2 DSP-side

*5.4.2.1 Overview*

The DSP-side of the DSPLINK data transfer functionality shall conform to the IOM interface of DSP/BIOS™. In addition, the messaging functionality shall conform to the MSGQ interface.

The *DSPLINK* data transfer protocol shall be contained within the IOM driver. The messaging content shall be within a separate library to allow users to scale in only data transfer or only messaging. In addition, the common hardware specific functionality required by both data transfer and messaging shall be within a separate H/W link library. The H/W link library includes the SHMIPS component for sending events to and receiving them from the GPP. It also includes the common hardware initialization, finalization and handshaking code required for the functioning of both data and message transfer.

Scalability for CHNL and MSGQ shall be provided through compile-time flags, which shall be set by the common configuration tool. In addition, the choice of static/dynamic MSGQ shall also be made through the common configuration tool.

This design allows the flexibility of an optimized and high-performance implementation of the MQT and data transfer protocol for a particular physical link. Since all the protocol content shall be within the driver libraries, the implementation can be optimized for the case where only the zero-copy transfer mechanism is desired. In addition, the common functionality between the different physical link data transfer protocols can also be separated out into the common IOM functionality layer.

#### *5.4.2.2 Component interaction*

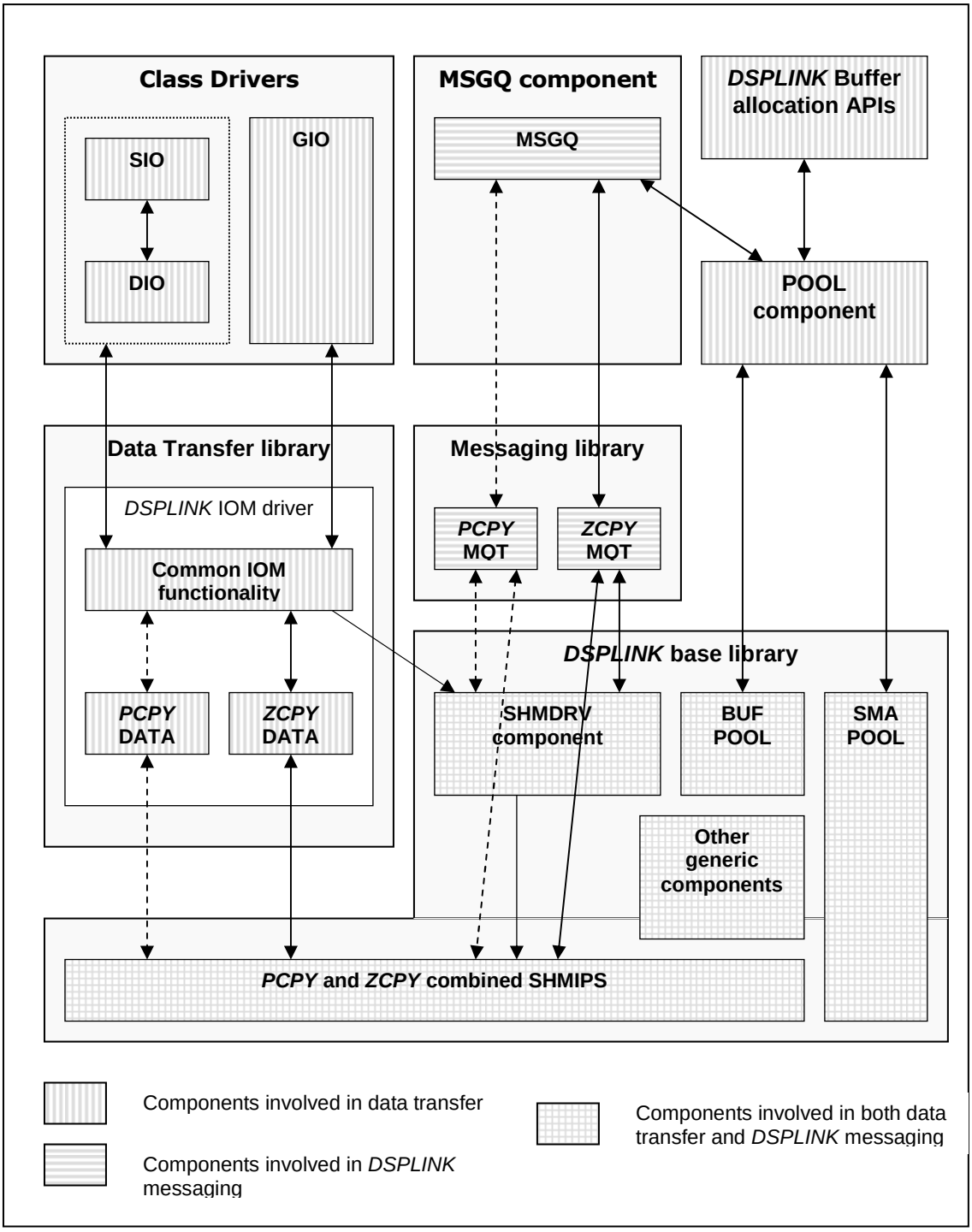


**Figure 8.** DSP-side component interaction

## 5.5 SHMIPS component

### 5.5.1 Overview

The Shared Memory Inter-processor signaling (SHMIPS) component is responsible for notifying an event to its peer on the remote processor. This component shall use the

services provided on the hardware platform. It shall provide APIs, which shall be used by upper layers to establish communication amongst peers at that level.

### 5.5.2 Services provided

The SHMIPS component shall provide the following basic services:

1. Register an event.
2. Unregister an event.
3. Send a buffer/message to the remote processor.
4. Notify the remote processor about an event.

### 5.5.3 Design

The SHMIPS component shall maintain lists of messages, which are shared between the GPP and the DSP. There shall be two unidirectional lists of messages, for messages to and from the DSP. Similar lists shall also be used for data transfer.

To protect these shared lists, the SHMIPS component shall utilize the services of a generic component that shall provide critical section protection between the two processors. For more details, please refer to the SMA design document [Ref. 5].

The SHMIPS component shall maintain the following information to allow transfer of messages and data between the two processors:

1. Shared lists of messages and CHIRPs are maintained within the SHMIPS control structure.
2. Free list of CHIRPs is maintained within the shared memory to be able to queue data transfer requests for transfer to the other processor. This list shall be configured with the maximum number of outstanding requests on all channels at any time. Information for this shall be obtained from the static configuration.
3. Shared critical section objects are maintained within the SHMIPS control structure to provide protection for the shared lists.

In the SHMIPS component, the prioritization between data and message transfer shall happen on the receiving side within the ISR. On the sender's end, the CHIRP/message shall simply be queued on the appropriate shared list.

The physical link interface for the ZCPY driver can be split into SHM DRV, ZCPY Data and ZCPY MQT. These interfaces are explained in details in the following sections.

## 5.6 SHMDRV component

### 5.6.1 Overview:

The SHMDRV component is responsible for initialization and finalization of the shared memory link sub-components. The SHMDRV implementation of the generic driver layer requires a shared memory area between the two processors.

### 5.6.2 Services provided

The SHMDRV shall be responsible for supporting the following features:

- Initialization and finalization of the link sub-components including the IPS component(s) and POOL interface.
- Handshaking with the remote processor for synchronization of the drivers.

- Verification of configuration match between the two processors. This component checks whether the scalability configuration on both processors is the same, for example if MSGQ, CHNL etc are enabled.

### 5.6.3 Design

The SHMDRV uses a reserved area within the shared memory region for maintaining the control structure for synchronization of the GPP and DSP-side *DSPLINK* drivers through handshaking.

It provides an implementation of the generic link interface as required by the LDRV_DRV layer:

```
LinkInterface SHMDRV_Interface = {
    &SHMDRV_Initialize,
    &SHMDRV_Finalize,
    &SHMDRV_Handshake
#if defined (DDSP_DEBUG)
    ,&SHMDRV_Debug
#endif /* if defined (DDSP_DEBUG) */
} ;
```

## 5.7 ZCPY MQT component

### 5.7.1 Overview

The ZCPY MQT component on a processor is responsible for implementing the message transfer protocol for the transport. The messages transferred by this MQT shall not require an intermediate copy into shared memory.

### 5.7.2 Services provided

The MQT shall be responsible for supporting the following features:

- Locating MSGQs on the remote processor by name.
- Releasing previously located MSGQs on the remote processor.
- Sending messages to MSGQs on the remote processor.
- Receiving messages from the remote processor and transferring to the appropriate local MSGQ.

### 5.7.3 Design

The SHMIPS provides the basic capability to transfer buffers between the GPP and DSP. This gives the complete functionality for transferring messages between the GPP-side and DSP-side MQTs as required by the ZCPY MQT. Hence the ZCPY message transfer protocol does not require any additional shared control information between the two MQTs for coordination between them.

The ZCPY MQT implements the functions within the MQT interface table as required by the MSGQ component.

The basic services provided by the ZCPY MQT are tabulated below along with additional information about their design:

| Basic services provided to MSGQ | SHMIPS services used | Design information |
|---|---|---|

| | | |
|---|---|---|
| Open MQT | Register | ▪ Create and initialize the MQT state object<br>▪ Register callback with the SHMIPS component. |
| Close MQT | Unregister | ▪ Unregister callback with the SHMIPS component.<br>▪ Delete MQT state object |
| Send message | Send | ▪ Send event to SHMIPS. |
| Receive message | Callback | ▪ Receive event from SHMIPS<br>▪ Send message to local MSGQ |
| Locate remote message queue | Send, Callback | ▪ Make locate request to remote MQT by sending event to SHMIPS containing control message.<br>▪ If the locate call is synchronous, wait for receiving locate acknowledgement message from remote MQT as an SHMIPS event containing control message.<br>If the locate call is asynchronous, return from the function without blocking. When the locate acknowledgement arrives from the remote processor, allocate and send an asynchronous locate message to the reply message queue specified by the caller. |
| Release remote message queue | | Nothing to be done |

## 5.8 ZCPY DATA component

### 5.8.1 Overview

The ZCPY DATA component on a processor is responsible for implementing the data transfer protocol between the two processors based on the zero-copy pointer exchange mechanism. The data buffers transferred using this driver shall not require an intermediate copy into shared memory.

### 5.8.2 Services provided

The ZCPY DATA shall be responsible for supporting the following features:

- Opening and closing logical data channels between the two processors.
- Issuing and reclaiming data buffers on these logical channels.
- Canceling data transfer and idling the channels.

### 5.8.3 ZCPYDATA Interface

The ZCPYDATA is the interface for the Data transfer functionality of the physical link. The interfaces are exported by the `ZCPYDATA` to the `LDRV_DATA` layer for hookup with LDRV component. The interface exported to the `LDRV_DATA` layer is as follows:

```
LinkDataInterface ZCPYDATA_Interface = {
    &ZCPYDATA_Initialize,
    &ZCPYDATA_Finalize,
    &ZCPYDATA_OpenChannel,
```

```
    &ZCPYDATA_CloseChannel,
    &ZCPYDATA_CancelIo,
    &ZCPYDATA_Request
#if defined (DDSP_DEBUG)
    ,&ZCPYDATA_Debug
#endif /* if defined (DDSP_DEBUG) */
} ;
```

### 5.8.4 Design

The ZCPY data transfer protocol works on the issue-reclaim model. It transfers buffers between the processors using pointer exchange mechanism. The data transfer protocol provides connectivity between the two processors through a configured number of logical channels. For this, the component uses a control region shared between the ZCPY DATA components on the two processors. This control region contains information about data buffer availability on the various logical channels, and other information used for synchronizing between the drivers on the two processors.

The ZCPY DATA transfer component uses the features provide by the IPS component for transferring the data between the DSP and GPP. Both the GPP and DSP issue buffers for data transfer and the ZCPY driver exchanges the buffer pointers to complete the transfer.

# 6 Sequence Diagrams

The following sequence diagrams show the control flow for a few of the important functions to be implemented within the *DSPLINK* zero copy link driver.

The sequence diagrams indicate the messaging control flow through the MQT component and its interaction with the rest of the *DSPLINK* components and the MSGQ and POOL components on the DSP-side.

While the following sequence diagrams show the control flow for the DSP-side of *DSPLINK*, the control flow on the GPP-side is similar, and is not detailed in this document.

- *The dashed arrow in all sequence diagrams indicates an indirect control transfer, which does not happen through a direct function call.*

## 6.1 ZCPYMQT_init ()

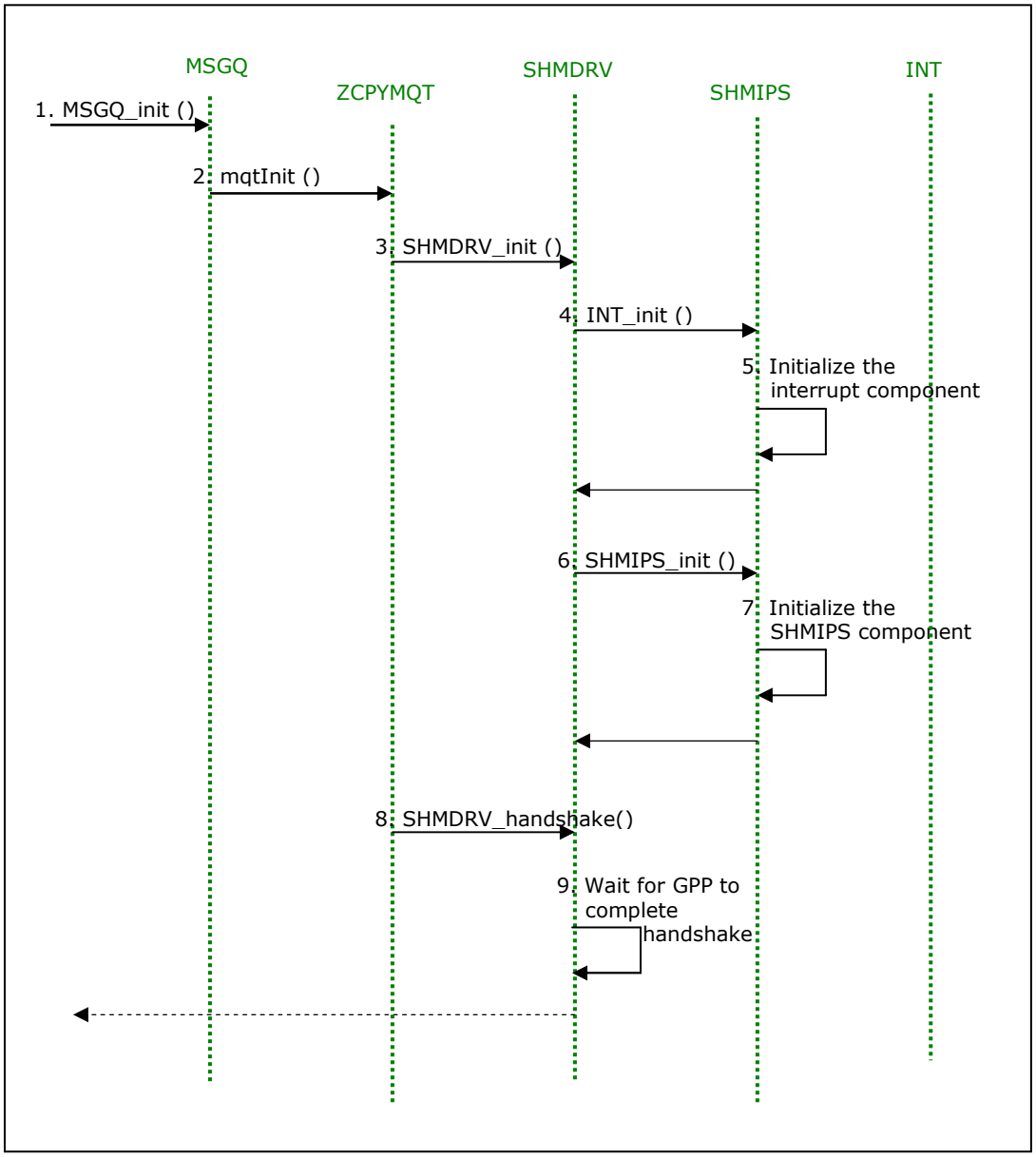


**Figure 9.** On the DSP: ZCPYMQT_init () control flow

- *MSGQ_init () is not called by the application. It gets invoked internally during DSP/BIOS™ initialization.*

## 6.2 ZCPYMQT_open ()

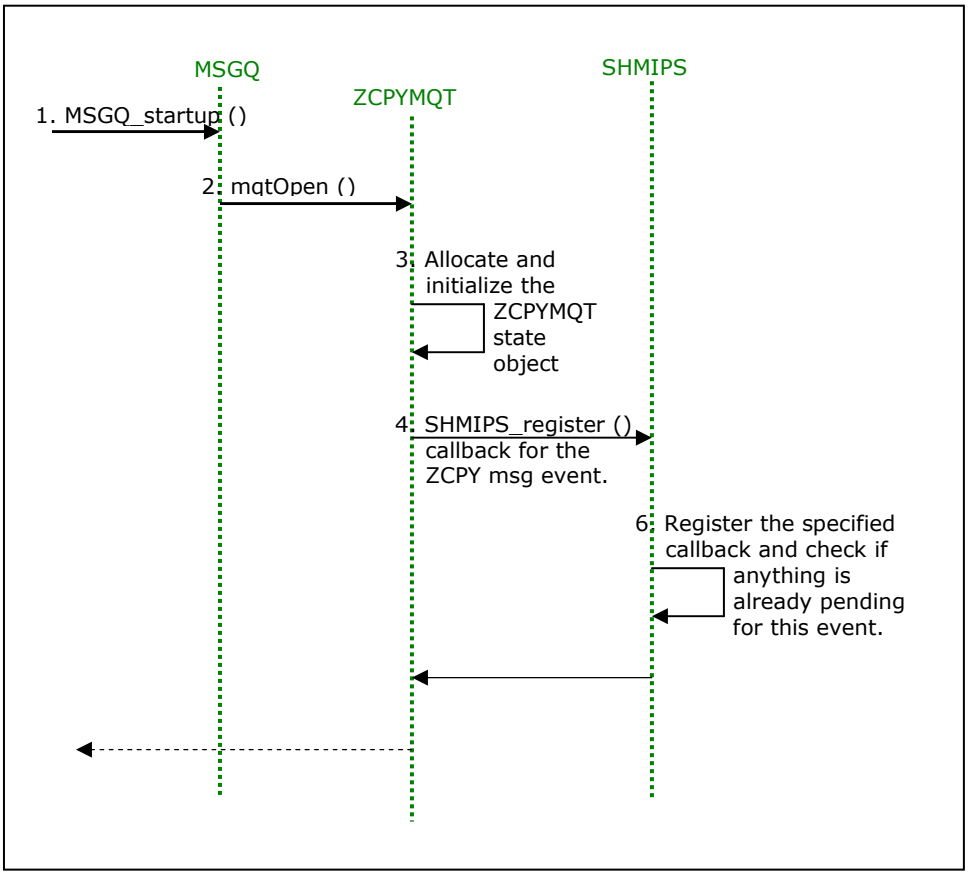


**Figure 10.** On the DSP: ZCPYMQT_open () control flow

- *MSGQ_startup () is not called by the application. It gets invoked internally during DSP/BIOS™ initialization.*

## 6.3 ZCPYMQT_close ()

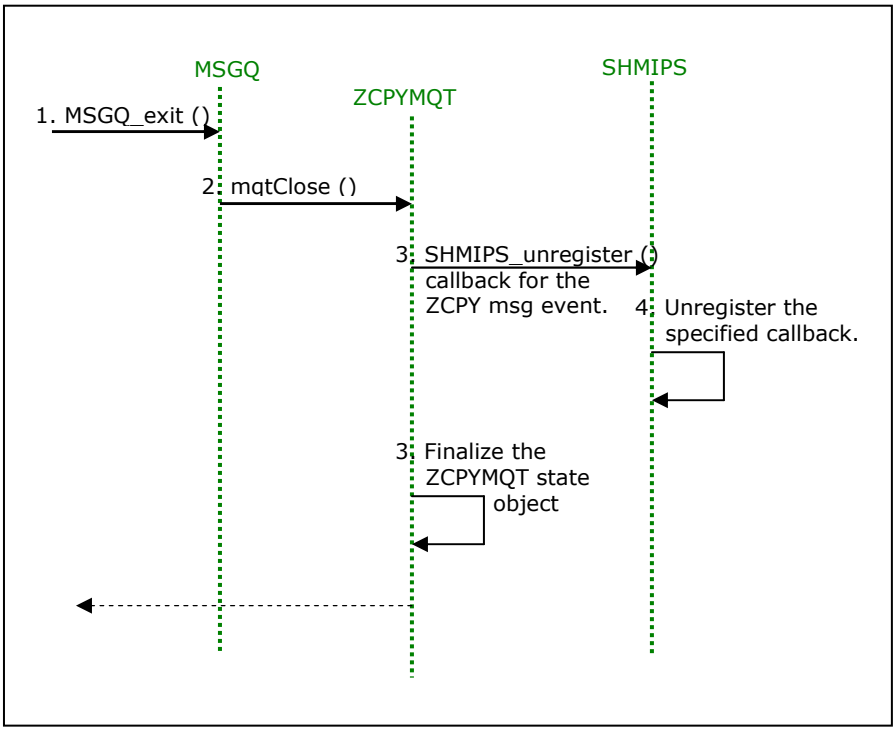


**Figure 11.** On the DSP: ZCPYMQT_close () control flow

- *MSGQ_exit () is not called by the application.*

## 6.4 ZCPYMQT_put ()

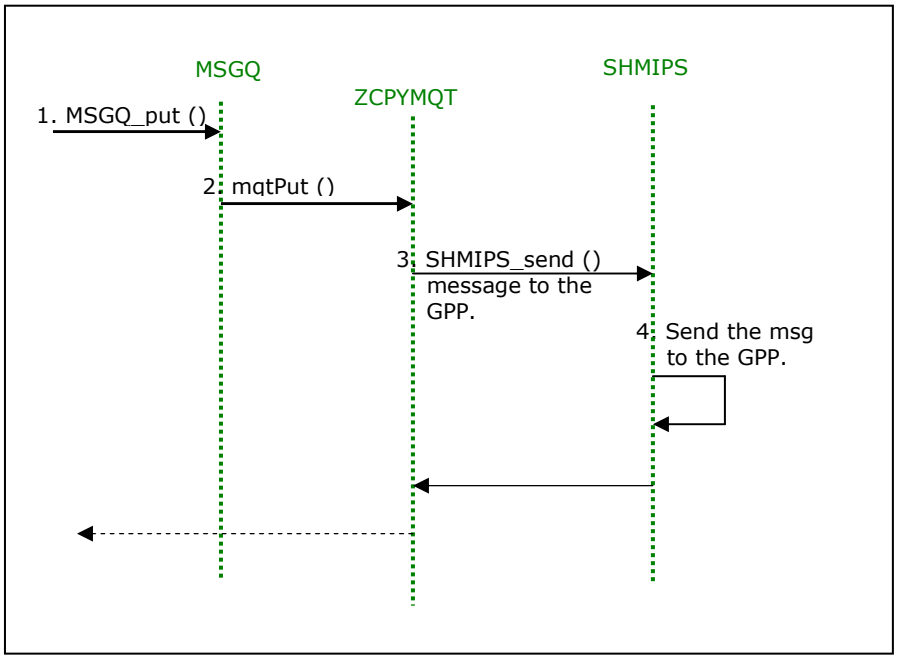


**Figure 12.** On the DSP: ZCPYMQT_put () control flow

## 6.5 ZCPYMQT_locate ()

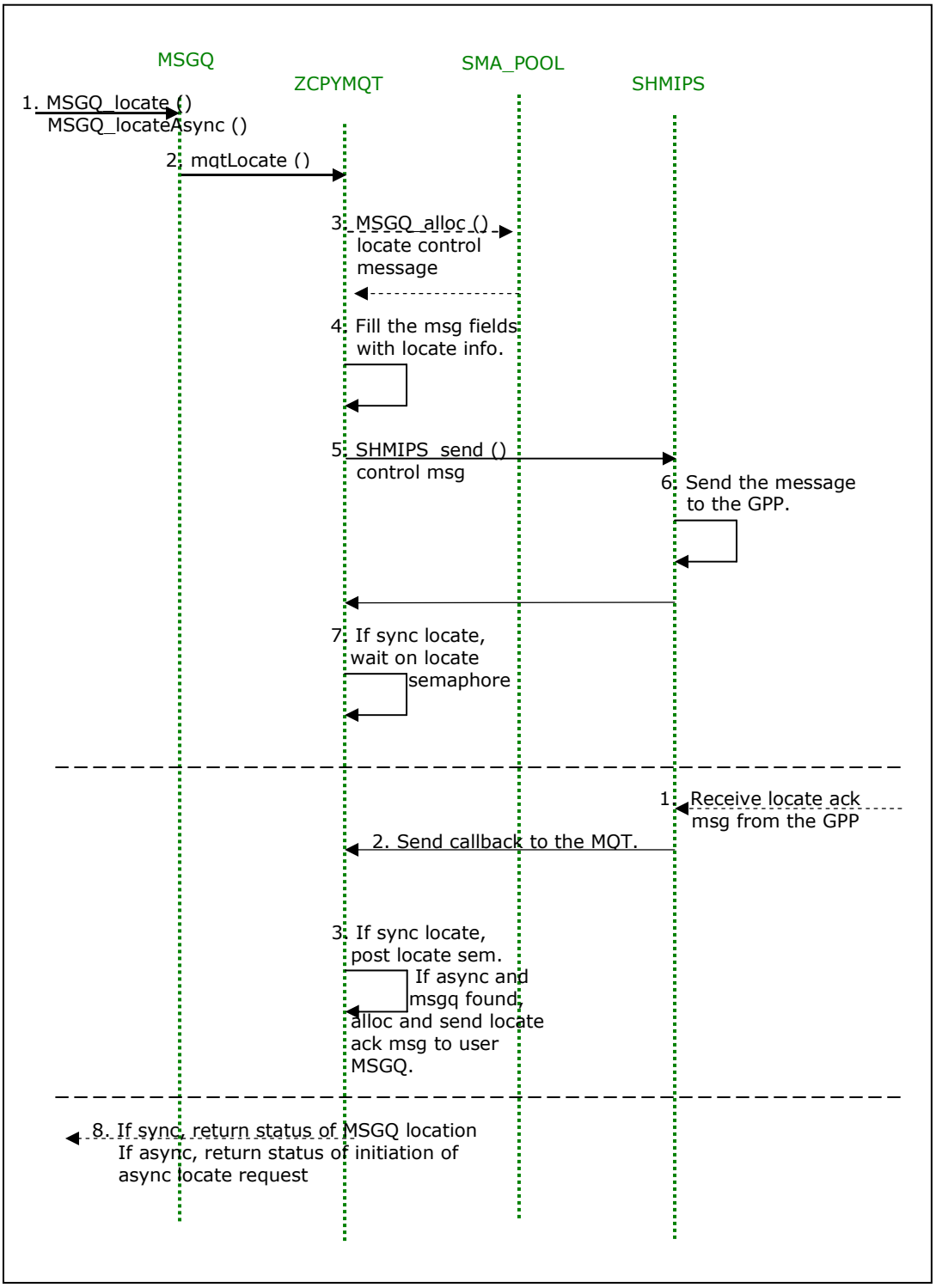


**Figure 13.** On the DSP: ZCPYMQT_locate () control flow

- *The sequence between the two horizontal dashed lines occurs asynchronously, and may occur before or after locate has timed out (for synchronous locate) or function call has returned (for asynchronous locate).*

## 6.6 ZCPYMQT_release ()

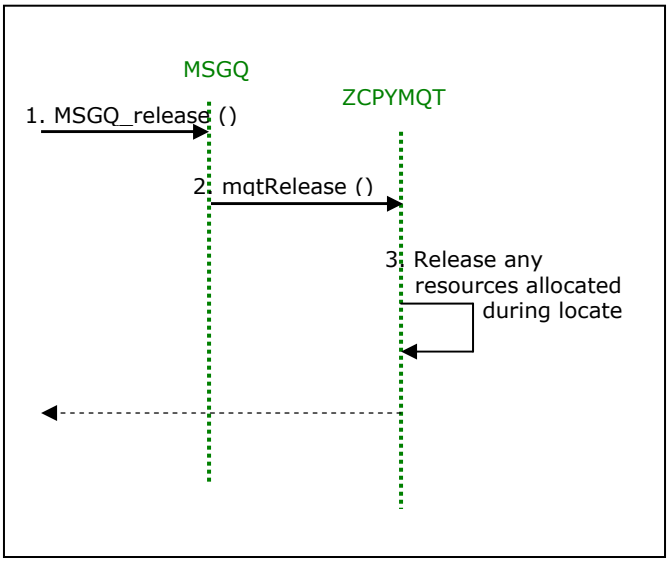


**Figure 14.** On the DSP: ZCPYMQT_release control flow

# 7 SHMIPS

The SHMIPS component has the same design on both the GPP and DSP sides. This section primarily refers to the GPP side design. However, the DSP-side design shall contain the same enumerations, structures, and API definitions, with minimal changes for different naming conventions on the GPP and DSP-sides.

## 7.1 GPP and DSP side low level design

### 7.1.1 Constants & Enumerations

#### *7.1.1.1 SHMIPS_CTRL_SIZE*

This constant defines the size of the shared memory control structure required by the SHMIPS component.

**Definition**

```
#define SHMIPS_CTRL_SIZE (sizeof (ShmIpsShmCtrl))
```

**Comments**

This constant is used by configuration script for assigning shared memory regions to the various components using the same memory area.

**DSP-side:**

The constant definition is:

```
#define SHMIPS_CTRL_SIZE (sizeof (SHMIPS_ShmControl))
```

**Constraints**

None.

**See Also**

`ShmIpsShmCtrl`

*7.1.1.2 SHMIPS_IRP_SIZE*

This constant defines size of the IO Request Packet used within the SHMIPS.

**Definition**

```
#if defined (CHNL_COMPONENT)
#define SHMIPS_IRP_SIZE       (sizeof (LDRVChnlIRP))
#else /* if defined (CHNL_COMPONENT) */
#define SHMIPS_IRP_SIZE       0
#endif /* if defined (CHNL_COMPONENT) */
```

**Comments**

This constant is used for calculation of the total shared memory size required by the SHMIPS component. In addition to the control structure, the SHMIPS also needs shared memory space for the free CHIRP list in shared memory when the CHNL component is enabled.

**DSP-side:**

The constant definition is:

```
#if defined (CHNL_COMPONENT)
#define SHMIPS_IRP_SIZE       (sizeof (CHNL_Irp))
#else /* if defined (CHNL_COMPONENT) */
#define SHMIPS_IRP_SIZE       0
#endif /* if defined (CHNL_COMPONENT) */
```

**Constraints**

None.

**See Also**

None.

*7.1.1.3 CHNL_EVENTS*

Defines the number of events supported for CHNL. At one time, multiple physical links can be selected for CHNL.

**Definition**

```
#if defined (CHNL_COMPONENT)
#if defined (CHNL_ZCPY_LINK) && defined (CHNL_PCPY_LINK)
#define CHNL_EVENTS 2
#else /* if defined (CHNL_ZCPY_LINK) && defined (CHNL_PCPY_LINK) */
#define CHNL_EVENTS 1
#endif /* if defined (CHNL_ZCPY_LINK) && defined (CHNL_PCPY_LINK) */
#else /* if defined (CHNL_COMPONENT) */
#define CHNL_EVENTS 0
#endif /* if defined (CHNL_COMPONENT) */
```

**Comments**

The value of this constant differs based on the physical link scalability option selected. This constant is used for calculation of the value of maximum events supported by the SHMIPS component for a particular scalability configuration.

**Constraints**

None.

**See Also**

MAX_SHMIPS_EVENTS
MSGQ_EVENTS

#### *7.1.1.4 MSGQ_EVENTS*

Defines the number of events supported for MSGQ. At a time, only one physical link can be selected for MSGQ.

**Definition**

```
#if defined (MSGQ_COMPONENT)
#define MSGQ_EVENTS 1
#else /* if defined (MSGQ_COMPONENT) */
#define MSGQ_EVENTS 0
#endif /* if defined (MSGQ_COMPONENT) */
```

**Comments**

The value of this constant differs based on the physical link scalability option selected. This constant is used for calculation of the value of maximum events supported by the SHMIPS component for a particular scalability configuration.

**Constraints**

None.

**See Also**

MAX_SHMIPS_EVENTS
CHNL_EVENTS

#### *7.1.1.5 MAX_SHMIPS_EVENTS*

This constant defines the maximum number of events supported by the SHMIPS component.

**Definition**

```
#define MAX_SHMIPS_EVENTS  (CHNL_EVENTS + MSGQ_EVENTS)
```

**Comments**

The number of events supported by the SHMIPS component is different based on whether the CHNL and/or the MSGQ component(s) are enabled. It also differs based on the physical link scalability option selected.

**Constraints**

None.

**See Also**

CHNL_EVENTS
MSGQ_EVENTS

*7.1.1.6 ShmIpsEvent*

This enumeration defines the types of events supported by the SHMIPS component. This enumeration defines the events based on scalability options selected for CHNL and MSGQ, as well as physical links.

**Definition**

```
typedef enum {
#if     defined (MSGQ_COMPONENT)                \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  defined (CHNL_PCPY_LINK)
    ShmIpsEventMsg      = 0,
    ShmIpsEventDataZcpy = 1,
    ShmIpsEventDataPcpy = 2
#endif /* if     defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     !defined (MSGQ_COMPONENT)               \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  defined (CHNL_PCPY_LINK)
    ShmIpsEventDataZcpy = 0,
    ShmIpsEventDataPcpy = 1
#endif /* if     !defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     !defined (MSGQ_COMPONENT)               \
    &&  !defined (CHNL_ZCPY_LINK)               \
    &&  defined (CHNL_PCPY_LINK)
    ShmIpsEventDataPcpy = 0
#endif /* if     !defined (MSGQ_COMPONENT)
             &&  !defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     !defined (MSGQ_COMPONENT)               \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  !defined (CHNL_PCPY_LINK)
    ShmIpsEventDataZcpy = 0
#endif /* if     !defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  !defined (CHNL_PCPY_LINK) */

#if     defined (MSGQ_COMPONENT)                \
    &&  !defined (CHNL_ZCPY_LINK)               \
    &&  !defined (CHNL_PCPY_LINK)
    ShmIpsEventMsg      = 0
#endif /* if     defined (MSGQ_COMPONENT)
             &&  !defined (CHNL_ZCPY_LINK)
             &&  !defined (CHNL_PCPY_LINK) */

#if     defined (MSGQ_COMPONENT)                \
    &&  !defined (CHNL_ZCPY_LINK)               \
    &&  defined (CHNL_PCPY_LINK)
    ShmIpsEventMsg      = 0,
    ShmIpsEventDataPcpy = 1
```

```
#endif /* if     defined (MSGQ_COMPONENT)
             &&  !defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     defined (MSGQ_COMPONENT)                \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  !defined (CHNL_PCPY_LINK)
    ShmIpsEventMsg      = 0,
    ShmIpsEventDataZcpy = 1
#endif /* if     defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  !defined (CHNL_PCPY_LINK) */
} ShmIpsEvent ;
```

**Fields**

| | |
|---|---|
| `ShmIpsEventMsg` | Message transfer event.<br><br>Only defined if MSGQ component is enabled. |
| `ShmIpsEventDataZcpy` | ZCPY data transfer event.<br><br>Only defined if ZCPY LINK and CHNL component are enabled. |
| `ShmIpsEventDataPcpy` | PCPY data transfer event.<br><br>Only defined if PCPY LINK and CHNL component are enabled. |

**Comments**

**DSP-side:**

The enumeration definition is:

```
typedef enum {
#if     defined (MSGQ_COMPONENT)                \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  defined (CHNL_PCPY_LINK)
    SHMIPS_EventMsg      = 0,
    SHMIPS_EventDataZcpy = 1,
    SHMIPS_EventDataPcpy = 2
#endif /* if     defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     !defined (MSGQ_COMPONENT)               \
    &&  defined (CHNL_ZCPY_LINK)                \
    &&  defined (CHNL_PCPY_LINK)
    SHMIPS_EventDataZcpy = 0,
    SHMIPS_EventDataPcpy = 1
#endif /* if     !defined (MSGQ_COMPONENT)
             &&  defined (CHNL_ZCPY_LINK)
             &&  defined (CHNL_PCPY_LINK) */

#if     !defined (MSGQ_COMPONENT)               \
    &&  !defined (CHNL_ZCPY_LINK)               \
    &&  defined (CHNL_PCPY_LINK)
```

```
        SHMIPS_EventDataPcpy = 0
    #endif /* if     !defined (MSGQ_COMPONENT)
                &&  !defined (CHNL_ZCPY_LINK)
                &&  defined (CHNL_PCPY_LINK) */

    #if     !defined (MSGQ_COMPONENT)                \
        &&  defined (CHNL_ZCPY_LINK)                 \
        &&  !defined (CHNL_PCPY_LINK)
        SHMIPS_EventDataZcpy = 0
    #endif /* if     !defined (MSGQ_COMPONENT)
                &&  defined (CHNL_ZCPY_LINK)
                &&  !defined (CHNL_PCPY_LINK) */

    #if     defined (MSGQ_COMPONENT)                 \
        &&  !defined (CHNL_ZCPY_LINK)                \
        &&  !defined (CHNL_PCPY_LINK)
        SHMIPS_EventMsg      = 0
    #endif /* if     defined (MSGQ_COMPONENT)
                &&  !defined (CHNL_ZCPY_LINK)
                &&  !defined (CHNL_PCPY_LINK) */

    #if     defined (MSGQ_COMPONENT)                 \
        &&  !defined (CHNL_ZCPY_LINK)                \
        &&  defined (CHNL_PCPY_LINK)
        SHMIPS_EventMsg      = 0,
        SHMIPS_EventDataPcpy = 1
    #endif /* if     defined (MSGQ_COMPONENT)
                &&  !defined (CHNL_ZCPY_LINK)
                &&  defined (CHNL_PCPY_LINK) */

    #if     defined (MSGQ_COMPONENT)                 \
        &&  defined (CHNL_ZCPY_LINK)                 \
        &&  !defined (CHNL_PCPY_LINK)
        SHMIPS_EventMsg      = 0,
        SHMIPS_EventDataZcpy = 1
    #endif /* if     defined (MSGQ_COMPONENT)
                &&  defined (CHNL_ZCPY_LINK)
                &&  !defined (CHNL_PCPY_LINK) */
    } SHMIPS_Event ;
```

**Constraints**

None.

**See Also**

SHMIPS_Register
SHMIPS_Send
SHMIPS_Notify

### 7.1.2 Typedefs & Data Structures

#### *7.1.2.1 FnShmIpsCbck*

This type defines the signature of the callback function to be registered with the SHMIPS component.

**Definition**

```
typedef Void (*FnShmIpsCbck) (IN OPT Pvoid arg, IN OPT Pvoid info) ;
```

**Comments**

The first parameter to this function is the event-specific argument passed to the SHMIPS component when registering the event.

The second parameter depends on the implementation of the SHMIPS component, and provides additional run-time information about the specific callback to the upper layers.

**DSP-side:**

The `typedef` definition is:

```
typedef Void (*SHMIPS_Cbck) (Ptr arg, Ptr info) ;
```

**Constraints**

None.

**See Also**

`ShmIpsObject`

*7.1.2.2 ShmIpsObject*

This structure defines the SHMIPS object, which contains all the component-specific information.

**Definition**

```
typedef struct ShmIpsObject_tag {
    Uint32              dspId ;
    Uint32              addrGppShmBase ;
    Uint32              addrDspShmBase ;
    FnShmIpsCbck        cbckFxn [MAX_SHMIPS_EVENTS] ;
    Pvoid               cbckArg [MAX_SHMIPS_EVENTS] ;
    InterruptObject     intObj ;
    IsrObject *         isrObj ;
    Uint32              dspMaduSize ;
    Bool                wordSwap ;
    Uint32              ptrCtrl ;
} ShmIpsObject ;
```

**Fields**

| | |
|---|---|
| `dspId` | ID of the DSP with which the SHM IPS communicates. |
| `addrGppShmBase` | Base address of the shared memory area reserved for use by the SHMIPS component. |
| `addrDspShmBase` | Base address of the shared memory area reserved for use by the SHMIPS component in DSP address space. |
| `cbckFxn` | Array of callback functions that can be registered with the SHMIPS component. One callback function is registered for each event supported by the SHMIPS component. |
| `cbckArg` | Array of arguments to the callback functions registered with the SHMIPS component. |
| `intObj` | Interrupt object used by the SHM IPS. |
| `isrObj` | ISR object used by the SHM IPS. |
| `dspMaduSize` | DSP Minimum Addressable Data Unit size. |
| `wordSwap` | Indicates whether word-swap is enabled for the DSP MMU. |
| `ptrCtrl` | Pointer to the SHMIPS control structure in shared memory. |

**Comments**

**DSP-side:**

The structure definition is:

```
typedef struct SHMIPS_Object_tag {
    SHMIPS_Cbck         cbckFxn   [MAX_SHMIPS_EVENTS] ;
    Ptr                 cbckArg   [MAX_SHMIPS_EVENTS] ;
    SHMIPS_ShmCtrl *    ptrCtrl ;
} SHMIPS_Object ;
```

**Constraints**

None.

**See Also**

FnShmIpsCbck
ShmIpsShmCtrl
SHMIPS_Register

#### *7.1.2.3 ShmIpsShmEventCtrl*

Defines the SHMIPS control structure for an event shared between the two processors.

**Definition**

```
typedef struct ShmIpsShmEventCtrl_tag {
    volatile List     toDspList ;
    volatile List     fmDspList ;
    volatile MpcsObj  csToDspList ;
    volatile MpcsObj  csFmDspList ;
} ShmIpsShmEventCtrl ;
```

**Fields**

| | |
|---|---|
| `toDspList` | Holds the list of buffers to be sent to the DSP. |
| `fmDspList` | Holds the list of buffers to be received from the DSP. |
| `csToDspList` | Shared critical section object for protection of operations by the two processors on the `toDspList`. |
| `csFmDspList` | Shared critical section object for protection of operations by the two processors on the `fmDspList`. |

**Comments**

**DSP-side:**

The structure definition is:

```
typedef struct SHMIPS_ShmEventCtrl_tag {
    volatile QUE_Elem  toDspList ;
    volatile QUE_Elem  fmDspList ;
    volatile MPCS_Obj  csToDspList ;
    volatile MPCS_Obj  csFmDspList ;
} SHMIPS_ShmEventCtrl ;
```

**Constraints**

None.

**See Also**

`ShmIpsShmCtrl`

#### *7.1.2.4 ShmIpsShmCtrl*

This structure defines the SHMIPS control structure shared between the two processors.

**Definition**

```
struct ShmIpsShmCtrl_tag {
    ShmIpsShmEventCtrl eventCtrl [MAX_SHMIPS_EVENTS] ;

#if defined (CHNL_COMPONENT)
    volatile List      freeChirps ;
    volatile MpcsObj   csFreeChirps ;
#endif /* if defined (CHNL_COMPONENT) */
} ;
```

**Fields**

| | |
|---|---|
| `eventCtrl` | Array of control structures for the events supported by the SHMIPS component. |
| `freeChirps` | Holds a free list of CHIRPs shared between the two processors. Defined only if CHNL component is included in the build config. |
| `csFreeChirps` | Shared critical section object for protection of operations by the two processors on the free CHIRP. Defined only if CHNL component is included in the build config. |

**Comments**

**DSP-side:**

The structure definition is:

```
struct SHMIPS_ShmCtrl_tag {
    SHMIPS_ShmEventCtrl eventCtrl [MAX_SHMIPS_EVENTS] ;

#if defined (CHNL_COMPONENT)
    volatile QUE_Elem  freeChirps ;
    volatile MPCS_Obj  csFreeChirps ;
#endif /* if defined (CHNL_COMPONENT) */
} ;
```

**Constraints**

None.

**See Also**

`ShmIpsShmEventCtrl`

### 7.1.3 API Definition

*7.1.3.1 SHMIPS_Initialize*

This function initializes the SHMIPS component.

**Syntax**

```
DSP_STATUS SHMIPS_Initialize (ProcessorId dspId, Uint32 ipsId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | DSP Identifier. | |
| IN | Uint32 | ipsId |
| | IPS Identifier. | |

**Return Value**

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

**Comments**

**GPP-side:**

This function performs the following initialization:

- Initialization of the global SHMIPS object.
- Initialization of the shared lists within the SHMIPS control structure after address translation to the DSP address space.
- Creation of the shared free CHIRP list by queuing up the CHIRPS within the reserved shared memory area onto the list in the SHMIPS control structure.
- Initialization of the shared critical section objects within the SHMIPS control structure.
- Creation, installation and enabling of the interrupt for communication with the DSP.

**DSP-side:**

The API definition is:

```
Void SHMIPS_init ()
```

This DSP-side function performs the following initialization:

- Initialization of the global SHMIPS object.
- Opening the shared critical section objects within the SHMIPS control structure.
- Registration of the interrupt service routine for communication with the GPP.

However, it does not initialize the shared lists within the SHMIPS control structure. These shared structures are initialized by the GPP-side SHMIPS component.

**Constraints**

The component must not be initialized.

**See Also**

`SHMIPS_Finalize`

*7.1.3.2 SHMIPS_Finalize*

This function finalizes the SHMIPS component.

**Syntax**

```
DSP_STATUS SHMIPS_Finalize (IN ProcessorId dspId, IN Uint32 ipsId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | DSP Identifier. | |
| IN | Uint32 | ipsId |
| | IPS Identifier. | |

**Return Value**

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

**Comments**

**GPP-side:**

This function performs the following finalization:

- Disabling, uninstall and deletion of the interrupt object used for communication with the DSP.
- Finalization of the shared lists within the SHMIPS control area
- Finalization of the shared critical section objects within the SHMIPS control area.
- Finalization of the global SHMIPS object.

**DSP-side:**

This function is not required within the DSP side implementation.

**Constraints**

The SHMIPS component must be initialized before calling this function.

**See Also**

SHMIPS_Initialize

*7.1.3.3 SHMIPS_Register*

This function registers a callback for a specific event with the SHMIPS component.

**Syntax**

```
DSP_STATUS SHMIPS_Register (ShmIpsEvent  event,
                           FnShmIpsCbck cbckFxn,
                           Pvoid        cbckArg) ;
```

**Arguments**

`IN` `ShmIpsEvent` `event`

Event to be registered.

`IN` `FnShmIpsCbck` `cbckFn`

Callback function to be registered for the specified event.

`IN OPT` `Pvoid` `cbckArg`

Optional argument to the callback function to be registered for the specified event. This argument shall be passed to each invocation of the callback function.

**Return Value**

| | |
|---|---|
| `DSP_SOK` | The event has been successfully registered. |
| `DSP_EFAIL` | General failure. |

**Comments**

**DSP-side:**

The API definition is:

```
Void SHMIPS_register (SHMIPS_Event  event,
                      SHMIPS_Cbck   cbckFn,
                      Ptr           cbckArg) ;
```

**Constraints**

The SHMIPS component must be initialized before calling this function.

The `cbckFxn` parameter must be valid.

The event must be supported by the SHMIPS component.

**See Also**

`ShmIpsEvent`
`FnShmIpsCbck`
`SHMIPS_Unregister ()`

*7.1.3.4 SHMIPS_Unregister*

This function unregisters a callback for a specific event with the SHMIPS component.

**Syntax**

```
DSP_STATUS SHMIPS_Unregister (ShmIpsEvent event) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ShmIpsEvent | event |

Event to be unregistered.

**Return Value**

| | |
|---|---|
| DSP_SOK | The event has been successfully unregistered. |
| DSP_EFAIL | General failure. |

**Comments**

**DSP-side:**

The API definition is:

```
Void SHMIPS_unregister (SHMIPS_Event event) ;
```

**Constraints**

The SHMIPS component must be initialized before calling this function.

The event must be supported by the SHMIPS component.

The event must have been registered with the SHMIPS component earlier.

**See Also**

```
ShmIpsEvent
SHMIPS_Register ()
```

*7.1.3.5 SHMIPS_Send*

This function sends an event to the DSP.

**Syntax**

```
DSP_STATUS SHMIPS_Send (ShmIpsEvent event, Pvoid bufPtr) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ShmIpsEvent | event |

Event to be sent to the DSP.

| | | |
|---|---|---|
| IN | Pvoid | bufPtr |

Event-specific argument.

**Return Value**

| | |
|---|---|
| DSP_SOK | The event has been successfully sent. |
| DSP_EFAIL | General failure. |

**Comments**

In case of a data transfer event, the argument is a pointer to the CHIRP for the buffer to be sent to the other processor.

In case of a message transfer event, the argument is a pointer to the message to be sent to the other processor.

**DSP-side:**

The API definition is:

```
Void SHMIPS_send (SHMIPS_Event event, Ptr bufPtr) ;
```

**Constraints**

The SHMIPS component must be initialized before calling this function.

The event must be supported by the SHMIPS component.

The buffer pointer parameter must be valid.

**See Also**

ShmIpsEvent

*7.1.3.6 SHMIPS_Notify*

This function sends a notification of an event to the DSP.

**Syntax**

```
DSP_STATUS SHMIPS_Notify (ShmIpsEvent event) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ShmIpsEvent | event |

Event to be notified to the DSP.

**Return Value**

| | |
|---|---|
| DSP_SOK | Operation successfully completed. |
| DSP_ETIMEOUT | Timed out while sending interrupt to the DSP. |
| DSP_EFAIL | General failure. |

**Comments**

In case of an interrupt-based protocol, this function sends an interrupt to the DSP along with information about the event to be notified.

**DSP-side:**

The API definition is:

```
Void SHMIPS_notify (SHMIPS_Event event) ;
```

**Constraints**

The SHMIPS component must be initialized before calling this function.

The event must be supported by the SHMIPS component.

**See Also**

ShmIpsEvent

#### *7.1.3.7 SHMIPS_Debug*

This function prints the current status of the SHMIPS subcomponent.

**Syntax**

```
Void SHMIPS_Debug () ;
```

**Arguments**

None.

**Return Value**

None.

**Comments**

This function is defined only on the GPP-side if debugging is enabled.

**Constraints**

None.

**See Also**

None

# 8 SHMDRV

The SHMDRV component has a similar design on both the GPP and DSP sides. This section primarily refers to the GPP side design. However, the DSP-side design shall contain the same enumerations, structures, and API definitions, with minimal changes for different naming conventions on the GPP and DSP-sides.

## 8.1 GPP and DSP side low level design

### 8.1.1 Constants & Enumerations

#### *8.1.1.1 SHMDRV_CTRL_SIZE*

This constant defines the shared memory control structure size required by the SHMDRV component.

**Definition**

```
#define SHMDRV_CTRL_SIZE      (sizeof (ShmDrvControl))
```

**Comments**

This constant is used by configuration script for assigning shared memory regions to the various components using the same memory area.

**DSP-side:**

The constant definition is:

```
#define SHMDRV_CTRL_SIZE      (sizeof (SHMDRV_ShmControl))
```

**Constraints**

None.

**See Also**

ShmDrvControl

*8.1.1.2 GPP_HANDSHAKE*

Handshake value written by GPP

**Definition**

```
#define GPP_HANDSHAKE 0xC0C0
```

**Comments**

None.

**Constraints**

None.

**See Also**

DSP_HANDSHAKE
SHMDRV_Handshake ()

*8.1.1.3 DSP_HANDSHAKE*

Handshake value written by the DSP

**Definition**

```
#define DSP_HANDSHAKE 0xBAB0
```

**Comments**

The lowermost nibble in the DSP handshake value is reserved for configuration information supplied by the DSP-side SHMDRV component along with the handshake. This value is compared with the ARM-side configuration information to verify that the drivers on both processors have been built with the same configuration.

**Constraints**

None.

**See Also**

```
GPP_HANDSHAKE
SHMDRV_Handshake ()
```

### 8.1.2 Typedefs & Data Structures

#### *8.1.2.1 ShmDrvControl*

This structure defines the control structure used by GPP and DSP for SHM Link driver

**Definition**

```
typedef struct ShmDrvControl_tag {
    volatile Uint16  handshakeGpp   ;
    volatile Uint16  handshakeDsp   ;
} ShmDrvControl ;
```

**Fields**

| | |
|---|---|
| `handshakeGpp` | Handshake field to be updated by GPP. |
| `handshakeDsp` | Handshake field to be updated by DSP. |

**Comments**

**DSP-side:**

The structure definition is:

```
typedef struct SHMDRV_ShmControl_tag {
    volatile Uint16  handshakeGpp ;
    volatile Uint16  handshakeDsp ;
} SHMDRV_ShmControl ;
```

**Constraints**

None.

**See Also**

`SHMDRV_Handshake ()`

*8.1.2.2 ShmDrvObject*

This structure defines the SHM link driver object, which contains all the component-specific information.

**Definition**

```
typedef struct ShmDrvObject_tag {
    ShmDrvControl *  ptrCtrl ;
} ShmDrvObject ;
```

**Fields**

| | |
|---|---|
| `ptrCtrl` | Pointer to the SHM Driver control structure in shared memory. |

**Comments**

**DSP-side:**

The structure definition is:

```
typedef struct SHMDRV_Object_tag {
    Uint32  ptrCtrl ;
    Bool    isSync ;
} SHMDRV_Object ;
```

The second field indicates whether the driver has already been synchronized. This is required since there may be multiple invocations of the handshaking function, but the handshaking must be performed only once.

**Constraints**

None.

**See Also**

`SHMDRV_Initialize ()`

### 8.1.3 API Definition

The SHMDRV APIs are exposed to LDRV_DRV through a function table:

```
LinkInterface SHMDRV_Interface = {
    &SHMDRV_Initialize,
    &SHMDRV_Finalize,
    &SHMDRV_Handshake
#if defined (DDSP_DEBUG)
    ,&SHMDRV_Debug
#endif /* if defined (DDSP_DEBUG) */
} ;
```

On the DSP-side, the functions are directly exposed through APIs, and not through a function pointer interface.

#### *8.1.3.1 SHMDRV_Initialize*

This function initializes the SHMDRV component.

**Syntax**

```
DSP_STATUS SHMDRV_Initialize (IN ProcessorId dspId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |

Processor Identifier.

**Return Value**

| | |
|---|---|
| DSP_SOK | This component has been successfully initialized. |
| DSP_EMEMORY | Failure during memory operation. |
| DSP_EFAIL | General failure. |

**Comments**

**GPP-side:**

This function performs the following initialization:

- Initialization of the global SHMDRV object.
- Initialization of the shared memory control area for the SHMDRV component.
- Initialization of the IPS for the driver.

**DSP-side:**

The API definition is:

```
Void SHMDRV_init  ()
```

This function performs the following initialization:

- Initialization of the global SHMDRV object.
- Initialization of the *DSPLINK* Interrupt component.
- Initialization of the IPS for the driver.

**Constraints**

`dspId` must be valid.

**See Also**

`SHMDRV_Finalize ()`

#### *8.1.3.2 SHMDRV_Finalize*

This function finalizes the SHMDRV component.

**Syntax**

```
DSP_STATUS SHMDRV_Finalize (IN ProcessorId dspId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |

Processor Identifier.

**Return Value**

| | |
|---|---|
| DSP_SOK | This component has been successfully finalized. |
| DSP_EMEMORY | Failure during memory operation. |
| DSP_EFAIL | General failure. |

**Comments**

**GPP-side:**

This function performs the following initialization:

- Finalization of the IPS for the driver.
- Finalization of the shared memory control area for the SHMDRV component.
- Finalization of the global SHMDRV object.

**DSP-side:**

There is no finalization function for the DSP-side.

**Constraints**

`dspId` must be valid.

**See Also**

`SHMDRV_Initialize`

*8.1.3.3 SHMDRV_Handshake*

This function performs the necessary handshake between the drivers on the GPP & DSP.

**Syntax**

```
DSP_STATUS SHMDRV_Handshake (IN ProcessorId    dspId,
                             IN DrvHandshake   hshkCtrl) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |

Processor Identifier.

| | | |
|---|---|---|
| IN | DrvHandshake | hshkCtrl |

Handshake control action to be executed.

**Return Value**

| | |
|---|---|
| DSP_SOK | This operation has been successfully completed. |
| DSP_EFAIL | Operation failed. |

**Comments**

**DSP-side:**

The API definition is:

```
Void SHMDRV_handshake  ()
```

The API on the DSP-side completes handshaking with the GPP-side driver. It does not provide separate actions for setup, start and completion of handshake. It ensures that even if the function is called multiple times, handshaking is only performed once.

**Constraints**

`dspId` must be valid.

**See Also**

`ShmDrvControl`

#### *8.1.3.4 SHMDRV_Debug*

This function prints the current status of the SHMDRV subcomponent.

**Syntax**

```
Void SHMDRV_Debug (IN ProcessorId dspId) ;
```

**Arguments**

`IN` `ProcessorId` `dspId`

Processor Identifier.

**Return Value**

None.

**Comments**

This function is defined only on the GPP-side if debugging is enabled.

**Constraints**

`dspId` must be valid.

**See Also**

`None.`

# 9 ZCPY MQT

The ZCPY MQT component on a processor is responsible for implementing the message transfer protocol for the remote transport. The messages transferred by this MQT shall not require an intermediate copy into shared memory.

This section provides a detailed design for the specific implementation of the zero-copy MQT. The ZCPY MQT complies with the interface defined by the generic messaging component on both the GPP and DSP sides. For details about the generic messaging component and MQT designs, please refer to the messaging design document [Ref. 4].

The ZCPY MQT component has a similar design on both the GPP and DSP sides.

## 9.1 GPP side low level design

### 9.1.1 Constants & Enumerations

#### *9.1.1.1 ZCPYMQT_CTRLMSG_SIZE*

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

**Definition**

```
#define ZCPYMQT_CTRLMSG_SIZE 128
```

**Comments**

This constant is available to the user at the API level.

The ZCPY MQT uses the default pool for allocating control messages required for communication with other processors. The number of control messages required depends on the frequency of usage of APIs requiring control messages, such as `MSGQ_Locate ()`. The user must consider this requirement while configuring the default pool.

The value of this constant is platform-specific, and it is therefore defined within the `platform.h` header file.

**Constraints**

The user must always use this constant when configuring the default pool for the ZCPY MQT. The user must not hard-code the size within the application. This allows future compatibility with later versions of the ZCPY 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.

If applicable for the platform, the size of the ZCPY MQT control message must be aligned to the data cache boundary of the DSP by being equal to or a multiple of the cache line size.

**See Also**

`ZcpyMqtCtrlMsg`

#### *9.1.1.2 ZCPYMQT_CTRLCMD_LOCATE*

This macro defines the control command message ID for location of a remote MSGQ.

**Definition**

```
#define ZCPYMQT_CTRLCMD_LOCATE     MSGQ_MQTMSGIDSSTART
```

**Comments**

This ID is used as the message ID within the message header, when the message is an MQT control message indicating a request for location of a remote Message Queue. A control message has the destination ID as `MSGQ_INVALIDMSGQ`, 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. The actual message content differs depending on the control command.

**Constraints**

The value of this control message ID must lie within the range defined by the MSGQ component for MQTs: `MSGQ_MQTMSGIDSSTART` to `MSGQ_MQTMSGIDSEND`.

**See Also**

`ZCPYMQT_CTRLCMD_LOCATEACK`
`ZcpyMqtCtrlMsg`
`ZCPYMQT_Locate ()`

#### *9.1.1.3 ZCPYMQT_CTRLCMD_LOCATEACK*

This macro defines the control command message ID for acknowledgement for location of a remote MSGQ..

**Definition**

```
#define ZCPYMQT_CTRLCMD_LOCATEACK (MSGQ_MQTMSGIDSSTART + 1)
```

**Comments**

This ID is used as the message ID within the message header, when the message is an MQT control message indicating an acknowledgement of a request for location of a remote Message Queue. A control message has the destination ID as `MSGQ_INVALIDMSGQ`, 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. The actual message content differs depending on the control command.

**Constraints**

The value of this control message ID must lie within the range defined by the MSGQ component for MQTs: `MSGQ_MQTMSGIDSSTART` to `MSGQ_MQTMSGIDSEND`.

**See Also**

```
ZCPYMQT_CTRLCMD_LOCATE
ZcpyMqtCtrlMsg
ZCPYMQT_Locate ()
```

### 9.1.2 Typedefs & Data Structures

#### *9.1.2.1 ZcpyMqtAttrs*

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

**Definition**

```
typedef struct ZcpyMqtAttrs_tag {
    PoolId    poolId ;
} ZcpyMqtAttrs ;
```

**Fields**

| | |
|---|---|
| `poolId` | Pool ID used for allocating control messages. This pool is also used in case the ID within the message received from the DSP is invalid. This can occur in case of a mismatch between pools configured on the GPP and the DSP. |

**Comments**

This structure is available to the user at the API level.

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

**Constraints**

None.

**See Also**

`ZCPYMQT_Open ()`

#### *9.1.2.2 ZcpyMqtState*

This structure defines the ZCPYMQT state object, which contains all the component-specific information.

**Definition**

```
typedef struct ZcpyMqtState_tag {
    PoolId          poolId ;
    List *          msgList ;
    List *          ackMsgList ;
    DpcObject *     dpcObj ;
    Bool            wordSwap ;
} ZcpyMqtState ;
```

**Fields**

| | |
|---|---|
| `poolId` | The default Pool to be used by the ZCPY MQT. |
| `msgList` | List of messages received from the DSP. |
| `ackMsgList` | List of `locateAck` messages received from the DSP. |
| `dpcObj` | DPC object used by the ZCPY MQT. |
| `wordSwap` | Indicates whether word-swap is enabled for the DSP MMU. |

**Comments**

An instance of this object is created and initialized during `ZCPYMQT_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**

`ZCPYMQT_Open ()`

*9.1.2.3 ZcpyMqtCtrlMsg*

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

**Definition**

```
typedef struct ZcpyMqtCtrlMsg_tag {
    MsgqMsgHeader msgHeader ;
    union {
        struct {
            Uint32      semHandle ;
            Uint32      replyQueue;
            Uint32      arg;
            Uint16      poolId;
            Uint16      padding;
            Uint16      msgqName [DSP_MAX_STRLEN] ;
       } locateMsg ;

        struct {
            Uint32      semHandle ;
            Uint32      replyQueue;
            Uint32      arg;
            Uint16      poolId;
            Uint16      padding;
            Uint32      msgqQueue ;
        } locateAckMsg ;
    } ctrlMsg ;
} ZcpyMqtCtrlMsg ;
```

**Fields**

`msgHeader` Fixed message header required for all messages.

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

```
locateMsg:
    semHandle   -> Semaphore handle for sync locate
    replyQueue  -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    poolId      -> Pool ID to allocate async
                   response messages
    padding     -> Padding for alignment.
    msgqName    -> Name of the MSGQ to be located on
                   the remote processor.
locateAckMsg:
    semHandle   -> Semaphore handle for sync locate
    replyQueue  -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    poolId      -> Pool ID to allocate async
                   response messages
    padding     -> Padding for alignment.
    msgqQueue   -> Handle to the MSGQ located on the
                   remote processor.
```

**Comments**

The control messages are used for communication between the MQTs.

**Constraints**

None.

**See Also**

`ZcpyMqtCtrlCmd`

### 9.1.3 API Definition

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

```
MqtInterface ZCPYMQT_Interface = {
    &ZCPYMQT_Initialize,
    &ZCPYMQT_Finalize,
    &ZCPYMQT_Open,
    &ZCPYMQT_Close,
    &ZCPYMQT_Locate,
    &ZCPYMQT_Release,
    &ZCPYMQT_Put
#if defined (DDSP_DEBUG)
    ,&ZCPYMQT_Debug
#endif /* defined (DDSP_DEBUG) */
} ;
```

#### *9.1.3.1 ZCPYMQT_Initialize*

This function performs global initialization of the ZCPY MQT.

**Syntax**

```
Void ZCPYMQT_Initialize () ;
```

**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 ZCPY MQT.

**Constraints**

None.

**See Also**

`ZCPYMQT_Finalize ()`

*9.1.3.2 ZCPYMQT_Finalize*

This function performs global finalization of the ZCPY MQT.

**Syntax**

```
Void ZCPYMQT_Finalize () ;
```

**Arguments**

None.

**Return Value**

None.

**Comments**

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

**Constraints**

None.

**See Also**

`ZCPYMQT_Initialize ()`

#### *9.1.3.3 ZCPYMQT_Open*

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

**Syntax**

```
DSP_STATUS ZCPYMQT_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 ZCPY MQT has been successfully opened. |
| `DSP_EMEMORY` | Operation failed due to a memory error. |
| `DSP_EFAIL` | General failure |

**Comments**

This function is called during `MSGQ_TransportOpen ()`, when the processor ID passed uses the ZCPY 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 `ZcpyMqtState`, and returns it to the `LDRV MSGQ` component.

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

**Constraints**

`mqtHandle` must be valid.

`mqtAttrs` must be valid.

**See Also**

`ZcpyMqtState`
`ZcpyMqtAttrs`
`ZCPYMQT_Close ()`

*9.1.3.4 ZCPYMQT_Close*

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

**Syntax**

```
DSP_STATUS ZCPYMQT_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_EFAIL` | General failure. |

**Comments**

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

**Constraints**

`mqtHandle` must be valid.

**See Also**

```
ZcpyMqtState
ZCPYMQT_Open ()
```

*9.1.3.5 ZCPYMQT_Locate*

This function attempts to locate a message queue present on the remote processor. The message queue to be located is identified by its system-wide unique name.

**Syntax**

```
Void ZCPYMQT_Locate (LdrvMsgqTransportHandle mqtHandle,
                     Pstr                    queueName,
                     Bool                    sync,
                     MsgqQueue *             msgqQueue,
                     Pvoid                   locateAttrs) ;
```

**Arguments**

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the LDRV MSGQ transport object.

`IN` `Pstr` `queueName`

Name of the message queue to be located.

`IN` `Bool` `sync`

Indicates whether the location is synchronous.

`IN OUT` `MsgqQueue *` `msgqQueue`

If synchronous: indicates the location to store the handle to the located message queue.

If asynchronous: indicates the message queue to be used to receive the response message for location.

`IN` `Pvoid` `locateAttrs`

If synchronous: indicates the attributes for synchronous location of the MSGQ.

If asynchronous: indicates the attributes for asynchronous location of the MSGQ.

**Return Value**

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

**Comments**

This API is called to locate a message queue on the processor to which the ZCPY MQT connects. The message queue is identified through its system-wide unique name. Before sending a message to the remote MSGQ, a handle to the message queue must be obtained by calling this API. After the message queue has been successfully located, the message queue handle can be used for further actions on the MSGQ, including sending messages to it.

The caller specifies whether the location must be synchronous or asynchronous.

Synchronous: When called synchronously, the `msgqQueue` parameter is used for returning the located MSGQ. The API blocks until the remote MSGQ has been located.

Asynchronous: When called asynchronously, 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 `MsgqAsyncLocateMsg` message, and sends it to the reply MSGQ specified by the user.

This function allocates a control message, fills its fields with the information about the message queue to be located and sends it to the SHMIPS. The SHMIPS sends this control message to its DSP-side counterpart, which forwards it on to the ZCPY MQT on the DSP. The DSP-side ZCPY MQT attempts to locate the message queue locally, and sends the corresponding information back to the DSP through a `locateAck` message in the same way.

**Constraints**

`mqtHandle` must be valid.

`queueName` must be valid.

`msgqQueue` must be valid.

**See Also**

`ZCPYMQT_Release ()`

*9.1.3.6 ZCPYMQT_Release*

This function releases the remote MSGQ located earlier.

**Syntax**

```
DSP_STATUS ZCPYMQT_Release (LdrvMsgqTransportHandle mqtHandle,
                            MsgqQueue               msgqQueue) ;
```

**Arguments**

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MsgqQueue` `msgqQueue`

Handle to the message queue to be released.

**Return Value**

| | |
|---|---|
| `DSP_SOK` | The message queue has been successfully released. |
| `DSP_EMEMORY` | Operation failed due to a memory error. |
| `DSP_EFAIL` | General failure |

**Comments**

This function is called during `LDRV_MSGQ_Release ()` if the message queue to be released is on the remote processor connected by the ZCPY MQT.

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

**Constraints**

`mqtHandle` must be valid.

`msgqQueue` must be valid.

**See Also**

`ZCPYMQT_Locate ()`

*9.1.3.7 ZCPYMQT_Put*

This function sends a message to the specified remote MSGQ.

**Syntax**

```
DSP_STATUS ZCPYMQT_Put (LdrvMsgqTransportHandle mqtHandle,
                        MsgqMsg                 msg) ;
```

**Arguments**

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`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_EFAIL` | General failure. |

**Comments**

This function is called during `LDRV_MSGQ_Put ()` if the destination message queue is on the remote processor connected by the ZCPY MQT.

This function sends a message transfer event to the `SHMIPS` component.

**Constraints**

`mqtHandle` must be valid.

`msg` must be valid.

**See Also**

`SHMIPS_Send`

*9.1.3.8 ZCPYMQT_Debug*

This function prints debug information about the MQT.

**Syntax**

```
Void ZCPYMQT_Debug (LdrvMsgqTransportHandle mqtHandle) ;
```

**Arguments**

`IN` `LdrvMsgqTransportHandle` `mqtHandle`

Handle to the message queue.

**Return Value**

None.

**Comments**

None.

**Constraints**

`mqtHandle` must be valid.

This function is defined only for debug builds.

**See Also**

None.

*9.1.3.9 ZCPYMQT_Callback*

This function implements the callback invoked by the SHMIPS component on receiving a message from the remote processor.

**Syntax**

```
Void ZCPYMQT_Callback (Pvoid arg, Pvoid info) ;
```

**Arguments**

`IN` `Pvoid` `arg`

Argument registered with the SHMIPS component along with the callback function.

`IN` `Pvoid` `info`

Pointer to message received in the event.

**Return Value**

None.

**Comments**

This callback function is registered with the SHMIPS component to receive intimation about a received message. The SHMIPS component invokes this function when it receives the corresponding event.

This function is not part of the standard MQT interface expected by the LDRV_MSGQ component.

**Constraints**

`info` must be a valid pointer.

**See Also**

`SHMIPS_Register ()`

## 9.2 DSP side low level design

### 9.2.1 Constants & Enumerations

#### *9.2.1.1 ZCPYMQT_CTRLMSG_SIZE*

This constant defines the size (in MADUs) of control messages used between the ZCPY MQTs on the two processors.

**Definition**

```
#define ZCPYMQT_CTRLMSG_SIZE 128
```

**Comments**

This constant is available to the user at the API level.

The ZCPY MQT uses the default pool for allocating control messages required for communication with other processors. The number of control messages required depends on the frequency of usage of APIs requiring control messages, such as `MSGQ_Locate ()`. The user must consider this requirement while configuring the default pool.

The value of this constant is platform-specific, and it is therefore defined within the `platform.h` header file.

**Constraints**

The user must always use this constant when configuring the default pool for the remote MQT. The user must not hard-code the size within the application. This allows future compatibility with later versions of the ZCPY 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.

If applicable for the platform, the size of the ZCPY MQT control message must be aligned to the data cache boundary of the DSP by being equal to or a multiple of the cache line size.

**See Also**

`ZCPYMQT_CtrlMsg`

#### *9.2.1.2 ZCPYMQT_CTRLCMD_LOCATE*

This macro defines the control command message ID for location of a remote MSGQ.

**Definition**

```
#define ZCPYMQT_CTRLCMD_LOCATE     MSGQ_MQTMSGIDSSTART
```

**Comments**

This ID is used as the message ID within the message header, when the message is an MQT control message indicating a request for location of a remote Message Queue. A control message has the destination ID as `MSGQ_INVALIDMSGQ`, 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. The actual message content differs depending on the control command.

**Constraints**

The value of this control message ID must lie within the range defined by the MSGQ component for MQTs: `MSGQ_MQTMSGIDSSTART` to `MSGQ_MQTMSGIDSEND`.

**See Also**

```
ZCPYMQT_CTRLCMD_LOCATEACK
ZCPYMQT_CtrlMsg
ZCPYMQT_locate ()
```

#### *9.2.1.3 ZCPYMQT_CTRLCMD_LOCATEACK*

This macro defines the control command message ID for acknowledgement for location of a remote MSGQ..

**Definition**

```
#define ZCPYMQT_CTRLCMD_LOCATEACK (MSGQ_MQTMSGIDSSTART + 1)
```

**Comments**

This ID is used as the message ID within the message header, when the message is an MQT control message indicating an acknowledgement of a request for location of a remote Message Queue. A control message has the destination ID as `MSGQ_INVALIDMSGQ`, 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. The actual message content differs depending on the control command.

**Constraints**

The value of this control message ID must lie within the range defined by the MSGQ component for MQTs: `MSGQ_MQTMSGIDSSTART` to `MSGQ_MQTMSGIDSEND`.

**See Also**

```
ZCPYMQT_CTRLCMD_LOCATE
ZCPYMQT_CtrlMsg
ZCPYMQT_locate ()
```

### 9.2.2 Typedefs & Data Structures

#### *9.2.2.1 ZCPYMQT_Params*

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

**Definition**

```
typedef struct ZCPYMQT_Params_tag {
    Uint16  poolId ;
} ZCPYMQT_Params ;
```

**Fields**

| | |
|---|---|
| `poolId` | Pool ID used for allocating control messages. This pool is also used in case the ID within the message received from the DSP is invalid. This can occur in case of a mismatch between pools configured on the GPP and the DSP. |

**Comments**

This structure is available to the user at the API level.

These parameters are provided to the MQT once during its initialization, which takes place during the initialization of the MSGQ component.

The default parameters to be used in case user does not provide any attributes to the MQT are defined as:

```
static ZCPYMQT_Params ZCPYMQT_PARAMS = {0};
```

**Constraints**

None.

**See Also**

`ZCPYMQT_open ()`

#### *9.2.2.2 ZCPYMQT_State*

This structure defines the ZCPYMQT state object, which contains all the component-specific information.

**Definition**

```
typedef struct ZCPYMQT_State_tag {
    Uint16      poolId ;
    QUE_Obj     ackMsgQueue ;
#if defined (USE_SWI)
    QUE_Obj     msgQueue ;
    SWI_Handle  swiHandle ;
#endif /* if defined (USE_SWI) */
} ZCPYMQT_State ;
```

**Fields**

| | |
|---|---|
| `poolId` | Pool ID used for allocating control messages. This pool is also used in case the ID within the message received from the DSP is invalid. This can occur in case of a mismatch between pools configured on the GPP and the DSP. |
| `ackMsgQueue` | Queue of `locateAck` messages received from the GPP. |
| `msgQueue` | Queue of messages received from the GPP.<br><br>Only defined if callback processing is to be performed within a SWI instead of interrupt context. |
| `swiHandle` | SWI for processing of locate functionality in non-ISR context.<br><br>Only defined if callback processing is to be performed within a SWI instead of interrupt context. |

**Comments**

An instance of this object is created and initialized during `ZCPYMQT_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**

`ZCPYMQT_open ()`

*9.2.2.3 ZCPYMQT_CtrlMsg*

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

**Definition**

```
typedef struct ZCPYMQT_CtrlMsg_tag {
    MSGQ_MsgHeader msgHeader ;
    union {
        struct {
            Uint32      semHandle ;
            Uint32      replyQueue ;
            Uint32      arg ;
            Uint16      poolId ;
            Uint16      padding ;
            Uint16      msgqName [DSP_MAX_STRLEN] ;
        } locateMsg ;

        struct {
            Uint32      semHandle ;
            Uint32      replyQueue ;
            Uint32      arg ;
            Uint16      poolId ;
            Uint16      padding ;
            Uint32      msgqQueue ;
        } locateAckMsg ;
    } ctrlMsg ;
} ZCPYMQT_CtrlMsg ;
```

**Fields**

`msgHeader` Fixed message header required for all messages.

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

```
locateMsg:
    semHandle   -> Semaphore handle for sync locate
    replyQueue  -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    poolId      -> Pool ID to allocate async
                   response messages
    padding     -> Padding for alignment.
    msgqName    -> Name of the MSGQ to be located on
                   the remote processor.
locateAckMsg:
    semHandle   -> Semaphore handle for sync locate
    replyQueue  -> Reply MSGQ handle for async
                   locate
    arg         -> User-defined value passed to
                   locate
    poolId      -> Pool ID to allocate async
                   response messages
    padding     -> Padding for alignment.
    msgqQueue   -> Handle to the MSGQ located on the
                   remote processor.
```

**Comments**

The control messages are used for communication between the MQTs.

**Constraints**

None.

**See Also**

`ZCPYMQT_CtrlCmd`

### 9.2.3 API Definition

The MQT APIs are exposed to MSGQ through the following function table:

```
MSGQ_TransportFxns ZCPYMQT_FXNS = {
    &ZCPYMQT_open,
    &ZCPYMQT_close,
    &ZCPYMQT_locate,
    &ZCPYMQT_release,
    &ZCPYMQT_put
} ;
```

#### *9.2.3.1 ZCPYMQT_init*

This function performs global initialization of the ZCPY MQT.

**Syntax**

```
Void ZCPYMQT_init () ;
```

**Arguments**

None.

**Return Value**

None.

**Comments**

The DSP/BIOS™ OS calls the `mqtInit ()` function during its boot-up process.

This function initializes the SHMDRV component and waits for handshake completion with the GPP-side. In case messaging and data transfer are both enabled, the module getting initialized first ensures that the SHMDRV component is initialized, and also completes handshake with the GPP.

**Constraints**

None.

**See Also**

None.

*9.2.3.2 ZCPYMQT_open*

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

**Syntax**

```
Int ZCPYMQT_open (MSGQ_TransportHandle mqtHandle) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | MSGQ_TransportHandle | mqtHandle |

Handle to the MSGQ transport object.

**Return Value**

| | |
|---|---|
| SYS_OK | This component has been successfully opened. |
| SYS_EALLOC | Failure during memory operation. |

**Comments**

This API is called during the initialization of the MSGQ component. 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 ZCPYMQT_State, and sets it in the MSGQ transport object.

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

**Constraints**

This function cannot be called from SWI or HWI context.

The handle to the MSGQ transport object must be valid.

**See Also**

```
ZCPYMQT_State
ZCPYMQT_Params
ZCPYMQT_close ()
```

#### *9.2.3.3 ZCPYMQT_close*

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

**Syntax**

```
Int ZCPYMQT_close (MSGQ_TransportHandle mqtHandle) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | MSGQ_TransportHandle | mqtHandle |

Handle to the MSGQ transport object.

**Return Value**

| | |
|---|---|
| SYS_OK | This component has been successfully closed. |
| SYS_EFREE | Failure during memory operation. |

**Comments**

This API is called during the finalization of the MSGQ component. It carries out any required actions for finalizing the MQT.

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

**Constraints**

This function cannot be called from SWI or HWI context.

The handle to the MSGQ transport object must be valid.

**See Also**

```
ZCPYMQT_State
ZCPYMQT_open ()
```

*9.2.3.4 ZCPYMQT_locate*

This function attempts to locate a message queue present on the remote processor. The message queue to be located is identified by its system-wide unique name.

**Syntax**

```
Int ZCPYMQT_locate (MSGQ_TransportHandle mqtHandle,
                    String               queueName,
                    Bool                 sync,
                    MSGQ_Queue *         msgqQueue,
                    Ptr                  locateAttrs) ;
```

**Arguments**

`IN` `MSGQ_TransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `String` `queueName`

Name of the MSGQ to be located.

`IN` `Bool` `sync`

Indicates whether locate is synchronous or asynchronous.

`IN OUT` `MSGQ_Queue *` `msgqQueue`

If synchronous: indicates the location to store the handle to the located message queue.

If asynchronous: indicates the message queue to be used to receive the response message for location.

`IN` `Ptr` `locateAttrs`

If synchronous: indicates the attributes for synchronous location of the MSGQ.

If asynchronous: indicates the attributes for asynchronous location of the MSGQ.

**Return Value**

| | |
|---|---|
| `SYS_OK` | The message queue has been successfully located. |
| `SYS_ENOTFOUND` | The message queue does not exist on the remote processor. |
| `SYS_ETIMEOUT` | Timeout during location of the MSGQ. |
| `SYS_EALLOC` | Failure during memory operation. |

**Comments**

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

The caller specifies whether the location must be synchronous or asynchronous.

Synchronous: When called synchronously, the `msgqQueue` parameter is used for returning the located MSGQ. The API blocks until the remote MSGQ has been located.

Asynchronous: When called asynchronously, 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 `MSGQ_AsyncLocateMsg` message, and sends it to the reply MSGQ specified by the user.

This function allocates a control message, fills its fields with the information about the message queue to be located and sends it to the SHMIPS. The SHMIPS sends this control message to its GPP-side counterpart, which forwards it on to the ZCPY MQT on the GPP. The GPP-side ZCPY MQT attempts to locate the message queue locally, and sends the corresponding information back to the DSP through a `locateAck` message in the same way.

**Constraints**

The default pool 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.

The handle to the MSGQ transport object must be valid.

The `queueName` must be valid.

The `locateAttrs` must be valid.

**See Also**

`ZCPYMQT_release ()`

*9.2.3.5 ZCPYMQT_release*

This function releases the remote MSGQ located earlier.

**Syntax**

```
Int ZCPYMQT_release (MSGQ_TransportHandle mqtHandle,
                     MSGQ_Queue           msgqQueue) ;
```

**Arguments**

`IN` `MSGQ_TransportHandle` `mqtHandle`

Handle to the MSGQ transport object.

`IN` `MSGQ_Queue` `msgqQueue`

Handle to the MSGQ to be released.

**Return Value**

| | |
|---|---|
| `SYS_OK` | The message queue has been successfully released. |

**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**

The handle to the MSGQ transport object must be valid.

The handle to the message queue must be valid.

**See Also**

`ZCPYMQT_locate ()`

#### *9.2.3.6 ZCPYMQT_put*

This function sends a message to the specified remote MSGQ.

**Syntax**

```
Int ZCPYMQT_put (MSGQ_TransportHandle mqtHandle, MSGQ_Msg msg) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | MSGQ_TransportHandle | mqtHandle |

Handle to the MSGQ transport object.

| | | |
|---|---|---|
| IN | MSGQ_Msg | msg |

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

**Return Value**

| | |
|---|---|
| SYS_OK | The message has been successfully sent. |

**Comments**

This function is called during MSGQ_put () if the destination message queue is on the remote processor connected by the ZCPY MQT.

This function sends a message transfer event to the SHMIPS component.

This function is non-blocking and deterministic.

**Constraints**

The handle to the MSGQ transport object must be valid.

The pointer to the message must be valid.

**See Also**

ZCPYMQT_callback ()
SHMIPS_send ()

*9.2.3.7 ZCPYMQT_callback*

This function implements the callback invoked by the SHMIPS component on receiving a message from the remote processor.

**Syntax**

```
Void ZCPYMQT_callback (Ptr arg, Ptr info) ;
```

**Arguments**

`IN` `Ptr` `arg`

Argument registered with the SHMIPS component along with the callback function.

`IN` `Ptr` `info`

Pointer to message received in the event.

**Return Value**

None.

**Comments**

This callback function is registered with the SHMIPS component to receive intimation about a received message. The SHMIPS component invokes this function when it receives the corresponding event.

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

**Constraints**

None.

**See Also**

`SHMIPS_register ()`