# DSP/BIOS™ LINK

# LINK DRIVER

# LNK 012 DES

# Version 1.20

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 and Scope

This document describes the design of the Link Driver component of DSP/BIOS™ LINK. It defines the main functions and data structures used in the implementation of the Link Driver component. It also describes how each function is implemented.

This document is intended for developers implementing the Link Driver component of DSP/BIOS™ LINK. Developers implementing new link driver(s) can also use it as a reference.

## 1.2 Terms and Abbreviations

| | |
|---|---|
| *DSPLINK* | DSP/BIOS™ LINK |
| ARM | Advanced RISC Machines (ARM Ltd’s RISC Processor) |
| CHIRP | Channel IO Request Packets |
| DSP/BIOS™ | Built-In Operating System for DSP (TI’s proprietary OS) |
| GPP | General Purpose Processor |
| Link Driver/LDRV | Link Driver component of DSP/BIOS™ Link. |
| PMGR | Processor Manager component of DSP/BIOS™ Link. |
| SHM | Shared Memory Driver |

## 1.3 References

| | | |
|---|---|---|
| 1. | LNK 031 DES | DSP/BIOS™ LINK<br>Messaging Component<br>Version 1.11, dated AUG 09, 2004 |
| 2. | LNK 082 DES | DSP/BIOS™ LINK<br>POOL<br>Version 1.00, dated DEC 29, 2004 |
| 3. | LNK 076 DES | DSP/BIOS™ LINK<br>Buffer Pools<br>Version 1.00, dated DEC 29, 2004 |
| 4. | LNK 041 DES | DSP/BIOS™ LINK<br>Zero Copy Link Driver<br>Version 0.80, dated DEC 24, 2004 |

## 1.4 Overview

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

The Link Driver component is responsible for the low level control over the physical link between the GPP and DSP. It provides hardware specific control functions to the modules in Processor Manager.

This document provides a detailed description of the generic link driver and the framework provided by it for plugging in different hardware-specific physical links.

# 2 Requirements

R4 It must be possible to issue multiple buffers into a stream, and all buffers are reclaimed in the order in which they were issued.

R5 It must be possible to specify a finite timeout to wait for a buffer to be reclaimed.

R6 Multiple GPP threads must be able to communicate with multiple DSP threads simultaneously over multiple Link channels.

R7 DSP/BIOS™ LINK shall support communication to both SWI and TSK based DSP threads.

R8 An instrumented version of the GPP side shall provide the number of interrupts exchanged between GPP & DSP, total number of bytes read from and written to DSP memory space, the number of buffers transferred between GPP and DSP on each channel. It would also enable users to inspect the last few (configurable) number of buffers that were exchanged between GPP and DSP over DSPLINK.

R9 DSP/BIOS™ LINK shall provide APIs to query DSP/BIOS™ LINK at runtime for the values of Link driver statistics counters.

R10 DSP/BIOS™ LINK DSP side configuration must allow the exclusion of modules (via the DSP/BIOS™ configuration tool) not required in a specific usage scenario.

R11 DSP/BIOS™ LINK GPP side configuration shall allow exclusion of modules not required in a specific usage scenario.

# 3 Assumptions

The following are assumed in the design:

1. Though the current implementation does not support multiple processors, the design assumes support for multiple DSPs in near future.
2. The function pointer interface provides a reasonable degree of plug-in capability, necessary for scalability of DSP/BIOS™ LINK.
3. The initial implementation shall be tested with only one link driver. The actual test of scalability and plug-in capability may not be feasible until more physical link drivers are implemented.

# 4 High Level Design

## 4.1 Overview

DSP/BIOS LINK provides a uniform API for communication irrespective of the underling hardware/method used for communication.

The Link Driver layer connects the GPP and DSP through its components present on both processors.

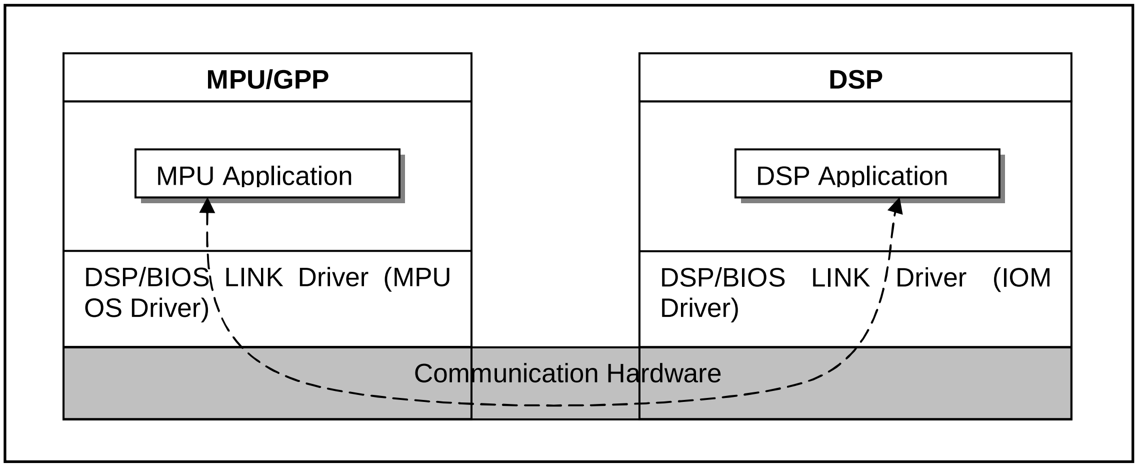
MPU/GPP
DSP
MPU Application
DSP Application
DSP/BIOS LINK Driver (MPU OS Driver)
DSP/BIOS LINK Driver (IOM Driver)
Communication Hardware

**Figure 1.** GPP-DSP connectivity through DSP/BIOS™ LINK

The link driver supports transfer of information between the two processors by two methods: Data transfer and message transfer.

- Data transfer happens through communication channels, which are conceptual entities in DSP/BIOS™ LINK. The channels are conduits used to communicate data between GPP and DSP. Channels can be addressed by specifying their number. These channels are unidirectional, which means a single channel can transfer data either from GPP to DSP or from DSP to GPP.
- Messaging provides logical connectivity between the GPP clients and DSP tasks. Unlike the data transfer channels where the client is waiting for data to arrive on a designated channel, the message transfer is completely asynchronous. The messages may be used to intimate occurrence of an error, change in state of the system, a request based on user input, etc.

DSP/BIOS™ LINK supports multiple links (communication hardware components) for transfer of data. Some examples of these links are USB, PCI, Serial Port, Shared Memory, Shared Memory with DMA, Shared Memory using pointer passing etc. The hardware to be used for data transfer is decided based on the channel identifier.

## 4.2 GPP side

### 4.2.1 Component interaction

The component interaction diagram gives an overview of the interaction of the GPP-side link driver layer with other layers within *DSPLINK*. It also gives information about the various subcomponents within the layer.

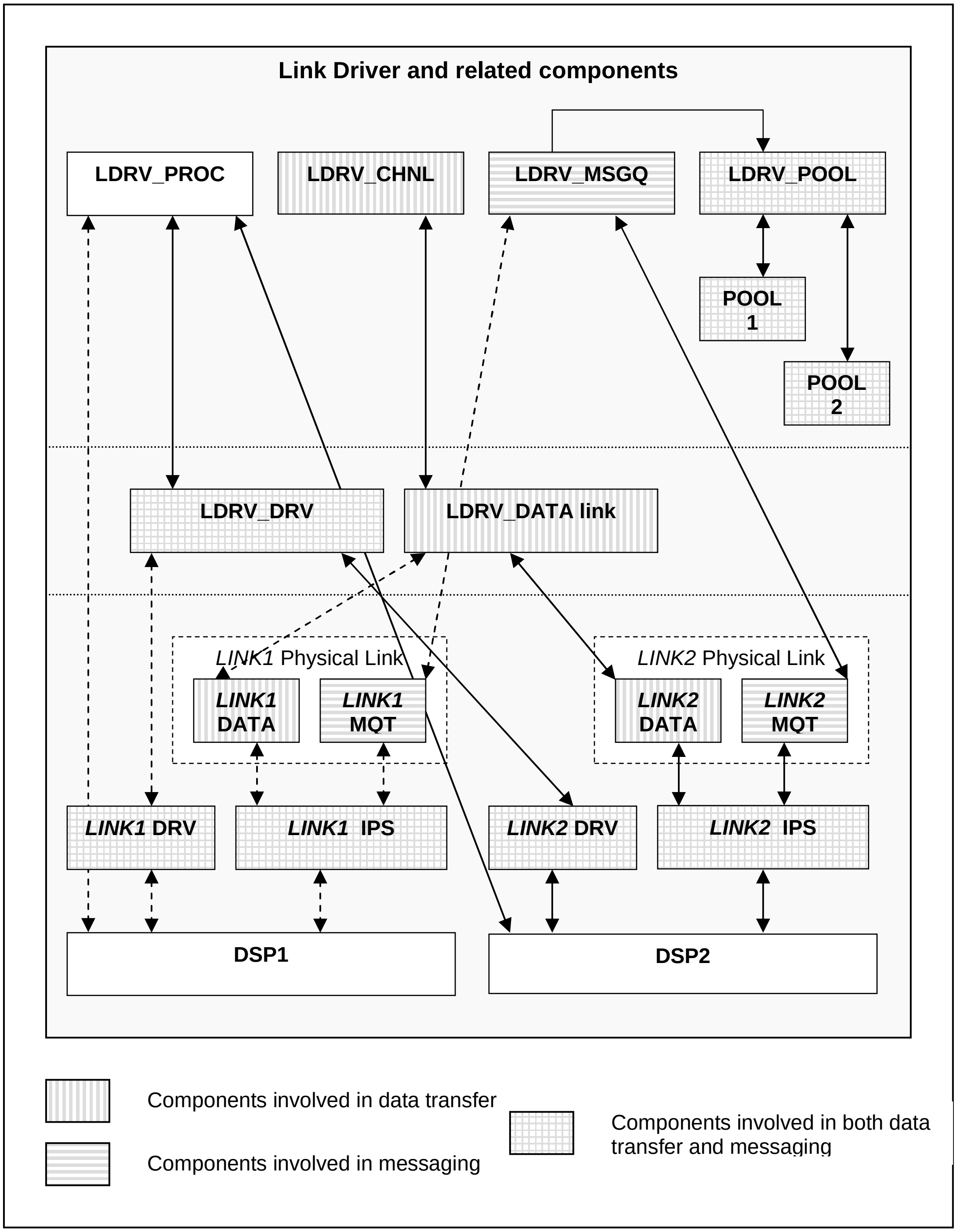

**Figure 2.** Link driver GPP-side component interaction

- *The IPS components for multiple physical links may be combined into a single component providing services common to the links.*

### 4.2.2 Details

The GPP-side Link Driver provides the functionality for PROC, CHNL and MSGQ components through the implementation of the following sub-components:

1. `LDRV_PROC`

This subcomponent provides APIs to access and control the target DSP(s). It also maintains the current state of the target DSP(s).

2. `LDRV_CHNL`

   This subcomponent provides APIs to transfer data between the GPP and the DSP. It allocates and de-allocates the user buffers while opening and closing the channel. During the data transfer, it is responsible for moving the buffers between *FREE*, *REQUESTED* and *COMPLETED* lists.

3. `LDRV_MSGQ`

   This subcomponent provides functions to transfer messages between the GPP and the DSP. It maintains the local message queues and provides the functionality for locating & releasing the local message queues, and sending & receiving messages to & from them. It also communicates with the Message Queue Transports for communication with the DSP(s) in the system.

In addition, support for multiple platforms is provided through the following subcomponents:

1. `DATA`

   The `LDRV_DATA` subcomponent acts as glue between the `LDRV_CHNL` subcomponent and the data driver for a specific platform. It uses the function pointer interface exported by the link driver to communicate with the Link Driver. The map between the channel ID and the underlying data driver ID is maintained by this subcomponent.

   This sub-component also contains implementations of specific data drivers for different platforms and physical links between the GPP and DSP.

2. `DRV`

   This subcomponent encapsulates low-level driver synchronization between the GPP and the DSP over a physical link. Services of this subcomponent are exported by a function pointer interface. This allows the `LDRV_DRV` subcomponent to interact with multiple physical link drivers. The integration of a new link driver into the system is also simple.

   This sub-component also contains implementations of specific link drivers for different platforms and physical links between the GPP and DSP.

3. `DSP`

   This subcomponent encapsulates physical hardware access to communicate with the target DSP. Services of this subcomponent are exported by a function pointer interface. This allows other subcomponents in `LDRV` to interact with multiple DSPs. The integration of a DSP into the system is also simple.

   This subcomponent is designed to be independent of the rest of the sub-system. Applications that do not need the `PROC` and `CHNL` abstractions provided by Processor Manager (PMGR provides) can directly use only the DSP subcomponent.

4. `HAL`

   The Hardware Abstraction Layer provides standard APIs for access and control of hardware specific modules to the sub-components within the *DSPLINK* link driver layer. The services provided by the HAL subcomponent are used by the DSP sub-component.

5. `IPS`

   The Inter-processor signaling (`IPS`) subcomponent provides the upper layers with the service to register an event from the GPP, about which is wishes to be notified. On receiving the event from the GPP, the IPS subcomponent provides information about the event to the registered subcomponent.

   This component uses the services provided on the hardware platform. It provides APIs, which are used by upper layers to establish communication amongst peers at that level.

   The IPS component provides basic services required by the data driver and Message Queue Transport components for transferring data buffers and messages between the processors. It abstracts the platform-specific details by providing standard services to the upper layer.

6. `MQT`

   The `LDRV_MQT` sub-component defines the abstract interface that the Message Queue Transports (MQTs) for specific platforms must implement. There may be multiple MQT implementations for a single platform, based on the physical connection used for connecting the two processors. However, only a single MQT each can be configured at a time for communication between the GPP and a DSP.

   The MQT plugs into the `LDRV_MSGQ` component and provides services to send & receive messages to & from the remote processor, and locate & release message queues on the remote processor.

7. `POOL`

   The `POOL` component provides services to allocate and free data buffers and messages, which can be transferred between the processors.

   The `LDRV_POOL` subcomponent acts as the glue between the `PMGR_POOL` subcomponent and the different pool implementations. It uses the function pointer interface exported by the pool implementations to abstract the functionality implemented by them. The configuration of pool objects in the system is maintained by this component.

   This sub-component also contains implementations of specific pools for the different types of data and message transfer supported by the system.

## 4.3 DSP side

### 4.3.1 Component interaction

The component interaction diagram gives an overview of the interaction of the DSP-side link driver layer with other components external to *DSPLINK*. It also gives information about the various subcomponents within the layer.

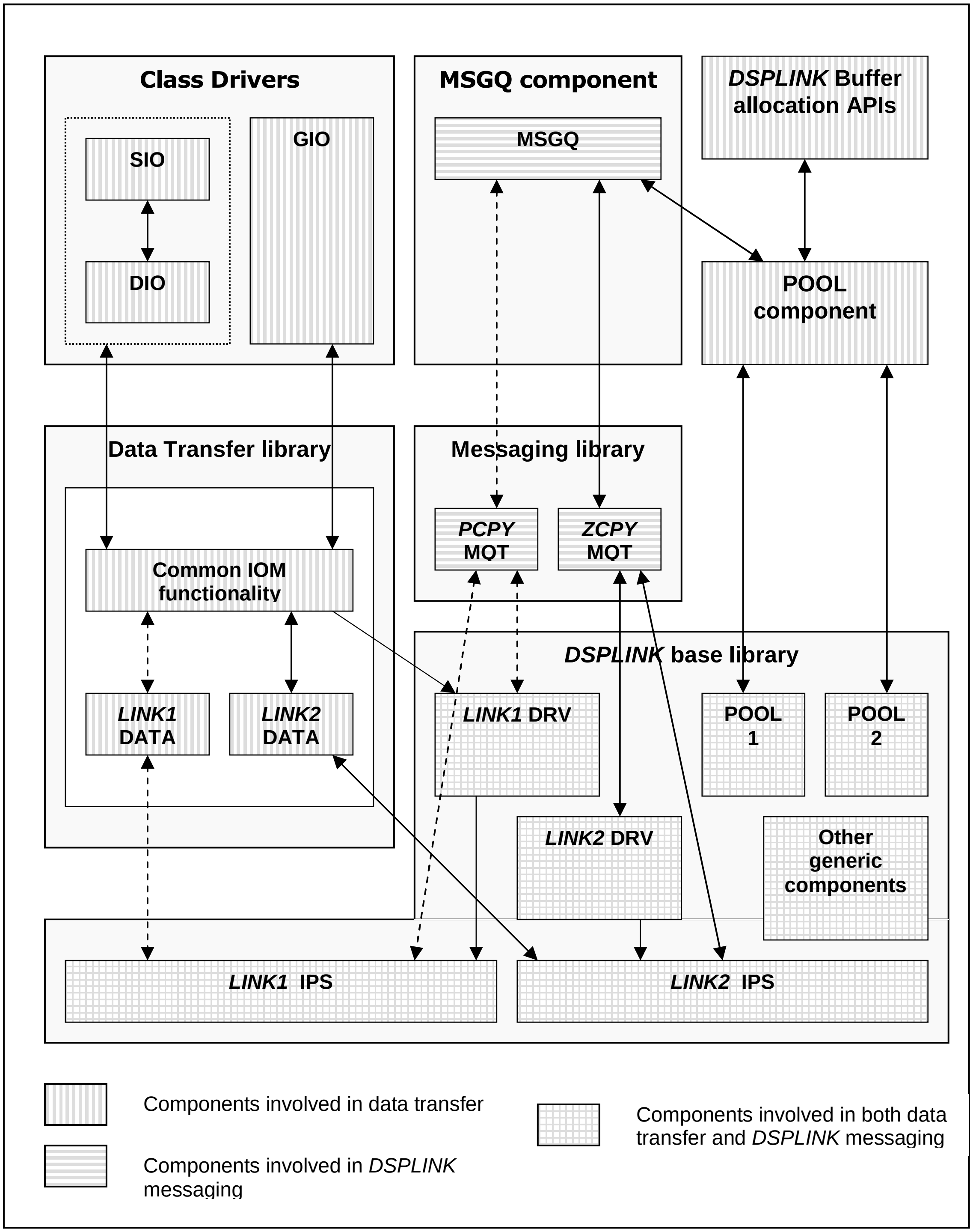

**Figure 3.** Link driver DSP-side component interaction

- *The IPS components for multiple physical links may be combined into a single component providing services common to the links.*

### 4.3.2 Details

The DSP-side of *DSPLINK* is sub-divided into three separate components, based on the functionality provided, and scalability options available:

1. Base component

   The Base *DSPLINK* component contains all the generic sub-components required for both data transfer and messaging. This includes the following:

   - `drv`: Driver initialization and synchronization sub-component
   - `gen`: Generic functions and utilities
   - `hal`: Hardware abstractions functions and utilities
   - `ips`: Inter-Processor-Signaling component
   - `pools`: POOL component for allocating and freeing data buffers and messages

2. Data driver

   The Data Driver is implemented as a DSP/BIOS™ IOM driver. It consists of the following subcomponents:

   1. Common IOM functionality

      This layer brings out the common functionality within the *DSPLINK* IOM driver, which is required by all physical links.

   2. Physical link layer

   The Physical link layer provides a pluggable component that provides the physical connectivity to the GPP. This component is specific to the hardware link available between the GPP and the DSP.

3. MQT

   The MQT sub-component provides messaging functionality between the GPP and the DSP. There may be multiple MQT implementations for a single platform, based on the physical connection used for connecting the two processors. However, only a single MQT each can be configured at a time for communication between the GPP and the DSP.

   The MQT complies with the interface expected by the MSGQ component in DSP/BIOS™. It provides services to send & receive messages to & from the GPP, and locate & release message queues on the GPP.

# 5 LDRV

This module provides a central place to initialize resources that the Link Driver uses.

## 5.1 Dependencies

### 5.1.1 Subordinates

CFG database

### 5.1.2 Preconditions

None.

## 5.2 Description

This subcomponent fetches the configuration data from the CFG database and maintains the information in a global object accessible to all its constituents - LDRV_Obj. The LDRV_Obj also contains the run time information required by LDRV component. This data includes information such as:

1. Number of DSPs configured in the system.
2. Number of data drivers configured in the system. A data driver may be shared between multiple DSPs.
3. Number of memory information tables configured in the system. A MEM table may be shared between multiple DSPs.
4. An array of all data drivers used in the system. If a data driver is configured, but not used, it is not available at run-time.
5. An array of all MEM tables used in the system. If a MEM table is configured, but not used, it is not available at run-time.
6. Array of DSP objects containing run-time information for all target DSPs.

Additional information for configuration of the system for messaging, POOLS etc. is also present.

## 5.3 Typedefs and Data Structures

### 5.3.1 LDRVObject

This structure defines the Link Driver object containing configuration information for the link driver. All the sub-components within the link driver use this object to retrieve information for configuring themselves.

**Definition**

```
typedef struct LdrvObject_tag {
    Uint32          numDsps       ;
    DspObject *     dspObjects    ;
    Uint32          numMemTables  ;
    LinkMemInfo **  memTables     ;

#if defined (CHNL_COMPONENT) || defined (MSGQ_COMPONENT)
    Uint32          numIpsTables  ;
    IpsObject **    ipsTables     ;
    LinkObject *    linkObjects   ;
    Uint32          numPools      ;
    PoolObject *    poolObjects   ;
#endif

#if defined (CHNL_COMPONENT)
    Uint32          queueLength    ;
    Uint32          numDataTables  ;
    DataObject **   dataTables     ;
#endif

#if defined (MSGQ_COMPONENT)
    Uint32          numMqts       ;
    Uint32          maxMsgqs      ;
    MqtObject *     mqtObjects    ;
#endif

#if defined (DDSP_PROFILE)
    ProcStats       procStats     ;
#if defined (CHNL_COMPONENT)
    ChnlStats       chnlStats     ;
#endif
#if defined (MSGQ_COMPONENT)
    MsgqStats       msgqStats     ;
#endif
#endif
} LdrvObject ;
```

**Fields**

| | |
|---|---|
| `numDsps` | Number of DSPs connected to the GPP. |
| `dspObjects` | Array of DSP objects. |
| `numMemTables` | Number of MEM tables specified in configuration database. |
| `memTables` | Array of pointers to link memory information tables. |

| | |
|---|---|
| `numIpsTables` | Number of IPS tables. |
| `ipsTables` | Array of pointers to IPS tables. |
| `linkObjects` | Array of link objects. |
| `numPools` | Number of pools. |
| `poolObjects` | Array of pool objects. |
| `queueLength` | Maximum number of data buffers that can be queued on a channel at a time pending transfer. |
| `numDataTables` | Number of data tables. |
| `dataTables` | Array of pointers to data tables. |
| `numMqts` | Number of Message Queue Transports. |
| `maxMsgqs` | Maximum number of local message queues. |
| `mqtObjects` | Array of MQT objects. |
| `procStats` | Statistics object for processor subcomponent. |
| `chnlStats` | Statistics object for channel subcomponent. |
| `msgqStats` | Statistics object for messaging subcomponent. |

#### Comments

None.

#### See Also

```
DspObject
LinkMemInfo
IpsObject
LinkObject
PoolObject
DataObject
MqtObject
ProcStats
ChnlStats
MsgqStats
```

## 5.4 API Definition

### 5.4.1 LDRV_Initialize

This function initializes the LDRV component. It fetches the configuration data from the CFG database and makes it available for access at run time. It also allocates and initializes the global runtime objects required within LDRV context.

**Syntax**

```
DSP_STATUS LDRV_Initialize () ;
```

**Arguments**

None.

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully. |
| DSP_EMEMORY | Generic failure while allocating memory. |
| DSP_EFAIL | General error returned from GPP OS |

**Comments**

None.

**Constraints**

None.

**See Also**

LDRV_Finalize

### 5.4.2 LDRV_Finalize

This function releases all the resources that were allocated earlier by a call to function LDRV_Initialize ().

**Syntax**

```
DSP_STATUS LDRV_Finalize () ;
```

**Arguments**

None.

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully. |
| DSP_EMEMORY | Generic failure while freeing memory |
| DSP_EFAIL | General error returned from GPP OS |

**Comments**

None.

### Constraints

None.

### See Also

`LDRV_Initialize`

# 6 LDRV_PROC

This subcomponent provides services to control the DSP processor. The generic control function may be – reset, start, stop, read, write, send interrupt, clear interrupt, etc.

## 6.1 Dependencies

### 6.1.1 Subordinates

DSP subcomponent

### 6.1.2 Preconditions

The PROC subcomponents in API and PMGR must validate all data before passing it to LDRV_PROC. LDRV_PROC does not perform a runtime check on the function arguments and assumes runtime validation of arguments by calling the functions.

## 6.2 Description

This component provides APIs to read from and write into the DSP memory space, allowing the PROC subcomponent (in PMGR) to load a DSP executable onto the target DSP. This subcomponent uses the services of a DSP module to perform its tasks.

This subcomponent also implements a state machine to encapsulate the current state of the DSP. Figure 4 shows the state transition diagram for LDRV_PROC.

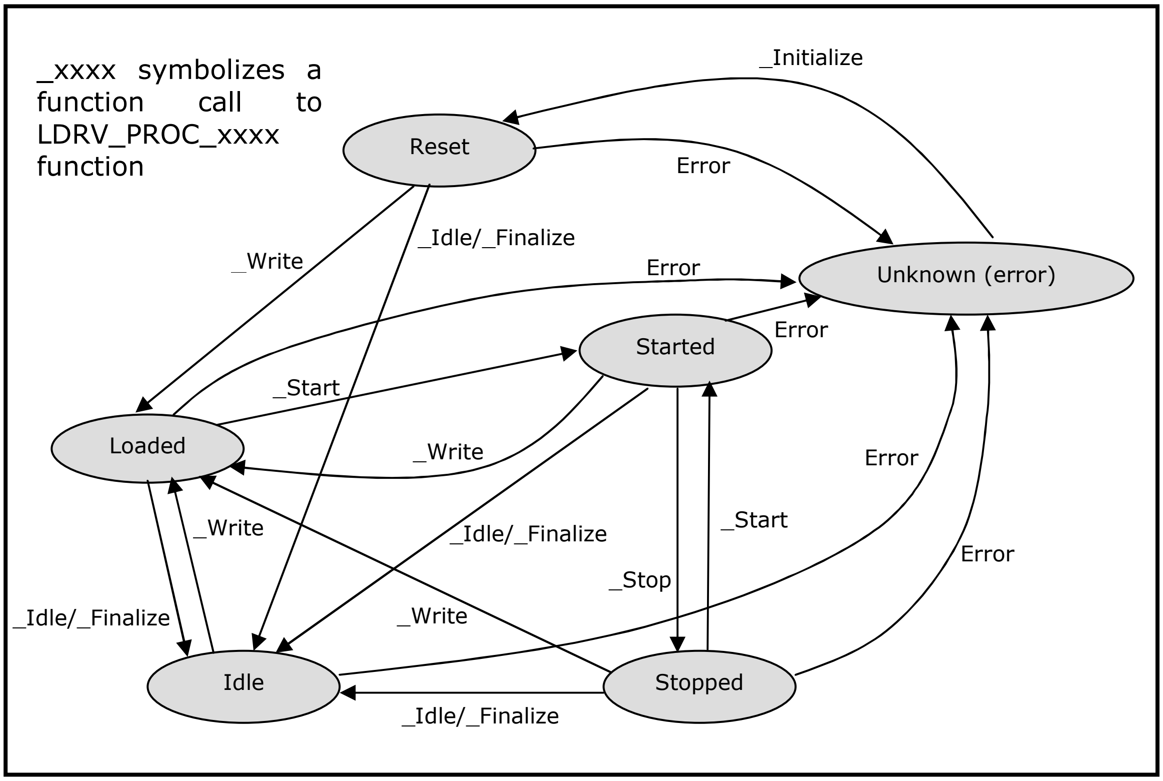

**Figure 4.** LDRV_PROC State Transition Diagram

## 6.3 API Definition

### 6.3.1 LDRV_PROC_Initialize

This function sets up the peripherals required to make the target DSP reachable from the GPP. This function also calls the *initialize* function exported by the corresponding the DSP subcomponent. The target DSP is in the `RESET` state after successful completion of this function.

**Syntax**

```
DSP_STATUS LDRV_PROC_Initialize (ProcessorId dspId)
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP to initialize | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

**See Also**

`LDRV_PROC_Finalize`

### 6.3.2 LDRV_PROC_Finalize

This function releases the communication between the GPP and the target DSP. This design ensures that the DSP is in `RESET` state after successful completion of this function. This behavior may be customized depending upon the application needs.

**Syntax**

```
DSP_STATUS LDRV_PROC_Finalize (ProcessorId dspId);
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP to finalize | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

`LDRV_PROC_Initialize`

### 6.3.3 LDRV_PROC_Start

This function starts the DSP run from the specified address. The target DSP is in the `STARTED` state after successful completion of this function.

Communication between the GPP and the target DSP may require handshake over certain physical links before any data transfer can happen. This function initiates the handshake process.

**Syntax**

```
DSP_STATUS LDRV_PROC_Start (ProcessorId dspId, Uint32 dspAddr)
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP to start | |
| `IN` | `Uint32` | `dspAddr` |
| | Address to start execution on the DSP | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |
| `DSP_EWRONGSTATE` | Operation performed in wrong state. |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must be either in the `Loaded` or in the `Stopped` state.

**See Also**

`LDRV_PROC_Stop`

### 6.3.4 LDRV_PROC_Stop

This function stops the DSP execution. The target DSP is in the `STOPPED` state after successful completion of this function.

**Syntax**

```
DSP_STATUS LDRV_PROC_Stop (ProcessorId dspId)
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP to stop | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |
| `DSP_EWRONGSTATE` | Operation performed in wrong state. |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must be either in the `Started` or in the `Stopped` state.

**See Also**

`LDRV_PROC_Start`

### 6.3.5 LDRV_PROC_Idle

This function puts the DSP in idle mode. On successful execution of this function, the DSP is running the *IDLE* code.

**Syntax**

```
DSP_STATUS LDRV_PROC_Idle (ProcessorId dspId)
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP to idle | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |
| `DSP_EWRONGSTATE` | Operation performed in wrong state. |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

```
LDRV_PROC_Initialize
LDRV_PROC_Finalize
```

### 6.3.6 LDRV_PROC_Read

This function reads specified number of bytes from the DSP memory space in a given buffer.

**Syntax**

```
DSP_STATUS LDRV_PROC_Read (ProcessorId   dspId,
                           Uint32        dspAddr,
                           Endianism     endianInfo,
                           Uint32 *      numBytes,
                           Uint8 *       buffer) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | Uint32 | dspAddr |
| | Address from where to read | |
| IN | Endianism | endianInfo |
| | Specifies endianism attribute of the target memory | |
| IN | Uint32 * | numBytes |
| | Number of bytes to read | |
| OUT | Uint8 * | buffer |
| | Buffer to store the read data | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General error from the GPP OS |
| DSP_EWRONGSTATE | Operation performed in wrong state |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

```
LDRV_PROC_Initialize
LDRV_PROC_Write
```

### 6.3.7 LDRV_PROC_Write

This function writes specified number of bytes to the DSP memory space from a given buffer.

**Syntax**

```
DSP_STATUS LDRV_PROC_Write (ProcessorId    dspId,
                            Uint32         dspAddr,
                            Endianism      endianInfo,
                            Uint32         numBytes,
                            Uint8 *        buffer) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | Uint32 | dspAddr |
| | Address to which we need to write | |
| IN | Endianism | endianInfo |
| | Specifies endianism attribute of the target memory | |
| IN | Uint32 | numBytes |
| | Number of bytes to write | |
| IN | Uint 8 * | buffer |
| | Buffer containing data to write | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General error from the GPP OS |
| DSP_EWRONGSTATE | Operation performed in wrong state |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

```
LDRV_PROC_Initialize
LDRV_PROC_Read
```

### 6.3.8 LDRV_PROC_GetState

This function gets the current state of the DSP.

**Syntax**

```
DSP_STATUS LDRV_PROC_GetState (ProcessorId  dspId,
                               ProcState *  procState) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| OUT | ProcState * | procState |
| | OUT argument to return the current state of the DSP | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

The state of the DSP is maintained locally by this subcomponent.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

```
LDRV_PROC_Initialize
LDRV_PROC_Finalize
LDRV_PROC_Idle
LDRV_PROC_Start
LDRV_PROC_Stop
```

### 6.3.9 LDRV_PROC_SetState

Sets the current state of processor to the specified state.

**Syntax**

```
DSP_STATUS LDRV_PROC_SetState (ProcessorId  dspId,
                               ProcState    procState) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | ProcState | procState |
| | The new state of the DSP | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

The state of the DSP is maintained locally by this subcomponent.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

**See Also**

```
LDRV_PROC_Initialize
LDRV_PROC_Finalize
LDRV_PROC_Idle
LDRV_PROC_Start
LDRV_PROC_Stop
```

### 6.3.10 LDRV_PROC_Control

Provides a hook to perform device dependent control operations.

**Syntax**

```
DSP_STATUS LDRV_PROC_Control (ProcessorId dspId,
                              Int32       cmd,
                              Pvoid       arg) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | Int32 | cmd |
| | Command identifier. | |
| IN | Pvoid | Arg |
| | Optional argument | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid argument |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before calling this function.

The DSP must not be in the `Error` state.

**See Also**

None.

### 6.3.11 LDRV_PROC_Debug

This is a debug mode function. It prints the debug information of the specified DSP.

**Syntax**

```
Void LDRV_PROC_Debug (IN ProcessorId procId)
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the target DSP | |

**Return Values**

None.

**Comments**

None.

**Constraints**

None.

**See Also**

None.

### 6.3.12 LDRV_PROC_Instrument

This function is defined only if profiling is enabled. It returns the statistics information (instrumentation data) of the specified DSP.

**Syntax**

```
DSP_STATUS LDRV_PROC_Instrument (ProcessorId      procId,
                                 ProcInstrument * retVal)
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the target DSP | |
| IN | ProcInstrument * | retVal |
| | OUT argument to contain the instrumentation information | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid argument |

**Comments**

None.

#### Constraints

`retVal` must be a valid pointer.

#### See Also

`None.`

# 7 LDRV_CHNL

This subcomponent provides buffer management services for all logical channels in DSP/BIOS™ LINK

## 7.1 Dependencies

### 7.1.1 Subordinates

`LDRV_IO`

### 7.1.2 Preconditions

The CHNL subcomponent in PMGR must validate all data before passing it to `LDRV_CHNL`. `LDRV_CHNL` does not perform a runtime check on the function arguments and assumes runtime validation of arguments by calling the functions.

## 7.2 Description

It creates three different queues to manage the data buffers. A queue of:

1. Free buffers
2. Buffers on which data transfer is requested, and,
3. Buffers on which data transfer has been completed or cancelled.

It also provides APIs for use by CHNL (of PMGR) subcomponent to perform data transfer between the GPP and the DSP. These APIs work in conjunction with the `LDRV_DATA` subcomponent.

## 7.3 Constants and Enumerations

### 7.3.1 IO Completion State flags.

Status of I/O completion

**Definition**

```
#define LDRV_CHNL_IOCSTATE_COMPLETE     0x0000
#define LDRV_CHNL_IOCSTATE_CANCELED     0x0002
#define LDRV_CHNL_IOCSTATE_TIMEOUT      0x0008
#define LDRV_CHNL_IOCSTATE_EOS          0x8000
```

**Comments**

LDRV_CHNL_IOCSTATE_COMPLETE: IO Completed

LDRV_CHNL_IOCSTATE_CANCELED: IO was cancelled

LDRV_CHNL_IOCSTATE_TIMEOUT: Wait for IOC timed out

LDRV_CHNL_IOCSTATE_EOS: End Of Stream reached

**Constraints**

None.

**See Also**

None.

### 7.3.2 ChannelState

Channel State type

**Definition**

```
typedef enum  {
    ChannelState_Ready  = 0x01,
    ChannelState_Idled  = 0x02,
    ChannelState_EOS    = 0x04,
    ChannelState_Closed = 0x08
} ChannelState ;
```

**Fields**

| | |
|---|---|
| ChannelState_Ready | Indicates channel is ready. |
| ChannelState_Idled | Indicates channel is idled. |
| ChannelState_EOS | Indicates channel is in End of Stream state. |
| ChannelState_Closed | Indicates channel is in closed state. |

**Comments**

None.

**Constraints**

None.

**See Also**

None.

### 7.3.3 IOState

Completion state of IO on a channel

**Definition**

```
typedef enum  {
    IOState_Completed    = 1,
    IOState_NotCompleted = 2
} IOState ;
```

**Fields**

| | |
|---|---|
| `IOState_Completed` | Indicates completion of IO for an IO request on a channel. |
| `IOState_NotCompleted` | Indicates non-completion of IO for an IO request on a channel. |

**Comments**

None.

**Constraints**

None.

**See Also**

None.

## 7.4 Typedefs and Data Structures

### 7.4.1 LDRVChnlObject

This structure defines the channel object maintained for every channel opened on a per DSP basis.

**Definition**

```
typedef struct LDRVChnlObject_tag {
    Uint32          signature     ;
    Uint32          bufSize       ;
    ChannelState    chnlState     ;
    List *          freeList      ;
    List *          requestList   ;
    List *          completedList ;
    ChannelAttrs    attrs         ;
    SyncEvObject *  syncEvent     ;
    SyncEvObject *  chnlIdleSync  ;
} LDRVChnlObject ;
```

**Fields**

| | |
|---|---|
| `signature` | Signature of object |
| `bufSize` | Size of buffers on this channel. |
| `chnlState` | State of the channel |
| `freeList` | List for free channel IO request packets (CHIRP) |
| `requestList` | List for requested CHIRPs |
| `completedList` | List for completed CHIRPs |
| `attrs` | Attributes of this CHIRPs |
| `syncEvent` | Event to signal when some IO is completed or cancelled for this channel |
| `chnlIdleSync` | Event to signal when channel has no more pending IO requests. |

**Comments**

None.

**See Also**

```
ChannelAttrs
LDRVChnlIRP
LDRVChnlIOInfo
```

### 7.4.2 LDRVChnlIRP

This structure encapsulates information associated with an IO buffer.

**Definition**

```
typedef struct LDRVChnlIRP_tag {
    ListElement         link          ;
    Uint32              buffer        ;
    Uint32              arg           ;
    Uint32              size          ;
    Uint32              iocStatus     ;
    ChannelId           chnlId        ;
} LDRVChnlIRP ;
```

**Fields**

| | |
|---|---|
| `link` | List element header needed for this structure |
| `buffer` | Buffer to fill/empty |
| `arg` | Issue reclaim argument |
| `size` | Buffer length |
| `iocStatus` | Status of IO completion |
| `chnlId` | Channel ID |

**Comments**

None.

**See Also**

`LDRVChnlObject`

### 7.4.3 LDRVChnlIOInfo

This structure encapsulates information about a data transfer buffer.

#### Definition

```
typedef struct LDRVChnlIOInfo_tag {
    Pvoid               buffer            ;
    Uint32              size              ;
    Uint32              arg               ;
    IOState             completionStatus  ;
} LDRVChnlIOInfo ;
```

#### Fields

| | |
|---|---|
| `buffer` | Pointer to the data buffer |
| `size` | Size of the data buffer |
| `arg` | Argument to send or received together with the data buffer |
| `completionStatus` | Completion status of this IO request |

#### Comments

None.

#### See Also

`LDRVChnlObject`

## 7.5 API Definition

### 7.5.1 LDRV_CHNL_Initialize

This function allocates and initializes the resources required by this module. It also initializes the data driver for the specific physical link by calling LDRV_DATA_Initialize ().

**Syntax**

```
DSP_STATUS  LDRV_CHNL_Initialize (ProcessorId procId)
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

None.

**Constraints**

procId must be valid.

**See Also**

LDRV_CHNL_Finalize
LDRV_CHNL_Open

### 7.5.2 LDRV_CHNL_Finalize

This function closes all open channels (if any). It then closes the data driver for the specific physical link by calling LDRV_DATA_Finalize ().

**Syntax**

```
DSP_STATUS LDRV_CHNL_Finalize (ProcessorId procId)
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |

| | |
|---|---|
| DSP_EFAIL | General failure |

**Comments**

None.

**Constraints**

procId must be valid.

**See Also**

LDRV_CHNL_Initialize

### 7.5.3 LDRV_CHNL_Open

This function prepares the specified channel for data transfer. It creates the three required queues for buffer management on the channel. It also creates the SYNC objects required for waiting on a pending data transfer request.

**Syntax**

```
DSP_STATUS LDRV_CHNL_Open (ProcessorId      procId,
                           ChannelId        chnlId,
                           ChannelAttrs *   attrs) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel to open | |
| IN | ChannelAttrs * | attrs |
| | Channel attributes | |

**Return Value**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |
| CHNL_E_BUSY | Channel already in use. |

**Comments**

None.

**Constraints**

procId must be valid.

chnlId must be valid.

attrs must be a valid pointer.

**See Also**

```
LDRV_CHNL_Initialize
```

### 7.5.4 LDRV_CHNL_Close

This function closes the specified channel. It frees all the resources allocated earlier in the call to `LDRV_CHNL_Open ()`.

Once a channel is closed, no further IO can be performed on it, unless it is opened again.

**Syntax**

```
DSP_STATUS LDRV_CHNL_Close (ProcessorId  procId,
                            ChannelId    chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Channel to close | |

**Return Value**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

```
LDRV_CHNL_Initialize
LDRV_CHNL_Open
```

### 7.5.5 LDRV_CHNL_AllocateBuffer

This function allocates an array of buffers of specified size and returns them to the client. The pool configured for usage by the data driver is used for allocating the data buffers.

**Syntax**

```
DSP_STATUS LDRV_CHNL_AllocateBuffer (IN  ProcessorId procId,
                                     IN  ChannelId   chnlId,
                                     OUT Char8 **    bufArray,
                                     IN  Uint32      size,
                                     IN  Uint32      numBufs) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Processor Identifier. | |
| IN | `ChannelId` | `chnlId` |
| | Channel Identifier. | |
| OUT | `Char8 **` | `bufArray` |
| | Pointer to receive array of allocated buffers. | |
| IN | `Uint32` | `size` |
| | Size of each buffer. | |
| IN | `Uint32` | `numBufs` |
| | Number of buffers to allocate. | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

None.

**Constraints**

Processor and channel ids must be valid.

`bufArray` must be valid.

`numBufs` must be less than maximum limit.

**See Also**

```
LDRV_CHNL_Initialize
LDRV_CHNL_FreeBuffer
```

### 7.5.6 LDRV_CHNL_FreeBuffer

This function frees buffer(s) allocated by `LDRV_CHNL_AllocateBuffer ()`. The pool configured for usage by the data driver is used for freeing the data buffers.

**Syntax**

```
DSP_STATUS LDRV_CHNL_FreeBuffer (IN ProcessorId procId,
                                 IN ChannelId   chnlId,
                                 IN Char8 **    bufArray,
                                 IN Uint32      numBufs) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Processor Identifier. | |
| IN | `ChannelId` | `chnlId` |
| | Channel Identifier. | |
| IN | `Char8 **` | `bufArray` |
| | Pointer to array of buffers to be freed. | |
| IN | `Uint32` | `numBufs` |
| | Number of buffers to be freed. | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |

**Comments**

None.

**Constraints**

Processor and channel ids must be valid.

`bufArray` must be valid.

`numBufs` must be less than maximum limit.

**See Also**

```
LDRV_CHNL_Initialize
LDRV_CHNL_AllocateBuffer
```

### 7.5.7 LDRV_CHNL_AddIORequest

This function adds an IO request on a channel. An IO request may be a request for transferring a buffer from the GPP to DSP or from the DSP to GPP. The attributes specified while creating the channel determines the direction of the data transfer.

**Syntax**

```
DSP_STATUS LDRV_CHNL_AddIORequest (ProcessorId      procId,
                                   ChannelId        chnlId,
                                   LDRVChnlIOInfo * ioInfo)
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Identifier for the DSP | |

IN ChannelId chnlId

Channel to send/receive data

IN LDRVChnlIOInfo * ioInfo

The IOInfo structure containing information regarding the IO request

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |
| CHNL_E_EOS | Channel is in EOS (End of Stream) state. |
| CHNL_E_NOIRPS | No more IO could be accepted because maximum limit of pending IO request has reached |

**Comments**

None.

**Constraints**

procId must be valid.

ChnlId must be valid.

IoInfo must be a valid pointer.

**See Also**

LDRVChnlIOInfo
LDRV_CHNL_GetIOCompletion

### 7.5.8 LDRV_CHNL_GetIOCompletion

This function gets a buffer on which IO is complete. It waits for a specified amount of time, if required and specified, for an IO completion event on a channel. On successful completion, the function returns a buffer to the caller. The contents of the buffer depend on the direction of channel.

For an input channel, the buffer contains valid data as received from the DSP and for an output channel, the buffer is an empty buffer that was earlier used to send data to the DSP.

**Syntax**

```
DSP_STATUS LDRV_CHNL_GetIOCompletion (ProcessorId      procId,
                                      ChannelId        chnlId,
                                      Uint32           timeout,
                                      LDRVChnlIOInfo * ioInfo) ;
```

**Arguments**

IN ProcessorId procId

Identifier for the DSP

| | | |
|---|---|---|
| IN | ChannelId | chnlId |
| | Channel on which to send/receive data | |
| IN | Uint32 | timeout |
| | Timeout value | |
| OUT | LDRVChnlIOInfo * | ioInfo |
| | Structure containing the OUT buffer pointer and also any values associated with the buffer | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Memory error occurred |
| DSP_EFAIL | General error from the GPP OS |
| DSP_ETIMEOUT | Timout occurred while performing the operation. |
| CHNL_E_NOIOC | Timeout parameter was "NO_WAIT", yet no I/O completions were queued. |

**Comments**

None.

**Constraints**

procId must be valid.

chnlId must be valid.

ioInfo must be a valid pointer.

**See Also**

LDRVChnlIOInfo
LDRV_CHNL_AddIORequest
LDRV_CHNL_AddIOCompletion

### 7.5.9 LDRV_CHNL_AddIOCompletion

This function performs the required operations for completing an IO operation on a CHIRP.

**Syntax**

```
DSP_STATUS LDRV_CHNL_AddIOCompletion (ProcessorId   procId,
                                      ChannelId     chnlId,
                                      LDRVChnlIRP * chirp) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |

Identifier for the DSP

| | | |
|---|---|---|
| IN | `ChannelId` | `chnlId` |
| | Identifier for the channel | |
| IN | `LDRVChnlIRP *` | `chirp` |
| | The IO request packet on which IO is complete | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EFAIL` | General error from the GPP OS |

**Comments**

This function adds the specified CHIRP to the queue containing CHIRPs on which IO is complete.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

`chirp` must be a valid pointer.

**See Also**

`None.`

### 7.5.10 LDRV_CHNL_Idle

In case of input mode channel this function discards all pending input requests from the channel. In case of output mode channel, action of this function depends upon the flush parameter and is as follows:

- If flush is TRUE this function blocks till all output buffers are transferred to the DSP.
- If flush is FALSE this function discards all the output requests pending on this channel without blocking.

**Syntax**

```
DSP_STATUS  LDRV_CHNL_Idle (ProcessorId  procId,
                            ChannelId    chnlId,
                            Bool         flush) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Identifier for the DSP | |
| IN | `ChannelId` | `chnlId` |
| | Channel on which to cancel IO | |

| | | |
|---|---|---|
| IN | Bool | flush |
| | This parameter tells whether to block or not on output mode channels. | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General error from the GPP OS |
| DSP_EMEMORY | Memory error occurred |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

```
LDRV_CHNL_AddIORequest
LDRV_CHNL_GetIOCompletion
```

### 7.5.11 LDRV_CHNL_Control

Provides a hook to perform device dependent control operations on channels.

**Syntax**

```
DSP_STATUS LDRV_CHNL_Control (ProcessorId  procId,
                              ChannelId    chnlId,
                              Int32        cmd,
                              Pvoid        arg) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Processor Identifier | |
| IN | ChannelId | chnlId |
| | Channel Identifier | |
| IN | Int32 | cmd |
| | Command id. | |
| IN | Pvoid | arg |
| | Optional argument | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |

| | |
|---|---|
| `DSP_ENOTIMPL` | Functionality not implemented |

**Comments**

This function provides a hook to perform the device dependent control operations on channels. Not implemented in current implementation

**Constraints**

None.

**See Also**

`LDRV_CHNL_Initialize`

### 7.5.12 LDRV_CHNL_GetChannelMode

This function gets the mode of the channel (Input or Output).

**Syntax**

```
ChannelMode LDRV_CHNL_GetChannelMode (ProcessorId   procId,
                                      ChannelId     chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Identifier for the DSP | |
| IN | `ChannelId` | `chnlId` |
| | Identifier for the channel | |

**Return Values**

| | |
|---|---|
| `ChannelMode_Input` | The channel is an input channel. |
| `ChannelMode_Output` | The channel is an output channel. |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

None.

### 7.5.13 LDRV_CHNL_GetChannelState

This function gets the current state of the channel.

**Syntax**

```
ChannelState LDRV_CHNL_GetChannelState (ProcessorId   procId,
                                        ChannelId     chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Identifier for the DSP | |
| IN | `ChannelId` | `chnlId` |
| | Identifier for the channel | |

**Return Value**

The current state of the channel

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

None.

### 7.5.14 LDRV_CHNL_SetChannelState

This function sets the channel's state.

**Syntax**

```
Void LDRV_CHNL_SetChannelState (ProcessorId   procId,
                                ChannelId     chnlId,
                                ChannelState  state) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `procId` |
| | Identifier for the DSP | |
| IN | `ChannelId` | `chnlId` |
| | Identifier for the channel | |
| IN | `ChannelState` | `state` |
| | New state of the channel. | |

**Return Value**

None.

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

None.

### 7.5.15 LDRV_CHNL_GetChannelEndianism

This function gets the data endianism associated with a channel.

**Syntax**

```
Endianism LDRV_CHNL_GetChannelEndianism (ProcessorId   procId,
                                         ChannelId     chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel | |

**Return Value**

The endianism associated with the specified channel.

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

None.

### 7.5.16 LDRV_CHNL_HasMoreChirps

This function returns TRUE if the channel has more chirps in the IO request queue.

**Syntax**

```
Bool LDRV_CHNL_HasMoreChirps (ProcessorId   procId,
                              ChannelId     chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |

Identifier for the channel

**Return Values**

| | |
|---|---|
| `TRUE` | The channel has more request CHIRPs. |
| `FALSE` | The requested queue in the channel is empty. |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

`None.`

### 7.5.17 LDRV_CHNL_NextRequestChirp

This function returns a pointer to a CHIRP from the request queue of a channel without removing it from the queue.

**Syntax**

```
LDRVChnlIRP * LDRV_CHNL_NextRequestChirp (ProcessorId   procId,
                                          ChannelId     chnlId) ;
```

**Arguments**

```
IN      ProcessorId              procId
```

Identifier for the DSP

```
IN      ChannelId                chnlId
```

Identifier for the channel

**Return Values**

| | |
|---|---|
| `NULL` | If the request list is empty |
| `Non-NULL` | Pointer to a CHIRP from the request queue |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

`None.`

### 7.5.18 LDRV_CHNL_GetRequestChirp

This function returns a pointer to a CHIRP from the request queue of a channel and removes it from the queue.

#### Syntax

```
LDRVChnlIRP * LDRV_CHNL_GetRequestChirp (IN ProcessorId   procId,
                                         IN ChannelId     chnlId) ;
```

#### Arguments

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel | |

#### Return Values

| | |
|---|---|
| NULL | If the request list is empty |
| Non-NULL | Pointer to a CHIRP from the request queue |

#### Comments

None.

#### Constraints

procId must be valid.

chnlId must be valid.

#### See Also

None.

### 7.5.19 LDRV_CHNL_Debug

This is a debug mode function. It prints the debug information of the specified channel.

#### Syntax

```
Void LDRV_CHNL_Debug (ProcessorId procId, ChannelId chnlId) ;
```

#### Arguments

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel | |

#### Return Value

None.

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

**See Also**

None.

### 7.5.20 LDRV_CHNL_Instrument

This function is defined only if profiling is enabled. It returns the statistics information (instrumentation data) of the specified channel.

**Syntax**

```
DSP_STATUS LDRV_CHNL_Instrument (ProcessorId       procId,
                                 ChannelId         chnlId, ,
                                 ChnlInstrument *  retVal)
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel | |
| OUT | ChnlInstrument * | retVal |
| | OUT argument to contain the instrumentation information. | |

**Return Value**

| | |
|---|---|
| DSP_SOK | Operation successfully completed. |
| DSP_SOK | retVal is invalid. |

**Comments**

None.

**Constraints**

`procId` must be valid.

`chnlId` must be valid.

`retVal` must be valid.

**See Also**

None.

# 8 LDRV_MSGQ

This subcomponent provides functions to transfer messages between the GPP and the DSP.

## 8.1 Dependencies

### 8.1.1 Subordinates

The MQT subcomponent is used by this subcomponent for interacting with the DSP.

The POOL subcomponent is used for allocating and freeing messages to be transferred between the processors.

## 8.2 Description

This subcomponent provides functions to transfer messages between the GPP and the DSP. It maintains the local message queues and provides the functionality for locating & releasing the local message queues, and sending & receiving messages to & from them. It passes all the requests to the actual MQT for the physical link, using its function pointer interface exported by the MQT.

Usage of function pointer interface ensures that multiple MQTs can be easily plugged into the system.

The configuration is used to determine the MQT to be used.

The design of the MSGQ component is provided within the Messaging Design document [Ref. 2].

# 9 LDRV_DATA

This subcomponent acts as glue between the LDRV_CHNL and the data driver(s) for the physical link.

## 9.1 Dependencies

### 9.1.1 Subordinates

The DSP subcomponent is used by this subcomponent for interacting with the DSP.

## 9.2 Description

This subcomponent provides the logical data transfer services to LDRV_CHNL. It passes all the requests to the actual data driver for the physical link, using its function pointer interface exported by the data driver(s).

Usage of function pointer interface ensures that multiple data drivers can be easily plugged into the system.

To determine the data driver to be used, it maintains a map between channel ID and the data driver ID.

## 9.3 API Definition

### 9.3.1 LDRV_DATA_Initialize

This function initializes the resources required by this module. It also calls the function *initialize* from the function pointer interface exported by all data drivers attached to the specified DSP.

**Syntax**

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

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Out of memory |
| DSP_EFAIL | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

```
LDRV_Initialize
LDRV_DATA_Finalize
```

### 9.3.2 LDRV_DATA_Finalize

This function releases the resources required by this module. It also calls the function *finalize* from the function pointer interface exported by all data drivers attached to the specified DSP.

**Syntax**

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

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | DSP ID of DSP for which the finalization must be performed | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Out of memory |

| | |
|---|---|
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

```
LDRV_Initialize
LDRV_DATA_Initialize
```

### 9.3.3 LDRV_DATA_OpenChannel

This function opens the physical channel corresponding to the specified logical channel by calling the function *openChannel* from corresponding data driver's function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DATA_OpenChannel (ProcessorId dspId,
                                  ChannelId   chnlId) ;
```

**Arguments**

```
IN      ProcessorId               dspId
```

Identifier for the DSP

```
IN      ChannelId                 chnlId
```

Identifier for the channel

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Out of memory |
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

```
LDRV_Initialize
LDRV_DATA_CloseChannel
```

### 9.3.4 LDRV_DATA_CloseChannel

This function closes the physical channel corresponding to the specified logical channel by calling the function *closeChannel* from corresponding link driver's function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DATA_CloseChannel (ProcessorId dspId,
                                   ChannelId   chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | ChannelId | chnlId |
| | Identifier for the channel | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EMEMORY | Out of memory |
| DSP_EFAIL | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

```
LDRV_Initialize
LDRV_DATA_OpenChannel
```

### 9.3.5 LDRV_DATA_AllocateBuffer

This function allocates an array of buffers of specified size and returns them to the client. The pool configured for usage by the data driver is used for allocating the data buffers.

**Syntax**

```
DSP_STATUS LDRV_DATA_AllocateBuffer (IN  ProcessorId procId,
                                     IN  ChannelId   chnlId,
                                     OUT Char8 **    bufArray,
                                     IN  Uint32      size,
                                     IN  Uint32      numBufs) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | procId |
| | Processor Identifier. | |

| | | |
|---|---|---|
| `IN` | `ChannelId` | `chnlId` |
| | Channel Identifier. | |
| `OUT` | `Char8 **` | `bufArray` |
| | Pointer to receive array of allocated buffers. | |
| `IN` | `Uint32` | `size` |
| | Size of each buffer. | |
| `IN` | `Uint32` | `numBufs` |
| | Number of buffers to allocate. | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Memory error occurred |
| `DSP_EFAIL` | General error from the GPP OS |

**Comments**

None.

**Constraints**

Processor and channel ids must be valid.

`bufArray` must be valid.

`numBufs` must be less than maximum limit.

**See Also**

```
LDRV_DATA_Initialize
LDRV_DATA_FreeBuffer
```

### 9.3.6 LDRV_CHNL_FreeBuffer

This function frees buffer(s) allocated by `LDRV_DATA_AllocateBuffer ()`. The pool configured for usage by the data driver is used for freeing the data buffers.

**Syntax**

```
DSP_STATUS LDRV_DATA_FreeBuffer (IN ProcessorId procId,
                                 IN ChannelId   chnlId,
                                 IN Char8 **    bufArray,
                                 IN Uint32      numBufs) ;
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `procId` |
| | Processor Identifier. | |
| `IN` | `ChannelId` | `chnlId` |

|  |  |  |
|---|---|---|
|  | Channel Identifier. |  |
| IN | `Char8 **` | `bufArray` |
|  | Pointer to array of buffers to be freed. |  |
| IN | `Uint32` | `numBufs` |
|  | Number of buffers to be freed. |  |

**Return Values**

|  |  |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Memory error occurred |
| `DSP_EFAIL` | General error from the GPP OS |

**Comments**

None.

**Constraints**

Processor and channel ids must be valid.

`bufArray` must be valid.

`numBufs` must be less than maximum limit.

**See Also**

```
LDRV_DATA_Initialize
LDRV_DATA_AllocateBuffer
```

### 9.3.7 LDRV_DATA_Request

This function sends an IO request on specified channel by calling the function ioRequest from corresponding data driver’s function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DATA_Request (ProcessorId dspId, ChannelId chnlId) ;
```

**Arguments**

|  |  |  |
|---|---|---|
| IN | `ProcessorId` | `dspId` |
|  | Identifier for the DSP |  |
| IN | `ChannelId` | `chnlId` |
|  | Identifier for the channel |  |

**Return Values**

|  |  |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Out of memory |

| | |
|---|---|
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

`LDRV_Initialize`

### 9.3.8 LDRV_DATA_Cancel

This function cancels pending IO on a channel by calling the function cancelIO from corresponding data driver's function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DATA_Cancel (ProcessorId dspId, ChannelId chnlId) ;
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP | |
| `IN` | `ChannelId` | `chnlId` |
| | Identifier for the channel | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Out of memory |
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

`LDRV_Initialize`

### 9.3.9 LDRV_DATA_Cancel

This function cancels pending IO on a channel by calling the function cancelIO from corresponding data driver's function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DATA_GetPoolId (ProcessorId    dspId,
                                ChannelId      chnlId,
                                   Uint32 *       poolId) ;
```

**Arguments**

```
IN       ProcessorId                 dspId
```

Identifier for the DSP

```
IN       ChannelId                   chnlId
```

Identifier for the channel

```
OUT      Uint32 *                    poolId
```

Placeholder for returning the pool ID.

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EFAIL | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

LDRV_Initialize () must be called before this function.

**See Also**

LDRV_Initialize

### 9.3.10 LDRV_DATA_Debug

This is a debug mode function. It prints the debug information for the data driver(s) towards specified target DSP.

**Syntax**

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

**Arguments**

```
IN       ProcessorId                 dspId
```

Identifier for the DSP

**Return Values**

None.

**Comments**

None.

**Constraints**

dspId must be valid.

**See Also**

`None.`

# 10 LDRV_DRV

This subcomponent encapsulates low-level driver synchronization between the GPP and the DSP over a physical link.

## 10.1 Dependencies

### 10.1.1 Subordinates

The DSP subcomponent is used by this subcomponent for interacting with the DSP.

## 10.2 Description

This subcomponent provides the driver initialization and synchronization services to LDRV_PROC. It passes all the requests to the actual link driver for the physical link, using its function pointer interface exported by the link driver.

Usage of function pointer interface ensures that multiple link drivers can be easily plugged into the system.

It determines the link driver to be used for each DSP through information obtained from the configuration.

## 10.3 Constants and Enumerations

### 10.3.1 DrvHandshake

Defines the types of handshake control actions.

**Definition**

```
typedef enum {
    DrvHandshakeSetup = 0,
    DrvHandshakeStart = 1,
    DrvHandshakeCompl = 2
} DrvHandshake ;
```

**Fields**

`DrvHandshakeSetup` Setup the handshaking between the processors.

`DrvHandshakeStart` Start the handshake process with the remote processor.

`DrvHandshakeCompl` Complete the handshaking with the remote processor.

**Comments**

None.

**Constraints**

None.

**See Also**

None.

## 10.4 Typedefs and Structures

### 10.4.1 LinkInterface

This structure defines the interface functions exported by the Link Driver.

**Definition**

```
struct LinkInterface_tag {
    FnLinkInitialize    initialize ;
    FnLinkFinalize      finalize ;
    FnLinkHandshake     handshake ;
#if defined (DDSP_DEBUG)
    FnLinkDebug         debug ;
#endif /* if defined (DDSP_DEBUG) */
} ;

typedef struct LinkInterface_tag LinkInterface ;
```

**Fields**

`initialize` Function pointer to `initialize` function for the Link Driver.

`finalize` Function pointer to `finalize` function for the Link Driver.

`handshake` Function pointer to the Link Driver function to setup, start and complete handshake.

`debug` Function pointer to the Link Driver function for printing debug information

**Comments**

None.

**See Also**

None.

### 10.4.2 LinkObject

Defines the link object for driver initialization and synchronization.

**Definition**

```
struct LinkObject_tag {
#if defined (DDSP_DEBUG)
    Char8           linkName [DSP_MAX_STRLEN] ;
    Char8           abbr [DSP_MAX_STRLEN]    ;
#endif /* if defined (DDSP_DEBUG) */
    LinkInterface * interface  ;
    Uint32          memEntry   ;
    Uint32          size       ;
    Uint32          numIps     ;
    Uint32          ipsTableId ;
} ;

typedef struct LinkObject_tag LinkObject ;
```

**Fields**

| | |
|---|---|
| `linkName` | Name of the link. |
| `abbr` | Abbreviation of the link name. |
| `interface` | Pointer to the interface table for the link. |
| `memEntry` | ID of the LINK mem information entry in the DSP memTable. |
| `size` | Size of the memory area used by the link. |
| `numIps` | Number of IPS objects used by the link. |
| `ipsTableId` | ID of the IPS table in the link driver object. |

**Comments**

None.

**See Also**

`None.`

## 10.5 API Definition

### 10.5.1 LDRV_DRV_Initialize

This function initializes the resources required by this module. It also calls the function *initialize* from the function pointer interface exported by the link driver attached to the specified DSP.

**Syntax**

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

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Out of memory |
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

`LDRV_Initialize`
`LDRV_DRV_Finalize`

### 10.5.2 LDRV_DRV_Finalize

This function releases the resources required by this module. It also calls the function *finalize* from the function pointer interface exported by all the link driver attached to the specified DSP.

**Syntax**

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

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | DSP ID of DSP for which the finalization must be performed | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EMEMORY` | Out of memory |

| | |
|---|---|
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

`LDRV_Initialize`
`LDRV_DRV_Initialize`

### 10.5.3 LDRV_DRV_Handshake

This function performs the necessary handshake (if required) for the link between the GPP and the target DSP by calling the *handshake* function from the corresponding link driver's function pointer interface.

**Syntax**

```
DSP_STATUS LDRV_DRV_Handshake (ProcessorId  dspId,
                               DrvHandshake hshkCtrl) ;
```

**Arguments**

| | | |
|---|---|---|
| `IN` | `ProcessorId` | `dspId` |
| | Identifier for the DSP | |
| `IN` | `DrvHandshake` | `hshkCtrl` |
| | Handshake control action to be executed. | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_ETIMEOUT` | Timed out during handshake with the DSP |
| `DSP_EFAIL` | General failure returned from GPP OS |

**Comments**

None.

**Constraints**

`LDRV_Initialize ()` must be called before this function.

**See Also**

`LDRV_Initialize`

### 10.5.4 LDRV_DATA_Debug

This is a debug mode function. It prints the debug information for the link driver towards specified target DSP.

**Syntax**

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

**Arguments**

```
IN       ProcessorId                dspId
```

Identifier for the DSP

**Return Values**

None.

**Comments**

None.

**Constraints**

`dspId` must be valid.

**See Also**

None.

# 11 DSP

This subcomponent provides interfaces to directly control and communicate with the target DSP.

## 11.1 Description

This subcomponent directly interacts with the hardware and provides access to the target DSP. It essentially abstracts the DSP from other subcomponents in acts as an abstraction for the DSP.

## 11.2 Typedefs and Structures

### 11.2.1 LinkMemInfo

This structure defines a memory information entry for the DSP.

**Definition**

```
typedef struct LinkMemInfo_tag {
    Uint32  entry       ;
    Uint32  physAddr    ;
    Uint32  dspVirtAddr ;
    Uint32  gppVirtAddr ;
    Uint32  size        ;
    Uint32  mapInGpp    ;
} LinkMemInfo ;
```

**Fields**

| | |
|---|---|
| `entry` | Entry number for the MEM record. |
| `physAddr` | Physical address |
| `dspVirtAddr` | Virtual address in DSP address space |
| `gppVirtAddr` | Virtual address in GPP address space |
| `size` | Indicates the size of memory entry |
| `mapInGpp` | Flag indicating whether DSP address is mapped to GPP address space. |

**Comments**

None.

**See Also**

`DspObject`

### 11.2.2 DspObject

This structure defines the context under which the DSP subcomponent works.

**Definition**

```
struct DspObject_tag {
#if defined (DDSP_DEBUG)
    Char8              dspName    [DSP_MAX_STRLEN] ;
#endif /* if defined (DDSP_DEBUG) */
    DspArch            dspArch                     ;
    DspInterface *     interface                   ;
    LoaderInterface  * loaderInterface             ;
    Bool               autoStart                   ;
    Char8              execName   [DSP_MAX_STRLEN] ;
    Uint32             resetVector                 ;
    Uint32             maduSize                    ;
    Uint32             endian                      ;
    Uint32             numMemEntries               ;
    LinkMemInfo *      memTable                    ;
    Bool                wordSwap                    ;
#if defined (CHNL_COMPONENT) || defined (MSGQ_COMPONENT)
    LinkObject *        linkObject                  ;
#endif /* if defined (CHNL_COMPONENT) || defined (MSGQ_COMPONENT) */
#if defined (CHNL_COMPONENT)
    Uint32              numDataDrivers              ;
    DataObject *        dataObjects                 ;
#endif /* if defined (CHNL_COMPONENT) */
#if defined (MSGQ_COMPONENT)
    Uint32              mqtId                       ;
#endif /* if defined (MSGQ_COMPONENT) */
#if defined (DDSP_PROFILE)
    DspStats *          dspStats                    ;
#endif /* if defined (DDSP_PROFILE) */
} ;
```

**Fields**

`dspName` Name of the DSP

`dspArch` Architecture of the Dsp.

`interface` The function pointer interface to access the services of the DSP subcomponent for this DSP.

`loaderInterface` The function pointer interface to access the services of the loader subcomponent for this DSP.

`autoStart` Auto start flag for the DSP.

`execName` Name of default DSP executable.

`resetVector` Reset vector address for the dsp.

`maduSize` MADU size of the DSP.

`endian` Endianism of the DSP.

`numMemEntries` Number of MEM entries.

`memTable` Table of MEM entries.

`wordSwap` Indicates whether word-swap is enabled for the DSP MEM.

`linkObject` Pointer to link object for the DSP.

`numDataDrivers` Array of data driver objects supported for the DSP.

`dataObjects` Number of data drivers supported for the DSP.

`mqtId` ID of the MQT used by the DSP.

`dspStats` Profiling information related to the target DSP.

**Comments**

None.

**See Also**

```
DspInterface
LinkMemInfo
LinkObject
DataObject
DspStats
```

### 11.2.3 DspInterface

This structure defines the interface functions exported by the DSP subcomponent.

**Definition**

```
typedef struct DspInterface_tag {
    FnDspSetup              setup            ;
    FnDspInitialize         initialize       ;
    FnDspFinalize           finalize         ;
    FnDspStart              start            ;
    FnDspStop               stop             ;
    FnDspIdle               idle             ;
    FnDspEnableInterrupt    enableInterrupt  ;
    FnDspDisableInterrupt   disableInterrupt ;
    FnDspInterrupt          dspInterrupt     ;
    FnDspClearInterrupt     clearInterrupt   ;
    FnDspRead               read             ;
    FnDspWrite              write            ;
    FnDspControl            control          ;
#if defined (DDSP_PROFILE)
    FnDspInstrument         instrument       ;
#endif /* if defined (DDSP_PROFILE) */
#if defined (DDSP_DEBUG)
    FnDspDebug              debug            ;
#endif /* if defined (DDSP_DEBUG) */
} DspInterface ;
```

**Fields**

`setup` Function pointer to `setup` function for the DSP.

`initialize` Function pointer to `initialize` function for the DSP.

`finalize` Function pointer to `finalize` function for the DSP.

`start` Function pointer to `start` function for the DSP.

`stop` Function pointer to `stop` function for the DSP.

`idle` Function pointer to `idle` function for the DSP.

`enableInterrupt` Function pointer to `enableInterrupt` function for the DSP.

`disableInterrupt` Function pointer to `disableInterrupt` function for the DSP.

`dspInterrupt` Function pointer to `dspInterrupt` function for the DSP.

`clearInterrupt` Function pointer to `clearInterrupt` function for the DSP.

`read` Function pointer to `read` function for the DSP.

| | |
|---|---|
| `write` | Function pointer to `write` function for the DSP. |
| `control` | Function pointer to `control` function for the DSP. |
| `instrument` | Function pointer to `instrument` function for the DSP. |
| `debug` | Function pointer to `debug` function for the DSP. |

### Comments

None.

### See Also

`DspObject`

## 11.3 API Definition

### 11.3.1 DSP_Setup

This function performs necessary operations to make the DSP reachable from the GPP.

**Syntax**

```
DSP_STATUS DSP_Setup (ProcessorId dspId, DspObject *  dspObj) ;
```

**Arguments**

```
IN      ProcessorId              dspId
```

Identifier for the DSP

```
IN      DspObject *              dspObj
```

Pointer to object containing context information for DSP

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | General failure returned from GPP OS |

**Comments**

This function initializes the necessary hardware abstraction layer. It sets up the ARM port interface and the DSP boot configuration.

**Constraints**

None.

**See Also**

DspObject
DSP_Initialize

### 11.3.2 DSP_Initialize

This function resets the DSP and initializes peripherals required by the DSP (for example, MMU entries).

**Syntax**

```
DSP_STATUS DSP_Initialize (ProcessorId dspId, DspObject *  dspObj) ;
```

**Arguments**

```
IN      ProcessorId              dspId
```

Identifier for the DSP

```
IN      DspObject *              dspObj
```

Pointer to object containing context information for DSP

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

This function initializes the DSP. For example:

1. Resets the DSP
2. Sets up the MMU table
3. Sets up the clock divisors.

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

DspObject
DSP_Setup
DSP_Finalize

### 11.3.3 DSP_Finalize

This function idles the DSP.

**Syntax**

```
DSP_STATUS DSP_Finalize (ProcessorId dspId, DspObject *  dspObj)
```

**Arguments**

IN ProcessorId dspId

Identifier for the DSP

IN DspObject * dspObj

Pointer to object containing context information for DSP

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

None.

#### Constraints

`DSP_Setup ()` must be called before calling this function.

#### See Also

```
DspObject
DSP_Setup
DSP_Initialize
```

### 11.3.4 DSP_Start

This function starts the DSP run from the specified address.

#### Syntax

```
DSP_STATUS DSP_Start (ProcessorId  dspId,
                      DspObject *  dspObj,
                      Uint32       dspAddr)
```

#### Arguments

| | | |
|---|---|---|
| IN | `ProcessorId` | `dspId` |
| | Identifier for the DSP | |
| IN | `DspObject *` | `dspObj` |
| | Pointer to object containing context information for DSP | |
| IN | `Uint32` | `dspAddr` |
| | Location to start the execution on the DSP | |

#### Return Values

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EINVALIDARG` | Invalid `dspId` or `dspObj` specified |
| `DSP_EFAIL` | `DSP_Setup ()` was not called before calling this function |

#### Comments

None.

#### Constraints

`DSP_Setup ()` must be called before calling this function.

#### See Also

```
DspObject
DSP_Initialize
DSP_Stop
```

### 11.3.5 DSP_Stop

This function stops execution on the DSP.

#### Syntax

```
DSP_STATUS DSP_Stop (ProcessorId dspId, DspObject *  dspObj) ;
```

#### Arguments

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP | |

#### Return Values

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

#### Comments

This function configures the ARM port interface to put the DSP into a self loop and then puts the DSP into a self loop.

#### Constraints

DSP_Setup () must be called before calling this function.

#### See Also

DspObject
DSP_Initialize
DSP_Start

### 11.3.6 DSP_Idle

This function idles the DSP.

#### Syntax

```
DSP_STATUS DSP_Idle (ProcessorId dspId, DspObject *  dspObj) ;
```

#### Arguments

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP | |

#### Return Values

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |

| | |
|---|---|
| `DSP_EFAIL` | `DSP_Setup ()` was not called before calling this function |

**Comments**

This function writes the idle code onto the DSP and starts its execution.

**Constraints**

`DSP_Setup ()` must be called before calling this function.

**See Also**

```
DspObject
DSP_Setup
DSP_Stop
```

### 11.3.7 DSP_EnableInterrupt

This function enables the specified interrupt for communication with DSP.

**Syntax**

```
DSP_STATUS DSP_EnableInterrupt (ProcessorId        dspId,
                                DspObject *        dspObj,
                                InterruptObject *  intInfo) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | `ProcessorId` | `dspId` |
| | Identifier for the DSP | |
| IN | `DspObject *` | `dspObj` |
| | Pointer to object containing context information for DSP | |
| IN | `InterruptObject *` | `intInfo` |
| | Pointer to an object containing interrupt information | |

**Return Values**

| | |
|---|---|
| `DSP_SOK` | Operation completed successfully |
| `DSP_EINVALIDARG` | Invalid `dspId` or `dspObj` specified |
| `DSP_EFAIL` | `DSP_Setup ()` was not called before calling this function |

**Comments**

None.

**Constraints**

`DSP_Setup ()` must be called before calling this function.

**See Also**

```
DspObject
```

```
InterruptObject
DSP_DisableInterrupt
DSP_Interrupt
DSP_ClearInterrupt
```

### 11.3.8 DSP_DisableInterrupt

This function disables the specified interrupt for communication with DSP.

**Syntax**

```
DSP_STATUS DSP_EnableInterrupt (ProcessorId        dspId,
                                DspObject *        dspObj,
                                InterruptObject *  intInfo) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP | |
| IN | InterruptObject * | intInfo |
| | Pointer to an object containing interrupt information | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

None.

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

```
DspObject
InterruptObject
DSP_EnableInterrupt
DSP_Interrupt
DSP_ClearInterrupt
```

### 11.3.9 DSP_Interrupt

This function sends the specified interrupt to the DSP.

**Syntax**

```
DSP_STATUS DSP_Interrupt (ProcessorId        dspId,
                          DspObject *        dspObj,
                                          InterruptObject *  intObj,
                                          Pvoid              arg) ;
```

**Arguments**

IN      `ProcessorId`                `dspId`

Identifier for the DSP

IN      `DspObject *`                `dspObj`

Pointer to object containing context information for DSP

IN      `InterruptObject *`          `intInfo`

Pointer to an interrupt object containing the information regarding the interrupt to be sent to the DSP

IN OPT  `Pvoid`                      `arg`

Pointer to a value to send with the interrupt.

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid `dspId` or `dspObj` specified |
| DSP_EFAIL | `DSP_Setup ()` was not called before calling this function |

**Comments**

None.

**Constraints**

`DSP_Setup ()` must be called before calling this function.

**See Also**

```
DspObject
InterruptObject
DSP_EnableInterrupt
DSP_DisableInterrupt
DSP_ClearInterrupt
```

### 11.3.10 DSP_ClearInterrupt

This function clears an interrupt received from the DSP side on to the GPP side.

**Syntax**

```
DSP_STATUS DSP_ClearInterrupt (ProcessorId        dspId,
                               DspObject *        dspObj,
                               InterruptObject *  intObj) ;
                               Pvoid              retVal) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP | |
| IN | InterruptObject * | intInfo |
| | Pointer to an interrupt object containing the information regarding the interrupt to be sent to the DSP | |
| OUT | Pvoid | retVal |
| | Interrupt value present before clearing the interrupt | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

None.

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

DspObject
InterruptObject
DSP_EnableInterrupt
DSP_DisableInterrupt
DSP_Interrupt

### 11.3.11 DSP_Read

This function reads data from the DSP memory space.

**Syntax**

```
DSP_STATUS DSP_Read (ProcessorId  dspId,
                     DspObject *  dspObj,
                     Uint32       dspAddr,
                     Endianism    endianInfo,
                     Uint32 *     numBytes,
                     Uint8 *      buffer) ;
```

**Arguments**

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

Identifier for the DSP

IN `DspObject *` `dspObj`

Pointer to object containing context information for DSP

IN `Uint32` `dspAddr`

Address to read

IN `Endianism` `endianInfo`

Specifies the memory endianism of the target memory

OUT `Uint32 *` `numBytes`

IN/OUT argument to specify the number of bytes to read and upon return contain the actual number of bytes read

OUT `Uint8 *` `buffer`

Buffer to hold the read data

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

This function performs the endianism conversion required

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

DspObject
DSP_Write

### 11.3.12 DSP_Write

This function writes data into the DSP memory space.

**Syntax**

```
DSP_STATUS DSP_Write (ProcessorId dspId,
                      DspObject * dspObj,
                      Uint32      dspAddr,
                      Endianism   endianInfo,
                      Uint32      numBytes,
                      Uint8 *     buffer) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Identifier for the DSP | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP | |
| IN | Uint32 | dspAddr |
| | Address to write the data | |
| IN | Endianism | endianInfo |
| | Specifies the memory endianism of the target memory | |
| IN | Uint32 | numBytes |
| | Number of bytes to write | |
| IN | Uint8 * | buffer |
| | Buffer containing the data to write | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |
| DSP_EFAIL | DSP_Setup () was not called before calling this function |

**Comments**

This function performs the necessary endianism conversion on the data before writing it to the target memory.

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

DspObject
DSP_Write

### 11.3.13 DSP_Control

Hook for performing device dependent control operation.

**Syntax**

```
DSP_STATUS DSP_Control (IN  ProcessorId dspId,
                        IN  DspObject * dspObj,
                        IN  Int32       cmd,
                        OPT Pvoid       arg) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | ProcessorId | dspId |
| | Processor Id | |
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP. | |
| IN | Int32 | cmd |
| | Command id. | |
| IN | Pvoid | Arg |
| | Optional argument for the specified command. | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid dspId or dspObj specified |

**Comments**

This function performs the necessary endianism conversion on the data before writing it to the target memory.

**Constraints**

DSP_Setup () must be called before calling this function.

**See Also**

DspObject
DSP_Write

### 11.3.14 DSP_Instrument

Gets the instrumentation information related to the specified DSP object.

**Syntax**

```
DSP_STATUS DSP_Instrument (DspObject * dspObj, DspStats * retVal) ;
```

**Arguments**

| | | |
|---|---|---|
| IN | DspObject * | dspObj |
| | Pointer to object containing context information for DSP. | |
| OUT | DspStats * | retVal |
| | Placeholder to return the instrumentation information. | |

**Return Values**

| | |
|---|---|
| DSP_SOK | Operation completed successfully |
| DSP_EINVALIDARG | Invalid argument(s). |

### Comments

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

### Constraints

`DSP_Setup ()` must be called before calling this function.

### See Also

`None.`

## 11.3.15 DSP_Debug

Prints debug information of the specified DSP object.

### Syntax

```
Void DSP_Debug (IN DspObject * dspObj) ;
```

### Arguments

```
IN       DspObject *                dspObj
```

Pointer to object containing context information for DSP.

### Return Values

None.

### Comments

This function is defined only for debug build.

### Constraints

`None.`

### See Also

`None.`

# 12 HAL

The Hardware Abstraction Layer provides a low-level layer for access and control of hardware specific modules to the sub-components within the *DSPLINK* link driver layer.

The services provided by the HAL subcomponent are used by the DSP sub-component.

## 12.1 Dependencies

### 12.1.1 Subordinates

None.

## 12.2 Description

The implementation of the hardware abstraction layer is specific to the target platform. The hardware modules to be abstracted vary based on the platform supported by *DSPLINK*.

This document does not provide details for a specific hardware abstraction layer for any platform.

# 13 IPS

The Inter-processor signaling (IPS) subcomponent component provides basic services required by the data driver and Message Queue Transport components for transferring data buffers and messages between the processors. It abstracts the platform-specific details by providing standard services to the upper layer.

## 13.1 Dependencies

### 13.1.1 Subordinates

None.

## 13.2 Description

The IPS subcomponent provides the upper layers with the service to register an event from the GPP, about which is wishes to be notified. On receiving the event from the GPP, the IPS subcomponent provides information about the event to the registered subcomponent.

This component uses the services provided on the hardware platform. It provides APIs, which are used by upper layers to establish communication amongst peers at that level.

The design of IPS components is specific to the physical link for the target platform. This document does not provide details for a specific IPS component for any platform.

## 13.3 Typedefs and Structures

### 13.3.1 FnIpsInitialize

This type defines the signature of function that initializes an Inter Processor Signaling component.

**Definition**

```
typedef DSP_STATUS (*FnIpsInitialize) (IN ProcessorId dspId,
                                       IN Uint32      ipsId) ;
```

**Comments**

The function for initialization of the IPS is configured within the *DSPLINK* static configuration, and called by the DRV component during its initialization.

**Constraints**

None.

**See Also**

```
FnIpsFinalize
```

### 13.3.2 FnIpsFinalize

This type defines the signature of function that finalizes an Inter Processor Signaling component.

**Definition**

```
typedef DSP_STATUS (*FnIpsFinalize) (IN ProcessorId dspId,
                                     IN Uint32      ipsId) ;
```

**Comments**

The function for finalization of the IPS is configured within the *DSPLINK* static configuration, and called by the DRV component during its finalization.

**Constraints**

None.

**See Also**

```
FnIpsInitialize
```

### 13.3.3 IpsObject

This structure defines the Inter Processor Signaling object.

**Definition**

```
struct IpsObject_tag {
#if defined (DDSP_DEBUG)
    Char8           ipsName [DSP_MAX_STRLEN] ;
    Char8           abbr [DSP_MAX_STRLEN]    ;
#endif /* if defined (DDSP_DEBUG) */
    FnIpsInitialize initialize      ;
    FnIpsFinalize   finalize        ;
#if defined (CHNL_COMPONENT)
    Uint32          irpQueueLength  ;
    Uint32          irpSize         ;
#endif /* if defined (CHNL_COMPONENT) */
    Uint32          memEntry        ;
    Uint32          size            ;
    Uint32          arg1            ;
    Uint32          arg2            ;
} ;

typedef struct IpsObject_tag IpsObject ;
```

**Fields**

| | |
|---|---|
| `ipsName` | Name of the IPS |
| `abbr` | Abbreviation of the IPS name. |
| `initialize` | Initialize function for the IPS. |
| `finalize` | Finalize function for the IPS. |
| `irpQueueLength` | Length of the IRP queue within the IPS. |
| `irpSize` | Size of the I/O Request Packet used by the IPS. |
| `memEntry` | ID of the LINK mem information entry in the DSP memTable |
| `size` | Size of memory area configured for the IPS component. |
| `arg1` | First argument specific to the IPS. |
| `arg2` | Second argument specific to the IPS. |

**Comments**

None.

**See Also**

None.

# 14 MQT

This subcomponent provides functions to transfer messages between the GPP and a specific DSP over a physical link. It provides functionality to locate & release the remote message queues on the DSP, and transfer messages between the GPP and DSP.

## 14.1 Dependencies

### 14.1.1 Subordinates

The IPS subcomponent is used by this subcomponent for interacting with the remote processor.

## 14.2 Description

The MQT sub-component defines the abstract interface that the Message Queue Transports (MQTs) for specific platforms must implement. There may be multiple MQT implementations for a single platform, based on the physical connection used for connecting the two processors. However, only a single MQT each can be configured at a time for communication between the GPP and a DSP.

The MQT plugs into the `LDRV_MSGQ` component and provides services to send & receive messages to & from the remote processor, and locate & release message queues on the remote processor.

Usage of function pointer interface ensures that multiple MQTs can be easily plugged into the system.

The standard interface for the MQT component is provided within the Messaging Design document [Ref. 2].

The design of MQT components is specific to the physical link for the target platform. This document does not provide details for a specific MQT component for any platform.

# 15 LDRV_POOL

The POOL component provides services to allocate and free data buffers and messages, which can be transferred between the processors.

## 15.1 Dependencies

### 15.1.1 Subordinates

None.

## 15.2 Description

The LDRV_POOL subcomponent defines the abstract interface that the different POOLs must implement. It provides the connection between the PMGR_POOL subcomponent and the different pool implementations.

This sub-component also contains implementations of specific pools for the different types of data and message transfer supported by the system.

Usage of function pointer interface ensures that multiple POOLs can be easily plugged into the system.

The configuration of pool objects in the system is maintained by this component.

The standard interface for the LDRV_POOL component is provided within the POOL Design document [Ref. 3].

The design of example POOL components based on fixed-size buffers is available within the Buffer Pools Design document [Ref. 4]. This document does not provide details for a specific POOL implementation.

# 16 DSP-side

The DSP-side of *DSPLINK* provides functionality for transferring data buffers and messages between the GPP and DSP.

## 16.1 Dependencies

### 16.1.1 Subordinates

None.

## 16.2 Description

The DSP-side of DSPLINK is specific to the platform being supported.

The design of the DSP-side components is specific to the physical link for the target platform. This document does not provide details for the DSP-side design for any platform. The details of DSP-side design are available in the design document for the specific link driver. For example, design details of the DSP-side for the Zero Copy Link Driver are available in the Zero Copy Link Driver design document [Ref. 5].