DSP/BIOS™ LINK

PROGRAMMER'S GUIDE

LNK 161 USR

Version 1.65

This page has been intentionally left blank.

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

1.1 Purpose & Scope

This document is a Programmer's Guide for DSP/BIOS™ LINK. It gives information about the various concepts and components in DSP/BIOS™ LINK along with their features, concepts and programming tips.

The document is targeted at the application developers of DSP/BIOS™ LINK.

1.2 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
O	This bullet indicates important information. Please read such text carefully.
q	This bullet indicates additional information.

1.3 References

1.	UserGuide	DSP/BIOS™ LINK User Guide

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.

2 Getting started with writing applications

To write applications using DSP/BIOS™ LINK, it is important to select the most appropriate DSPLINK modules to be used as per the system and application design. Based on the application's requirements, all or a subset of the features provided by DSPLINK can be used.

The following section describes simple application scenarios. This information can be used to select the modules providing the most optimum performance and footprint for the system, while still giving the simplest application design.

2.1 Generic information

1. PROC component is always required for all applications using DSP/BIOS LINK. This component provides the basic functionality to setup, boot-load, control and communicate with the processors in the system.
2. To understand how the APIs for each component are used, the GPP and DSP-side of the sample applications provided with each DSPLINK release can be used as reference.
3. In addition, a description of the setup, execution and shutdown control flow for each component is given within the User Guide. The designs of the sample applications are also detailed in the User Guide.

2.1.1 Component features

Te following information may be useful to decide which DSPLINK component is best suited to meet the application's messaging and data transfer requirements:

NOTIFY

NOTIFY component may be used for messaging/data transfer if:

1. Only 32-bit information needs to be sent between the processors.
2. Prioritization of notifications is required. For example, one low priority event (e.g. 30) can be used for sending buffer pointers. A higher priority event (e.g. 5) can be used to send commands.
3. The notification is to be sent infrequently. If multiple notifications for the same event are sent very quickly in succession, each attempt to send a specific event spins, waiting till the previous event has been read by the other processor. This may result in inefficiency.
4. Multiple clients need to be able to register for the same notification event. When the event notification is received, the same notification with payload is broadcast to all clients registered for that event.

MSGQ

MSGQ component may be used for messaging/data transfer if:

1. Application requires single reader and multiple writers.
2. More than 32-bit information needs to be sent between the processors using application-defined message structures.
3. Variable sized messages are required.
4. Reader and writer operate on the same buffer sizes.
5. Messages need to be sent frequently. In this case, the messages are queued and there is no spin-wait for the previous event to be cleared.
6. The ability to wait when the queue is empty is desired. This is inbuilt within the MSGQ protocol, and no extra application code is required. If `MSGQ_get ()` is called on an empty queue, it waits till a message is received. If NOTIFY is used, the application must register the callback, or wait on a semaphore that is posted by the application's notification callback function.
7. It is desired to have the ability to move the Message Queue between processors. In this case, the `MSGQ_locate ()` on other processors internally takes care of locating the queue, and the application code sending messages to this queue does not need to change.
8. It is desired to also have DSP-DSP communication. In this case, Message Queue component in DSP/BIOS allows usage of different Message Queue Transport modules independent of DSPLINK to communicate between DSPs.

MPLIST

MPLIST component may be used for messaging/data transfer if:

1. Application requires multiple writers and multiple readers.
2. The application wishes to perform out-of-order processing on received packets. This is not possible with MSGQ or with NOTIFY. With MPLIST, the

reader may traverse the shared list and choose any buffer within the list to be removed.

3. Making a specific buffer as high priority is desired. The sender to an MPLIST can make a specific buffer/message as high priority by pushing it to the head of the queue instead of placing it at the end of the queue. APIs are provided to traverse the list and insert element before any specified element in the queue.
4. Inbuilt notification is not required. If the application desires flexibility in when notification is to be sent/received, MPLIST module can be used. The application may use NOTIFY module to send & receive notifications as per its specific requirements. This may result in better performance and lesser number of interrupts, tuned to application's requirements. However, the disadvantage is that additional application code needs to be written for notification, which is present inherently within MSGQ component.
5. Reader and writer operate on the same buffer sizes.
6. More than 32-bit information needs to be sent between the processors using application-defined message structures.
7. Variable sized messages/data buffers are required.
8. Messages/data buffers need to be sent frequently. In this case, the messages are queued directly. No notification/spin-wait for notification is performed.

CHNL

CHNL component may be used for data transfer if:

1. Single reader and single writer are required.
2. Fixed size data buffers are required.
3. Reader and writer operate on the same buffer sizes.
4. Existing SIO drivers for other peripherals are to be used in conjunction with the DSPLINK driver for GPP-DSP communication. In such scenarios, SIO provides a standard means of communication and inter-operability.
5. Simple synchronized data streaming is required. For such requirements, CHNL module provides a simple issue-reclaim protocol. The application only needs to issue empty/full buffers on both processors, and these get exchanged when buffers are available on both processors on the same channel. If buffer is not available, inbuilt wait & notification is available when attempt is made to reclaim the buffer.
6. Multiple buffers can be easily queued for better performance.

RingIO

RingIO component may be used for messaging & data transfer if:

1. Single reader and single writer are required.
2. Data, as well as attributes/messages associated with data are to be sent & received.
3. Writer and reader need to execute independently of each other. The size of buffer used by writer may be different from the buffer size needed by reader.

4. The buffer sizes for acquire and release for writer & reader do not need to match. Writer/reader may choose to release lesser buffer size than was acquired, or may choose to acquire more data before releasing any data.
5. Applications have different notification needs. The application can minimize interrupts by choosing the most appropriate type of notification based on watermark.
6. It is desired to have the capability to cancel unused data that was acquired but not released.
7. It is desired to flush the contents of the ring buffer to clear the ring buffer of released data. For example, when ongoing media file play is stopped and new media file is to be streamed and decoded.

The following sections give examples of possible application scenarios and suggest design and DSPLINK components to be used for each.

2.2 Static buffer system with minimal control communication with the DSP

2.2.1 Application requirements:

1. Boot-load the DSP
2. Statically reserve a region of memory to be shared with DSP. The complete system is static.
3. GPP and DSP may need to infrequently ping each other with some control information.

2.2.2 Suggested design

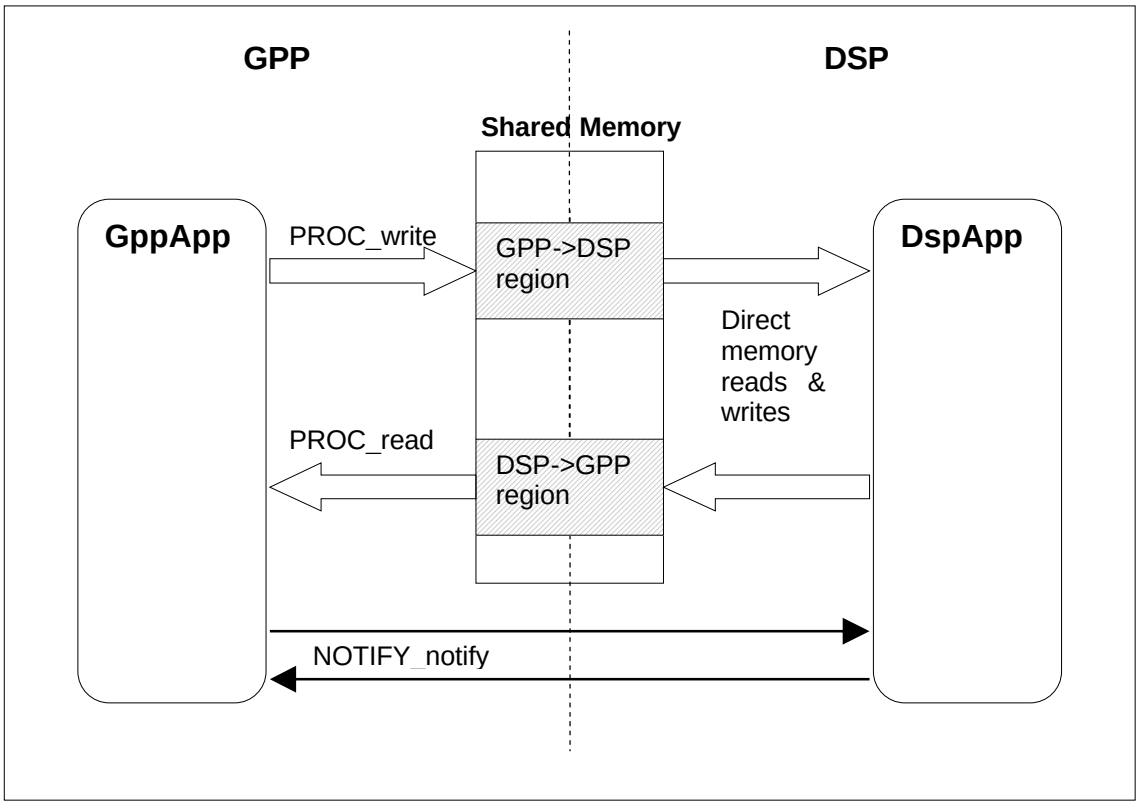

Figure 1. App scenario: Static buffer system with minimal control communication with the DSP

1. PROC module is used to boot-load the DSP. The DSP executable is present in the GPP file system.
2. Two regions of memory can be statically reserved (at compile time) through the DSPLINK dynamic configuration file (`CFG_<PLATFORM>.c`). On DSP-side, a similar configuration needs to be done within TCF file to reserve the memory. One region of memory can be used for GPP->DSP transfers, and the other for DSP->GPP transfers. Since the memory is statically reserved, both GPP and DSP are aware of their start addresses and sizes.
3. NOTIFY module can be used to send 32-bit control messages between the GPP and DSP.

2.2.3 DSP/BIOS LINK components used

1. PROC
2. NOTIFY

2.3 Dynamic buffer system with minimal control communication with the DSP

2.3.1 Application requirements:

1. Boot-load the DSP
2. Be able to dynamically allocate and free regions of memory to be shared with DSP. For example, this may be needed if same DSP executable is to be used with different GPP applications having different buffer size requirements.

3. Buffer requirements for each application are limited. For example, each application just needs to allocate one or two buffers during setup phase, and after this, the same buffers are used directly by GPP and DSP.
4. GPP and DSP may need to infrequently ping each other with some control information.

2.3.2 Suggested design

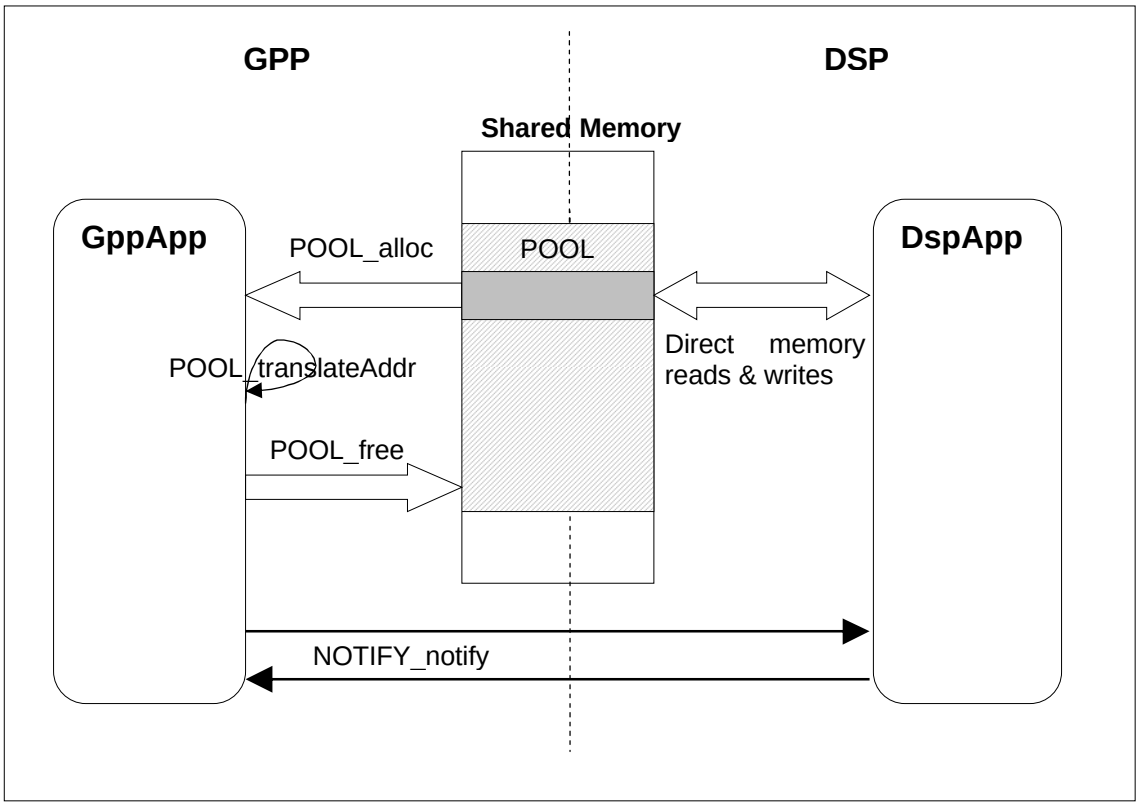

Figure 2. App scenario: Dynamic buffer system with minimal control communication with the DSP

1. PROC module is used to boot-load the DSP. The DSP executable is present in the GPP file system.
2. A POOL is opened with a configuration of the sizes of buffers to be shared between the GPP and DSP.
3. Buffers are allocated from the POOL as required by the GPP or DSP during setup phase of the application.
4. If allocated on GPP, the buffer address received from `POOL_alloc` can be translated to DSP address space to get the corresponding DSP address of the same buffer using `POOL_translateAddr`.
5. NOTIFY module can be used to send the 32-bit buffer addresses (or other 32-bit control information) between the GPP and DSP.
6. If the buffer is allocated on DSP-side, the DSP address received on the GPP can be translated using `POOL_translateAddr`.

7. The buffers are now used by GPP and DSP for sending/receiving data. NOTIFY module can be used to inform the processors when data is available/freed in the buffers.
8. If both GPP and DSP may have to simultaneously access the pool buffers, and mutually exclusive access is to be provided to the buffers, the MPCS module can be optionally used to protect access to the buffers.

2.3.3 DSP/BIOS LINK components used

1. PROC
2. NOTIFY
3. POOL
4. MPCS (optional)

2.4 Multiple buffers to be sent between GPP and DSP

2.4.1 Application requirements:

1. Boot-load the DSP
2. Be able to dynamically allocate and free regions of memory to be shared with DSP.
3. Multiple buffers are required by each application. The buffers may be allocated and freed at run-time. The buffers need to be sent between GPP and DSP during execution phase.

2.4.2 Suggested design

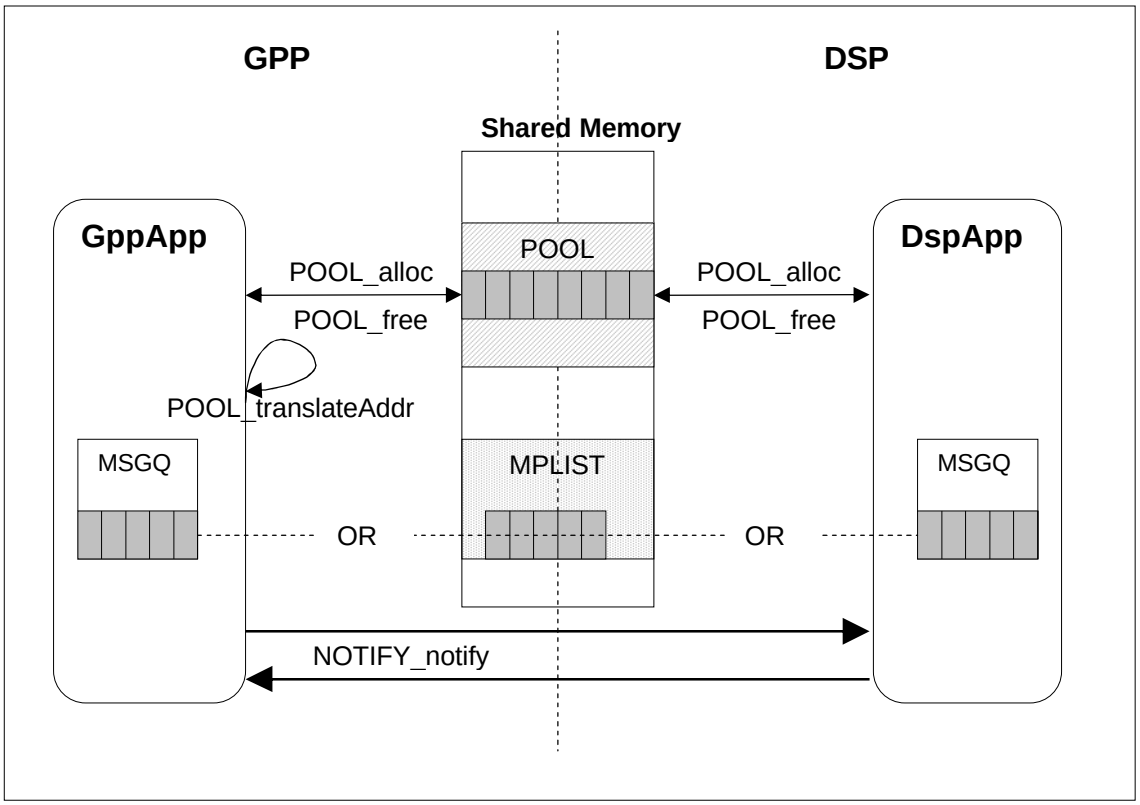

Figure 3. App scenario: Multiple buffers to be sent between GPP and DSP

1. PROC module is used to boot-load the DSP. The DSP executable is present in the GPP file system.
2. A POOL is opened with a configuration of the sizes of buffers to be shared between the GPP and DSP.
3. Buffers are allocated from the POOL as required by the GPP or DSP.
4. If allocated on GPP, the buffer addresses received from `POOL_alloc` can be translated to DSP address space to get the corresponding DSP addresses of the same buffers using `POOL_translateAddr`.
5. If the buffers are to be sent infrequently, the NOTIFY module can be used to send the 32-bit buffer addresses (or other 32-bit control information) between the GPP and DSP. If one processor may need to send multiple buffers to the other processor in one shot, either MSGQ or MPLIST module can be used. If additional information (e.g. buffer attributes) is required to be associated with the data buffer, a message structure can be defined that has these attributes, and MSGQ or MPLIST component can be used to send the message to the other processor.
6. If the buffers are allocated on DSP-side, the DSP addresses received on the GPP can be translated using `POOL_translateAddr`.

2.4.3 DSP/BIOS LINK components used

1. PROC
2. NOTIFY / MSGQ / MPLIST
3. POOL

3 PROC

3.1 Overview

The PROC module provides functionality to setup, boot-load, control, and communicate with the processors in the system. The master processor in the system is responsible for all control activities on the slave processors in the system. For example, in an SoC such as Davinci, the ARM processor is the master and the DSP is the slave.

The specific services provided by the PROC module are:

1. Setup and destroy the DSP/BIOS LINK driver.
2. Attach to and detach from a specific processor. Every process in the system wishing to communicate with a specific processor must do this to gain access to the processor.
3. Load a DSP executable on the target processor. This executable is present within the GPP file system.
4. Start execution of the DSP executable on the target processor.
5. Stop execution of the target processor
6. Write to and read from DSP memory

7. Get the current state of the PROC component. This indicates the last successful state transition for the DSP. It does not return the actual run-time state of the DSP.
8. Perform platform-specific control activities with the target processor.

3.2 Configuration and changing system memory map

The PROC module can be configured as part of the dynamic configuration.

- Instances of the LINKCFG_Dsp object contain all configuration information for each DSP in the system. The following information in this object most often needs to be customized by the system developer:
    - Number of memory entries in the memory map for DSP, which is visible to the GPP.
    - In case the CPU frequency for the DSP on the user platform is different from the BIOS-set value, this needs to be set as part of this object. On platforms like OMAP3530 and DM6467, the LSP allows querying of DSP clock rate. In these platforms, the LSP is queried to get the default DSP clock rate when default -1 is specified.
    - If the DSP address of the reset-vector memory entry has been changed, this needs to be reflected in the DSP object as well. The resume address is some number of bytes after the reset vector, and hence this needs to be changed as well.
- The memory map of the platform needs to be configured as part of the LINKCFG_MemEntry memory table.
    - By default, DSPLINK configures 1 MB of shared memory and 1MB of memory for loading the DSP code/data. If the system requires more memory, this needs to be modified/added in the memory table.
    - If the application wishes to reserve any additional memory to be used with PROC_read and PROC_write APIs to read from and write into DSP memory, this must be done by making additional memory entries within the memory table.
    - As per your application requirement, you can either add or remove memory entries. The number of memory entries must be correspondingly updated in LINKCFG_dspObject: field MEMENTRIES which indicates number of configured memory entries.
    - The fields in the memory entry that are usually changed are the physical address, DSP virtual address and size of the memory region.
    - To match the changes in the memory table, the DSP-side application's TCF file must be modified to indicate this information to DSP/BIOS configuration.
    - This information also needs to be conveyed to the GPP-side operating system to ensure that it does not place any of its code/data in this reserved region. On MVL Linux, this is done by specifying MEM=<> parameter in the boot args. The method to do this varies based on the GPP-side operating system. For more information on this, please refer to the platform-specific Install guide document available with the release.

- The shared field is used to decide whether mapping is required for ARM-side. If ARM-side mapping is required, set the shared field to TRUE. For example: In case of pool, shared memory etc. shared field should be TRUE. If ARM-side mapping is not required, set the shared field to FALSE. For example: In case of internal DSP regions (e.g. L4 core of OMAP2530 and OMAP3530), ARM-side mapping is not required. So set the shared field to FALSE. If any mappings are unnecessarily enabled, ARM-side can run out of virtual memory space.

- The physical interrupts to be used by the system for IPC and application use are configured within the LINKCFG_Ips instance.
  - The default configuration contains IPS configuration for one or more IPS instances. If the number of IPS instances is modified, this needs to be updated within the corresponding LINKCFG_LinkDrv instance.
  - The DSP-side interrupt vector number to be used for the ARM->DSP interrupts can be configured as per the system requirements to ensure that it does not clash with the other system usage.
  - The poll value indicates the number of cycles for which the IPS polls waiting for previous event to be cleared. If specified as -1, this wait is infinite. By specifying a value tuned to the application's requirements, error handling for DSP crash/block scenarios can be done by the application.
  - An IPS can be configured to work either in both GPP->DSP and DSP->GPP directions, or in one of the two. This can be configured based on availability of physical interrupts between the GPP and DSP.

3.2.1 Making configuration changes

For making configuration changes, it is very simple to update existing default configuration, instead of keeping the application's own copy of the configuration. This can be done by updating existing configuration prior to PROC_setup call. Using this method allows the application to remain independent of changes in configuration, and makes it easier to have platform-independent application code.

Example of runtime change of existing configuration:

```
/* Extern declaration to default configuration object in  (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

/* Increase maximum Message Queues to 32. */
LINKCFG_config.gppObject->maxMsgqs = 32 ;

        ...
        ...

/* Initialize and configure the DSPLink driver */
status = PROC_setup (&LINKCFG_config) ;
```

3.2.2 Making configuration changes to set the task priority and stack size

DSP/BIOS Link supports the task mode where the application can be configured for the TSK or SWI mode for the existing SWI functions of ZCPYMQT and ZCPYDATA. If

MPCS protection is TSK-base, then DSPLink MSGQ and CHNL drivers will use tasks on DSP-side.

In TSK based systems, application writer can set the task priority and the task stack size for ZCPYMQT task. This can be done by updating

- Argument 1 for Task priority
- Argument 2 FOR Task stack size

```
STATIC LINKCFG_Mqt  LINKCFG_mqtObjects [] =
    {
        "ZCPYMQT",              /* NAME           : Name of the Message Queue
Transport */
         …
         …
         …
        12,                     /* ARGUMENT1      : First MQT-specific
argument */
        0x2048                  /* ARGUMENT2      : Second MQT-specific
argument */
    }
}
```

In TSK based systems, application writer can set the task priority for the statically created ZCPYDATA task.

This can be done by updating

- Argument 1 for Task priority

```
STATIC LINKCFG_DataDrv  LINKCFG_dataTable_00 [] =
{
    {
        "ZCPYDATA",             /* NAME           : Name of the data driver */
        …
        …
        …
        14 ,                    /* ARGUMENT1      : First data driver specific
argument */
        0x0                     /* ARGUMENT2      : Second data driver
specific argument */
    }
} ;
```

3.3 Dsplinkcfg script

3.3.1 Support for legacy content in DSPLink 1.6x stream

DSPLink 1.6x releases added support for multiple DSPs connected to GPP in star-topology. In context of this change, certain API signature's changed between 1.5x streams and 1.6x streams.

The --legacy option is provided if application writer wants to use the DSPLink API's exactly the same as what they were in DSPLink 1.5x. If application writer has legacy content which they do not want to change, configure DSPLink using the --legacy option in the dsplinkcfg script.

If application developer does not have existing content, do not use this option; Start using API's as present in the latest 1.6x release.

3.3.2 Support for TSK mode and SWI mode

A command line option is provided in dsplinkcfg.pl static configuration script to enable TSK mode instead SWI mode. To enable TSK mode, pass –DspTskMode = 1 option to dsplinkcfg.pl script during DSPLink build configuration. This is an optional argument. If not provided, the current default of DSP SWI mode is assumed.

How does application writer choose between TSK mode or SWI mode?
Whenever any of our modules use MPCS for multi-processor protection of shared structures, SWI_disable is called on the DSP-side to protect locally i.e. no other tasks or SWI's are allowed to execute, and scheduler was disabled. While this would give better performance for DSPLink APIs (since you are making sure it runs to completion), it holds up any other tasks or SWI's that may want to execute even non DSPLink work, till the MPCS lock is released. Also, the time for which the DSP keeps spinning to get the lock with scheduler disabled can become high if the ARM thread has taken the lock and got preempted. MSGQ and CHNL modules drivers on the DSP-side also used SWI's for doing actual processing on receiving the IPC interrupt. Similarly, other applications could use DSPLink APIs from SWI context.

In DspTskMode, the behavior of MPCS is changed to have MPCS block on a semaphore instead of disabling scheduler. This ensures that only tasks that are actually using DSPLink would block waiting for the semaphore lock to be released. Other non DSPLink tasks and SWI's would continue executing. To enable this, MSGQ and CHNL drivers have to create server tasks to receive and handle the IPC interrupts. Another impact of this is that in DspTskMode, DSPLink APIs cannot be called from SWI context, and can be called only from TSK context, since they internally call MPCS which would block on a semaphore. This mode would give better latency and enable other tasks/SWI's to run even though DSPLink is blocked, but potentially the DSPLink API could take longer to run to completion, and hence give worse throughput for DSPLink.

Depending on the application need, choose the relevant mode.

3.4 Support for symbol stripped DSP executables

DSP/BIOS Link supports application writers wishing to use a DSP executable from which the symbol table has been stripped out. This is done to reduce the size of the DSP executable.

The size of the DSP side executable can be reduced by the following ways:

3.4.1 Remove debug information

The debug information can be reduced using post-processing utility provided within the DSP CGTOOLS. For example, for C6x based devices, strip6x utility is used. When strip6x utility is used with default options, no change in applications using DSP/BIOS Link needs to be made. However source level debugging is not possible.

3.4.2 Remove the full symbol table

The size of the DSP side executable can be reduced by using the –s option in the linker. Entire symbol table can be removed by using the –s option. If strip6x utility is used with -p option, it has the similar effect, and the DSP executable size is reduced further. Applications using this feature must set the value of the `.data:DSPLINK_shmBaseAddress` section in the application specific linker command file to the start of shared memory.

- If a non-symbol stripped DSP executable is used, or an executable from which only debug information is removed, but symbol table is still present, the below steps are not required. In this case, DSPLink internally determines the address of the _DSPLINK_shmBaseAddress symbol using the symbol table on the GPP-side and uses it to fill the DSP-side section with the right address.

3.4.2.1 *Example*

For example, the application linker command file must contain a directive similar to the following:

```
SECTIONS {
.data:DSPLINK_shmBaseAddress: fill=0x8FE05000 {} > DDR
}
```

The fill value should be the start address of the shared memory used for the DRV component and varies with the devices and memory configuration used.

Please refer to the configuration file `CFG_<platform.c>` for the start of the DRV component.

3.4.2.2 *Determining the location of the DRV component*

The DRV component is present at the very start of shared memory assigned to it through the DSPLink dynamic configuration file `CFG_<platform.c>`. For most devices, this is the start of the `DSPLINKMEM/DSPLINKMEM1` memory region, depending on which memory region is used for placement of the `LINKCFG_LinkDrv` sub-module.

3.5 Support for multiple DSP boot modes

3.5.1 Overview

Multiple applications/processes on the GPP may wish to use the services provided by DSPLink to control and communicate with the DSP. DSPLink supports multiple boot modes to enable different use cases.

DSPLink PROC module supports three different scenarios for DSP boot-loading:

- Normal Boot Mode: DSPLink loads and starts the DSP running
    - DSP_BootMode_Boot_NoPwr
    - DSP_BootMode_Boot_PwrDefault
- External Load Mode: DSPLink only starts the DSP running
    - DSP_BootMode_NoLoad_NoPwr
    - DSP_BootMode_NoLoad_Pwr
- External Load and Start Mode: DSPLink does not load or start the DSP running
    - DSP_BootMode_NoBoot

In all modes, the application calls all DSPLink APIs for PROC module. DSPLink internally checks the boot mode and accordingly determines the correct action to be taken for each API. For example, APIs `PROC_load`, `PROC_start`, `PROC_stop` need to be called even in External Load or External Load and Start mode.

3.5.2 Normal Boot Mode

In this boot mode:

- GPP boots first
- Uses DSPLink to load the DSP
- Uses DSPLink to start the DSP running

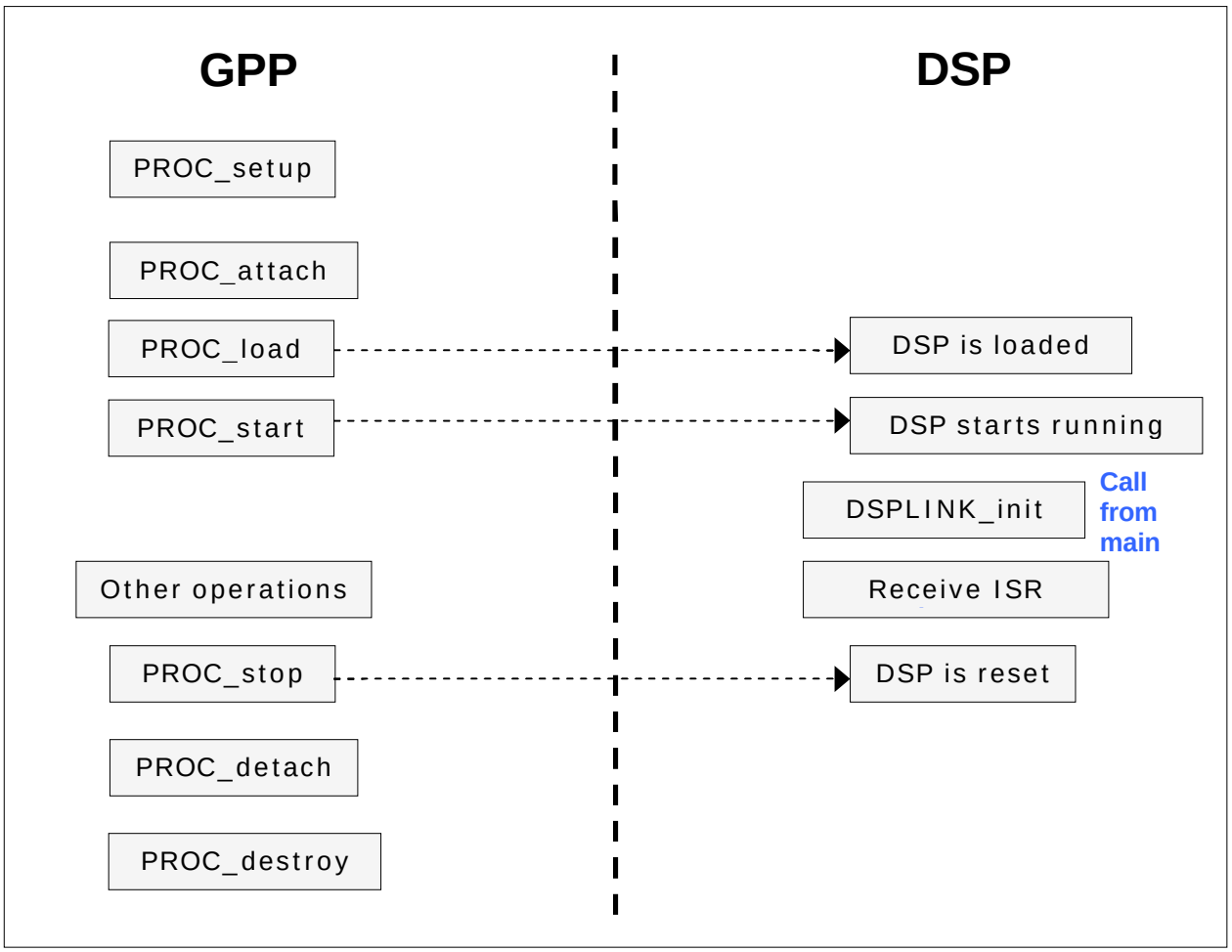

Figure 4. Normal Boot Mode

3.5.2.1 *DSP_BootMode_Boot_NoPwr*

In this boot mode, DSPLink does not do power management of DSP.

- PROC_attach places the DSP in local reset. It does not power up the DSP.
- PROC_load loads the DSP executable into DSP memory.
- PROC_start sets entry point for DSP i.e. c_int00 and release DSP from reset.
- PROC_stop places DSP in local reset.
- PROC_detach does not power down the DSP.

Application changes to support this boot mode

The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.

The configuration can be changed to use the DSP_BootMode_Boot_NoPwr boot mode in one of two possible ways:

3.5.2.1.1 Statically changing application-specific configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    DSP_BootMode_Boot_NoPwr, /* DODSPCTRL : Type of boot mode */
```

```
    ...
}
```

3.5.2.1.2 Changing default configuration file at run-time

```
/* Extern declaration to default configuration object in (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...

    /* Change dynamic configuration for boot mode */
    LINKCFG_config.dspConfigs [processorId]->dspObject->doDspCtrl =
DSP_BootMode_Boot_NoPwr ;
```

3.5.2.2 *DSP_BootMode_Boot_Pwr*

In this boot mode, DSPLink does power management of DSP.

- PROC_attach places the DSP in local reset. It powers up the DSP.
- PROC_load loads the DSP executable into DSP memory.
- PROC_start sets entry point for DSP i.e. c_int00 and release DSP from reset.
- PROC_stop places DSP in local reset.
- PROC_detach powers down the DSP.

Application changes to support this boot mode

The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.

The configuration can be changed to use the DSP_BootMode_Boot_Pwr boot mode in one of two possible ways:

3.5.2.2.1 Statically changing application-specific configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    DSP_BootMode_Boot_Pwr, /* DODSPCTRL : Type of boot mode */
    ...
}
```

3.5.2.2.2 Changing default configuration file at run-time

```
/* Extern declaration to default configuration object in (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

    /* Change dynamic configuration for boot mode */
    LINKCFG_config.dspConfigs [processorId]->dspObject->doDspCtrl =
DSP_BootMode_Boot_Pwr ;
```

3.5.2.2.3 Application changes to support this boot mode

The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.

Example of application configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    DSP_BootMode_Boot_Pwr,   /* DODSPCTRL : Type of boot mode */
```

```
    ...
}
```

3.5.3 External Load Mode

In this boot mode:

- GPP boots first
- Application/GPP boot-loader pre-loads the DSP
- Uses DSPLink to optionally power up the DSP
- Uses DSPLink to start the DSP running

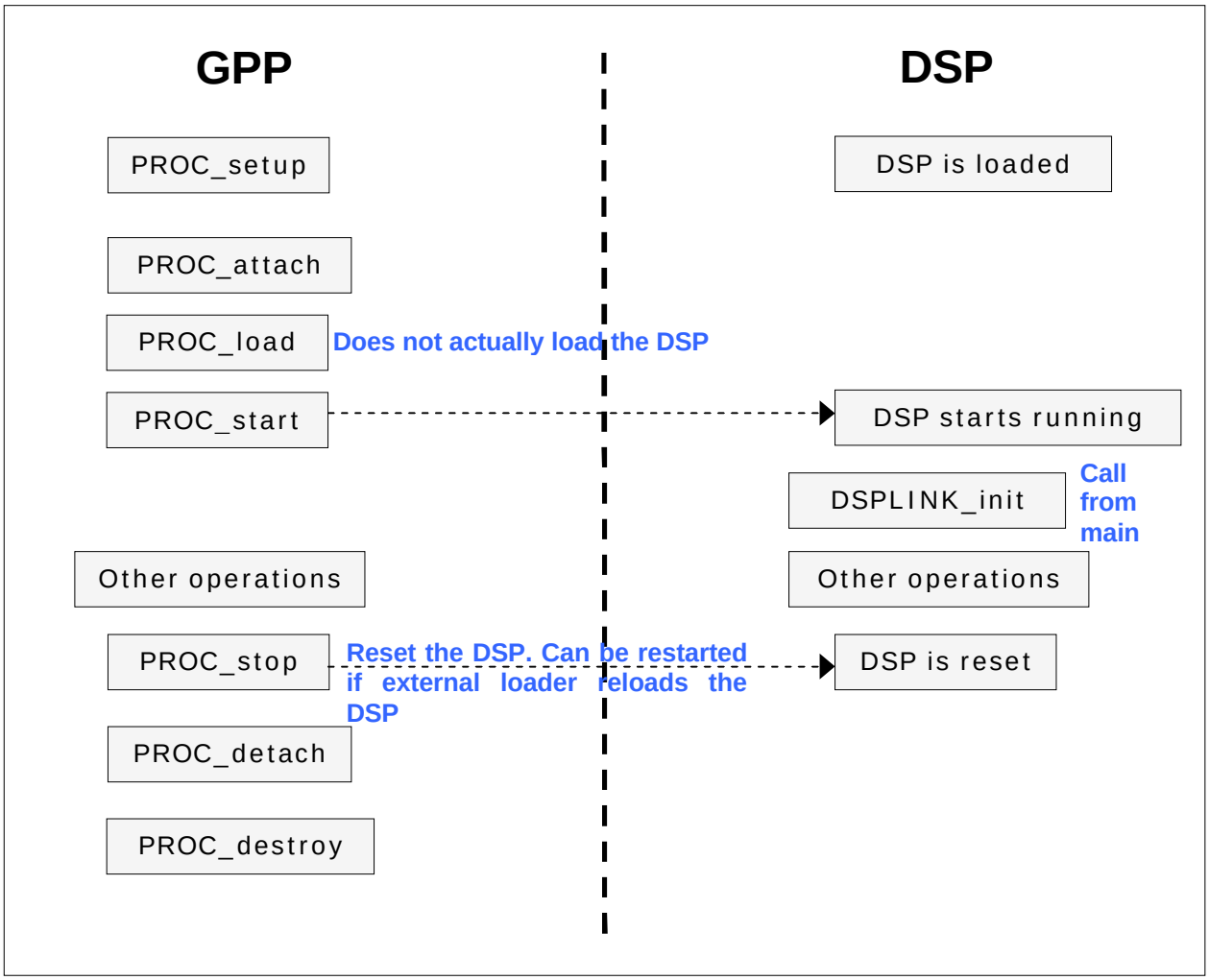

Figure 5. External Load Mode

3.5.3.1 *DSP_BootMode_NoLoad_NoPwr*

- PROC_attach places the DSP in local reset. It does not power up the DSP.
- PROC_load does not load the DSP executable into DSP memory.
- PROC_start sets entry point for DSP i.e. c_int00 and release DSP from reset.
- PROC_stop places DSP in local reset.
- PROC_detach does not power down the DSP.

Application changes to support this boot mode

1. The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.
2. Parameters passed to PROC_load API change to support the NOLOADER.

3.5.3.1.1 Step 1: Update DSPLink configuration

The configuration can be changed to use the DSP_BootMode_NoLoad_NoPwr boot mode in one of two possible ways:

3.5.3.1.1.1 Statically changing application-specific configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "NOLOADER",      /* LOADERNAME     : Name of the DSP executable loader */
    ...
    DSP_BootMode_NoLoad_NoPwr, /* DODSPCTRL : Type of boot mode */
    ...
}
```

3.5.3.1.1.2 Changing default configuration file at run-time

```
/* Extern declaration to default configuration object in (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

    /* Change dynamic configuration for boot mode */
    LINKCFG_config.dspConfigs [processorId]->dspObject->doDspCtrl =
DSP_BootMode_NoLoad_NoPwr ;
    strcpy (LINKCFG_config.dspConfigs [processorId]->dspObject->loaderName, "
NOLOADER") ;
```

3.5.3.1.2 Step 2: Call PROC_load with different parameters for NOLOADER

The NOLOADER requires additional information to enable DSPLink to successfully start the DSP. These are present as part of the NOLOADER_ImageInfo structure. This is passed to PROC_load instead of the DSP executable path.

In External Load boot modes, application can still use DSPLink to pass arguments to the DSP main function (if required).

```
#include <loaderdefs.h>
    ...

    NOLOADER_ImageInfo  image ;
    image.dspRunAddr  = 0x8FF2C780 ;  /* Address of the symbol c_int00 */
    image.shmBaseAddr = 0x8FF2EF00;   /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = 0x8ff30278 ;  /* Address of the .args section */
    image.argsSize    = 0x10  ;       /* Size of the .args section */

    ...

    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, argc, argv) ;
```

- The addresses mentioned above will be different based on the device and memory configuration used. They can be obtained using the ofd tool for the respective device. For example, for C6x based devices, the command to obtain the address of DSPLINK_shmBaseAddress is:

ofd6x.exe -v <dsp executable> | grep -rn A 2 DSPLINK_shmBaseAddress

- The value of argc i.e. number of arguments and argv i.e. arguments buffer is application dependent

The application may not need to pass arguments in .args buffer. In such cases, argc and argv can be passed as 0 and NULL respectively:

```
#include <loaderdefs.h>
    ...

    NOLOADER_ImageInfo  image ;
    image.dspRunAddr  = 0x8FF2C780;   /* Address of the symbol c_int00 */
    image.shmBaseAddr = 0x8FF2EF00 ;  /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = NULL ;
    image.argsSize    = 0x0  ;
    ...

    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, 0 , NULL) ;
```

- The addresses mentioned above will be different based on the device and memory configuration used. They can be obtained using the ofd tool for the respective device. For example, for C6x based devices, the command to obtain the address of DSPLINK_shmBaseAddress is:

  ofd6x.exe -v <dsp executable> | grep -rn A 2 DSPLINK_shmBaseAddress

3.5.3.2 *DSP_BootMode_NoLoad_Pwr*

- PROC_attach places the DSP in local reset. It powers up the DSP.
- PROC_load does not load the DSP executable into DSP memory.
- PROC_start sets entry point for DSP i.e. c_int00 and release DSP from reset.
- PROC_stop places DSP in local reset.
- PROC_detach powers down the DSP.

Application changes to support this boot mode

1. The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.
2. Parameters passed to PROC_load API change to support the NOLOADER.

3.5.3.2.1 Step 1: Update DSPLink configuration

The configuration can be changed to use the DSP_BootMode_NoLoad_Pwr boot mode in one of two possible ways:

3.5.3.2.1.1 Statically changing application-specific configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "NOLOADER",      /* LOADERNAME     : Name of the DSP executable loader */
    ...
    DSP_BootMode_NoLoad_Pwr, /* DODSPCTRL : Type of boot mode */
    ...
}
```

3.5.3.2.1.2 Changing default configuration file at run-time

```
/* Extern declaration to default configuration object in (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

    /* Change dynamic configuration for boot mode */
    LINKCFG_config.dspConfigs [processorId]->dspObject->doDspCtrl =
DSP_BootMode_NoLoad_Pwr ;
    strcpy (LINKCFG_config.dspConfigs [processorId]->dspObject->loaderName, "
NOLOADER") ;
```

3.5.3.2.2 Step 2: Call PROC_load with different parameters for NOLOADER

The NOLOADER requires additional information to enable DSPLink to successfully start the DSP. These are present as part of the `NOLOADER_ImageInfo` structure. This is passed to `PROC_load` instead of the DSP executable path.

In External Load boot modes, application can still use DSPLink to pass arguments to the DSP main function (if required).

```
#include <loaderdefs.h>
    ...

    NOLOADER_ImageInfo  image ;
    image.dspRunAddr  = 0x8FF2C780 ;  /* Address of the symbol c_int00 */
    image.shmBaseAddr = 0x8FF2EF00;   /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = 0x8ff30278 ;  /* Address of the .args section */
    image.argsSize    = 0x10  ;       /* Size of the .args section */

    ...

    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, argc, argv) ;
```

- The addresses mentioned above will be different based on the device and memory configuration used. They can be obtained using the ofd tool for the respective device. For example, for C6x based devices, the command to obtain the address of DSPLINK_shmBaseAddress is:

  ofd6x.exe -v <dsp executable> | grep -rn A 2 DSPLINK_shmBaseAddress

- The value of argc i.e. number of arguments and argv i.e. arguments buffer is application dependent

The application may not need to pass arguments in `.args` buffer. In such cases, `argc` and `argv` can be passed as 0 and `NULL` respectively:

```
#include <loaderdefs.h>
    ...

    NOLOADER_ImageInfo  image ;
    image.dspRunAddr  = 0x8FF2C780;   /* Address of the symbol c_int00 */
    image.shmBaseAddr = 0x8FF2EF00 ;  /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = NULL ;
```

```
image.argsSize    = 0x0  ;
...

status = PROC_load (ID_PROCESSOR, (Char8 *) &image, 0 , NULL) ;
```

- The addresses mentioned above will be different based on the device and memory configuration used. They can be obtained using the ofd tool for the respective device. For example, for C6x based devices, the command to obtain the address of DSPLINK_shmBaseAddress is:

  ofd6x.exe -v <dsp executable> | grep -rn A 2 DSPLINK_shmBaseAddress

3.5.4 External Load and Start Mode

There are two scenarios to be supported for this boot mode:

- GPP-based load
  1. GPP boots first
  2. Application/GPP boot-loader pre-loads the DSP
  3. Application/GPP boot-loader starts the DSP running
  4. Uses DSPLink only for IPC with the DSP

  OR
- DSP-based load
  5. DSP boots first, starts running an application
  6. ARM comes up later and sets up DSPLink, which initializes shared memory
  7. DSPLink is not used to load or start the DSP
  8. Uses DSPLink only for IPC with the DSP

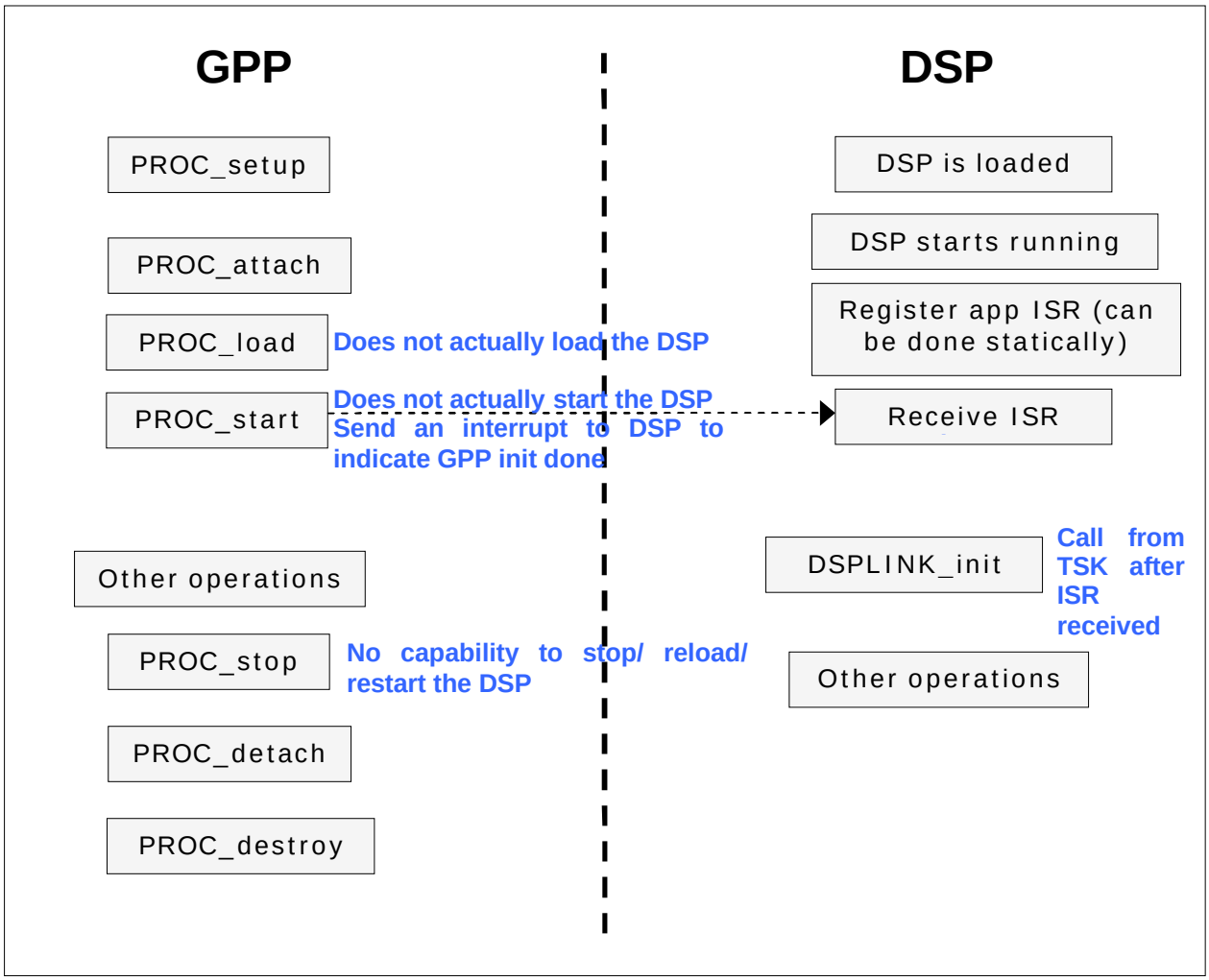

Figure 6. External Load And Start Mode

Only NoPwr based mode is supported when NoBoot mode is selected.

3.5.4.1 *DSP_BootMode_NoBoot*

- PROC_attach does not place the DSP in local reset. It does not power up the DSP.

- PROC_load does not load the DSP executable into DSP memory.
- PROC_start does not set entry point for DSP i.e. c_int00 and does not release DSP from reset.
- PROC_stop does not place DSP in local reset.
- PROC_detach does not power down the DSP.

Application changes to support this boot mode

1. The default DSPLink configuration, or application configuration passed to DSPLink in PROC_setup needs to be updated.
2. Parameters passed to PROC_load API change. PROC_load must be called. The parameters will not be used.
3. Changes on DSP-side to call DSPLINK_init after main i.e. in a TSK
   - Creation of DSPLink IOM driver must be changed from static to dynamic if CHNL component is configured in DSPLink
   - All DSPLink SMA Pools must be initialized after DSPLINK_init call in the TSK.
   - The MSGQ transport between GPP and DSP must be opened after DSPLINK_init call in the TSK.
   - DSPLink_init internally polls for the value of DSPLINK_shmBaseAddress to be a non-NULL value. One of the two below methods can be used for this:
     1. If the application uses the default polling mode, this variable must be set to NULL using the linker command file.
     2. DSPLink also supports a non-polling interrupt based mode for synchronization between the GPP and DSP

3.5.4.1.1 Step 1: Update DSPLink configuration

The configuration can be changed to use the DSP_BootMode_NoBoot boot mode in one of two possible ways:

3.5.4.1.1.1 Statically changing application-specific configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "NOLOADER",      /* LOADERNAME     : Name of the DSP executable loader */
    ...
    DSP_BootMode_NoBoot, /* DODSPCTRL : Type of boot mode */
    ...
}
```

3.5.4.1.1.2 Changing default configuration file at run-time

```
/* Extern declaration to default configuration object in (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

    /* Change dynamic configuration for boot mode */
    LINKCFG_config.dspConfigs [processorId]->dspObject->doDspCtrl =
DSP_BootMode_NoBoot ;
    strcpy (LINKCFG_config.dspConfigs [processorId]->dspObject->loaderName, "
NOLOADER") ;
```

3.5.4.1.2 Step 2: Call PROC_load with different parameters for NOLOADER

The NOLOADER may require additional information to enable DSPLink to successfully start the DSP. These are present as part of the NOLOADER_ImageInfo structure. This is passed to PROC_load instead of the DSP executable path.

In External Load & Start boot mode, application cannot use DSPLink to pass arguments to the DSP main function, because the DSP may already be running and main function completed, by the time DSPLink comes up.

```
#include <loaderdefs.h>
    ...

    NOLOADER_ImageInfo  image ;
    image.dspRunAddr  = NULL ;          /* Address of the symbol c_int00 */
    image.shmBaseAddr = 0x8FF2EF00;     /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = NULL ;          /* Address of the .args section */
    image.argsSize    = 0 ;             /* Size of the .args section */

    ...

    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, 0, NULL) ;
```

- The address mentioned above will be different based on the device and memory configuration used. It can be obtained using the ofd tool for the respective device. For example, for C6x based devices, the command to obtain the address of DSPLINK_shmBaseAddress is:

  ofd6x.exe -v <dsp executable> | grep -rn A 2 DSPLINK_shmBaseAddress

- dspRunAddr does not need to be provided, since DSPLink is not responsible for starting the DSP in this boot mode.

In case the user does not wish to specify the shmBaseAddr from ARM-side, the DSP-side can be built with the information about the shmBaseAddr. In this case, it is not required to specify the value to the NOLOADER, and the ARM-side application can become fully independent of the DSP-side build. In this case, the approach used for symbol stripped executables needs to be used, as described in section 3.2.2 in this document.

```
#include <loaderdefs.h>
    ...

    image.dspRunAddr  = NULL ;          /* Address of the symbol c_int00 */
    image.shmBaseAddr = NULL ;          /* Address of the symbol
                                         DSPLINK_shmBaseAddress from DSP COFF
                                         Executable */
    image.argsAddr    = NULL ;          /* Address of the .args section */
    image.argsSize    = 0 ;             /* Size of the .args section */
    ...

    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, 0 , NULL) ;
```

3.5.4.1.3 Step 3: Make changes on DSP-side to call DSPLINK_init from task instead of main

Creation of DSP-side IOM driver dynamically

If CHNL module is not enabled in the build, this step is not applicable.

By default, the application will usually create the DSP-side DSPLink driver statically, by including the following two TCI files within the application's TCF configuration file:

- DSPLink IOM driver: `dsplink-iom.tci`
- DSPLink DIO adapter for usage with SIO: `dsplink-dio.tci`

The IOM and DIO drivers need to be created dynamically for this boot-mode.

For dynamic creation of IOM driver:

- Comment out the static creation of the `dsplink` IOM driver in `dsplink-iom.tci` in the application TCF
- The creation of `ZCPYLINK_SWI_OBJ` can still be done statically.

```
/*  ============================================================================
 *  UDEV : DSP/BIOS LINK
 *  ============================================================================
 */
/*var dsplink = prog.module("UDEV").create("dsplink");
dsplink.initFxn      = prog.decl("ZCPYDATA_init");
dsplink.fxnTable     = prog.decl("ZCPYDATA_FXNS");
dsplink.fxnTableType = "IOM_Fxns";
dsplink.comment      = "DSP/BIOS LINK  - IOM Driver";*/

/*  ============================================================================
 *  SWI : ZCPYLINK_SWI_OBJ
 *  ============================================================================
 */
var ZCPYLINK_SWI_OBJ      = prog.module("SWI").create("ZCPYDATA_SWI_OBJ");
ZCPYLINK_SWI_OBJ.comment  = "This swi handles the data transfer in DSPLINK";
ZCPYLINK_SWI_OBJ.fxn      =  prog.decl("ZCPYDATA_SWI");
ZCPYLINK_SWI_OBJ.priority = 14;
ZCPYLINK_SWI_OBJ.arg0     =  $externPtr("ZCPYDATA_devObj");
```

For dynamic creation of DIO adapter:

- Do not include `dsplink-dio.tci` in the application TCF file

The code given below can be used as reference to create the IOM and DIO drivers dynamically.

```
extern IOM_Fxns ZCPYDATA_FXNS ;
extern Void ZCPYDATA_init (Void) ;

DIO_Params dioAttrs = {
     "/dsplink",
      NULL
} ;

DEV_Attrs devAttrs = {
        0,            /* devId */
        0,            /* dsplink deviceParams */
        DEV_IOMTYPE,  /* dsplink driver type */
        0             /* dsplink devp */
} ;

DEV_Attrs dioDevAttrs = {
        0,            /* devId */
        &dioAttrs,    /* DIO deviceParams */
        DEV_SIOTYPE,  /* DIO type */
```

```
        0             /* devp */
} ;

    ...

    /* Create IOM driver dynamically */
    status = DEV_createDevice("/dsplink", &ZCPYDATA_FXNS, (Fxn)
&ZCPYDATA_init, &devAttrs) ;

    /* Create DIO adapter dynamically */
    status = DEV_createDevice("/dio_dsplink", &DIO_tskDynamicFxns, NULL,
&dioDevAttrs);
```

Calling **POOL_open** in a task

Dummy configuration needs to be defined for the POOL so that DSP/BIOS will not internally call DSPLink SMAPOOL initialization functions.

```
/* Dummy base configuration for POOLs */
POOL_Obj MESSAGE_Pools [NUM_POOLS] =
{
    POOL_NOENTRY,
    POOL_NOENTRY
} ;

/* POOL_config variable as needed by DSP/BIOS */
POOL_Config POOL_config = {MESSAGE_Pools, NUM_POOLS} ;
```

In the application task after DSPLINK_init is called, the actual POOL configuration must be updated into the POOL configuration structure, and a call made to POOL_open.

```
/* Define actual global SMAPOOL parameters */
SMAPOOL_Params            MESSAGE_PoolParams [NUM_POOLS] ;

    /* Declare temporary local pool object for opening the pool */
    POOL_Obj                  poolObj ;

    ...

    /* Setup SMAPOOL parameters */
    MESSAGE_PoolParams [0].poolId        = 0 ;
    MESSAGE_PoolParams [0].exactMatchReq = TRUE ;

    /* Populate the global POOL configuration structure with actual POOL
configuration.*/
    poolObj.initFxn = SMAPOOL_init ;
    poolObj.fxns    = (POOL_Fxns *) &SMAPOOL_FXNS ;
    poolObj.params  = &(MESSAGE_PoolParams [0]) ;
    poolObj.object  = NULL ;

    /* Open the POOL dynamically */
    status = POOL_open (0, &poolObj) ;
```

Calling **MSGQ_transportOpen** in a task

Dummy configuration needs to be defined for the Message Queue transport so that DSP/BIOS will not internally call DSPLink MQT initialization functions.

```
/* Dummy base configuration for MQT */
```

```
MSGQ_TransportObj MESSAGE_Transports [MAX_PROCESSORS] =
{
     MSGQ_NOTRANSPORT,    /* Represents the local processor */
     MSGQ_NOTRANSPORT     /* Dummy transport for DSPLink */
}

/* MSGQ_config variable as needed by DSP/BIOS */
MSGQ_Config MSGQ_config =
{
    MESSAGE_MsgQueues,
    MESSAGE_Transports,
    NUM_MSG_QUEUES,
    MAX_PROCESSORS,
    0,
    MSGQ_INVALIDMSGQ,
    POOL_INVALIDID
} ;
```

In the application task after DSPLINK_init is called, the actual MQT configuration must be updated into the MSGQ configuration structure, and a call made to MSGQ_transportOpen.

```
/* Define actual global ZCPYMQT parameters */
ZCPYMQT_Params MESSAGE_MqtParams ;

    /* Declare temporary local transport object for opening the transport */
    MSGQ_TransportObj transport ;

    /* Initialize the transport object for ZCPYMQT */
    transport.initFxn = ZCPYMQT_init ;         /* Init Function */
    transport.fxns    = (MSGQ_TransportFxns *) &ZCPYMQT_FXNS ; /* Transport
interface functions */
    transport.params  = &MESSAGE_MqtParams ;   /* Transport params */
    transport.object  = NULL ;                 /* Filled in by transport */
    transport.procId  = ID_GPP  ;              /* Processor Id */

    /* Open the Message Queue Transport dynamically. */
    status = MSGQ_transportOpen (ID_GPP, &transport) ;
```

Calling **DSPLINK_init from TSK**

DSPLink_init internally polls for the value of DSPLINK_shmBaseAddress to be a non-NULL value. One of the two below methods can be used for this:

1. Polling Mode: This variable must be set to NULL using the linker command file.
2. Non-polling Mode: Use interrupt based synchronization between the GPP and DSP

1. Polling mode: Application linker command file update

A command needs to be added in the application linker command file to initialize the DSPLINK_shmBaseAddress value to NULL.

```
/* Set the contents of DSPLINK_shmBaseAddress to NULL. */
SECTIONS {
    .data:DSPLINK_shmBaseAddress: fill=0x00000000 {} > DDR
}
```

2. Non-Polling mode: Interrupt-based synchronization

1. The application can register for the DSPLink interrupt for IPS ID 0 using DSP/BIOS HWI APIs, or static configuration. In this case, as soon as DSPLink

GPP-side has completed configuration of shared memory, it sends an interrupt to the DSP.

2. The application-registered ISR shall get called, which can post a semaphore to wake-up a task that was blocked on the semaphore.
3. The task then calls `DSPLink_init` and opens the POOLs and MQT dynamically.
4. Once this is done, DSP-side application can start using DSPLink for IPC between the GPP and DSP.

3.6 Support for multiple types of COFF based loaders

3.6.1 Overview

DSPLink supports loading of COFF executable in DSP memory using PROC_load API. The COFF file can be loaded in DSP memory from a file or from memory. DSPLink supports multiple types of COFF loaders for different system needs. This section details the types of loaders and how they are used.

3.6.2 PROC_load using a COFF file

This is the default usage of PROC_load. The DSP executable is generated using the required tools and is present in the target file system.

Example of application configuration file

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "COFF",              /* LOADERNAME : Name of DSP executable loader */
    ...
}
```

Example of PROC_load API call

```
    #define DSP_EXECUTABLE "/opt/message.out"


    status = PROC_load (ID_PROCESSOR, DSP_EXECUTABLE, argc, argv) ;
```

- The value of argc i.e. number of arguments and argv i.e. arguments will be application dependant. If not required, they can be passed as 0 and NULL respectively.

3.6.3 PROC_load using optimized COFF loader on shared memory based platforms like DM6446, DRA44x etc

This is the optimized usage of the default COFF loader. In this loader the section data is directly copied to the DSP memory. This is possible because DSP memory is directly accessible for shared memory based devices. The DSP executable is generated using the required tools and is present in the target file system.

The configuration can be changed to use the COFFSHM loader in one of two possible ways:

3.6.3.1 *Statically changing application-specific configuration file*

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "COFFSHM",            /* LOADERNAME : Name of DSP executable loader */
    ...
}
```

3.6.3.2 *Changing default configuration file at run-time*

```
/* Extern declaration to default configuration object in  (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...
```

```
    /* Change dynamic configuration to use COFFSHM loader */
    strcpy (LINKCFG_config.dspConfigs [processorId]->dspObject->loaderName,
"COFFSHM") ;
```

Example of PROC_load API call

```
    #define DSP_EXECUTABLE "/opt/message.out"


    status = PROC_load (ID_PROCESSOR, DSP_EXECUTABLE, argc, argv) ;
```

- The value of argc i.e. number of arguments and argv i.e. arguments will be application dependant. If not required, they can be passed as 0 and NULL respectively.

3.6.4 PROC_load using a COFF file present in ARM memory

In certain application scenarios, the COFF file can be loaded into a memory buffer outside DSPLink. The DSP executable is loaded in DSP memory by reading the buffer contents. This type of loader is useful, if, for example, there is no file system on the ARM side.

The configuration can be changed to use the COFFMEM loader in one of two possible ways:

3.6.4.1 *Statically changing application-specific configuration file*

```
STATIC LINKCFG_Dsp  LINKCFG_dspObject =
{
    ...
    "COFFMEM",            /* LOADERNAME : Name of DSP executable loader */
    ...
}
```

3.6.4.2 *Changing default configuration file at run-time*

```
/* Extern declaration to default configuration object in  (CFG_<PLATFORM>.c)*/
extern LINKCFG_Object  LINKCFG_config ;
        ...
        ...

    /* Change dynamic configuration to use COFFMEM loader */
    strcpy (LINKCFG_config.dspConfigs [processorId]->dspObject->loaderName,
"COFFMEM") ;
```

DSPLink COFFMEM loader expects the file to be pre-loaded into a memory buffer. This may be done by applications in multiple possible ways.

The user needs to ensure that the buffer used for loading the file is physically contiguous in non-cacheable memory, and its physical address is known.

Example of PROC_load API call using POOL buffer for loading the file

```
    #include <loaderdefs.h>

    ...
    COFFLOADER_ImageInfo image ;

    ...
    /* Size of files in bytes */
    image.size = size_of_file_in_bytes ;
```

```
    /* Physical address of memory where COFF file is present. */
    image.fileAddr = bufAddr ;

    /* Load the buffer into DSP memory. */
    status = PROC_load (ID_PROCESSOR, (Char8 *) &image, argc, argv) ;
```

- The value of argc i.e. number of arguments and argv i.e. arguments will be application dependant. If not required, they can be passed as 0 and NULL respectively.

An example usage where a POOL is used to allocate the buffer used for loading the file is shown below. Using a POOL buffer ensures that the buffer is physically contiguous, and also enables the user to translate the buffer address to a physical address.

Example of PROC_load API call using POOL buffer for loading the file

```
    #include <loaderdefs.h>

    ...
    SAMPLE_POOL_ID = POOL_makePoolId (ID_PROCESSOR, 0 ) ; /* 0 is the
                                        Zeroth pool id of the DSP processor
                                        Identified by ID_PROCESSOR .*/

    COFFLOADER_ImageInfo image ;

    ...

    /* Configure pool for the file size. */
    SamplePoolAttrs.numBufPools = 1 ;
    SampleNumBuffers [0] = 1 ;
    SampleBufSizes [0]   = DSPLINK_ALIGN (size_of_file_in_bytes,
                                        DSPLINK_BUF_ALIGN) ;

    /* If approximate file size is used, change line below to use
     * exactMatchReq as FALSE
     */
    SamplePoolAttrs.exactMatchReq = TRUE ;

    /* Open the pool. */
    status = POOL_open (SAMPLE_POOL_ID, &SamplePoolAttrs) ;
    if (DSP_SUCCEEDED (status)) {
        status = POOL_alloc (SAMPLE_POOL_ID,
                             (Pvoid *) &srcAddr
                             (SampleBufSizes [0]) ;
    }

    /* ------------------------------------------------------------------
     * <Code to read the full contents of file to be loaded, into the
     * buffer.>
     * ------------------------------------------------------------------
     */

    /* Get physical address of the user buffer */
    if (DSP_SUCCEEDED (status)) {
        status = POOL_translateAddr (SAMPLE_POOL_ID ,
                                     &dstAddr,
                                     AddrType_Phy,
```

```
                                    srcAddr,
                                    AddrType_Usr) ;
    }

    if (DSP_SUCCEEDED (status)) {
        /* Size of files in bytes */
        image.size = size_of_file_in_bytes ;

        /* Physical address of memory where COFF file is present. */
        image.fileAddr = dstAddr ;

        /* Load the buffer into DSP memory. */
        status = PROC_load (ID_PROCESSOR, (Char8 *) &image, argc, argv) ;
    }

    ...

    /* Start the DSP running */
    if (DSP_SUCCEEDED (status)) {
        status = PROC_start (ID_PROCESSOR) ;
    }

    ...

    /* Now POOL can be closed to free up the shared memory, so that POOL
     * can be reopened (if needed) with different parameters.
     */
    if (DSP_SUCCEEDED (status)) {
        tmpStatus = POOL_close (SAMPLE_POOL_ID) ;
    }
```

- The value of argc i.e. number of arguments and argv i.e. arguments will be application dependant. If not required, they can be passed as 0 and NULL respectively.
- The POOL can be opened initially with all the shared memory requirements or first the POOL can be opened with only the file size requirement, closed after PROC_start and then re-opened with the shared memory requirements.

3.7 Concepts

3.7.1 Cleanup of the kernel driver

On operating system such as Linux, multiple processes and threads may use DSP/BIOS™ LINK to communicate with the DSP. It may happen that one or more of the processes may crash due to a user-space application issue or invalid state. In such cases, it is desirable to restore the kernel state for the DSPLINK driver such that applications may be able to restart and use the DSPLINK driver for further communication with the DSP.

In addition, if an application process is unable to perform the required shutdown calls corresponding to the startup calls made by it, the kernel state must still be restored to a state such that it does not affect the execution of other (possibly independent) application processes.

This is done using two mechanisms on Linux. A similar mechanism may be implemented on other GPP operating system having this support.

3.7.1.1 *Signal handling on Linux*

To support cleanup after an application crash or Ctrl C to terminate the process, the DSPLINK driver needs to cleanup the kernel resources used by it and restore the kernel driver to a consistent state. This is done using signals.

1. By default, DSPLINK registers signal handlers for process termination signals. Within the signal handler, DSPLINK cleans up the kernel driver. It also frees as many kernel resources as is possible.
2. If the application wishes to perform its own cleanup and does not wish DSPLINK to register signal handlers for the process termination signals, it can disable the signal handling through the OS-specific dynamic configuration file (CFG_<GPPOS>.c). In this case, it is the application's prerogative to ensure that it also makes the required DSPLINK shutdown calls within its signal handler.
3. If the system design requires some specific behavior for certain process termination signals, the specific signals to be handled by DSPLINK is also dynamically configurable within the GPP OS-specific dynamic configuration file. Note that if the number of signals within the array is modified, this needs to be reflected in the `NUMSIGNALS` field within the `LINKCFG_gppOsObject` object.

3.7.1.2 *Automatic cleanup on process exit in normal process termination*

To support the scenario where an application process is unable to perform the required shutdown calls corresponding to the startup calls made by it, `atexit` handler is registered by default by DSPLINK with Linux for each process. This handler performs all shutdown APIs for that process and gets called automatically when the process terminates.

Applications are also free to register their `atexit` handlers for their own usage. The `atexit` handlers are executed on a first-registered-last-executed basis.

Registration of the `atexit` handler is not made dynamically configurable to ensure that processes are not allowed to corrupt the system state.

q It is a good practice for applications to always make all shutdown calls corresponding to the startup calls. The atexit feature should not be relied upon by applications, because this may not be available on other operating systems, and affects the portability of the applications.

4 POOL

4.1 Overview

The POOL component provides APIs for configuring shared memory regions across processors.

These buffers are used by other modules from DSP/BIOS Link for providing inter-processor communication functionality.

The specific services provided by this module are:

- Configure the shared memory region through open & close calls.
- Allocate and free buffers from the shared memory region.
- Translate address of a buffer allocated to different address spaces (e.g. GPP to DSP)

- Synchronize contents of memory as seen by the different CPU cores.

This component is responsible for providing a uniform view of different memory pool implementations, which may be specific to the hardware architecture or OS on which DSP/BIOS™ LINK is ported. This component is based on the POOL interface in DSP/BIOS™.

The DSP/BIOS LINK POOL is not a heap-based pool. It is a fixed-size buffer pool. It requires the specific configuration of number of buffers and sizes as will be used by the application. The exactMatchReq property only allows users the flexibility of configuring an approximate size for each buffer. However, the maximum number of buffers still must be configured.

4.2 Configuration

4.2.1 Configuration parameters

The POOL_open () call is used to configure the shared memory requirement for the application. Since the pool is shared between DSP and GPP, the sizes of the buffers must be cache aligned. DSP/BIOS LINK provides an API DSPLINK_ALIGN which can be used to get the cache aligned size.

For SMA Pool, we need to configure a parameter of type SMAPOOL_Attrs in the POOL_open call. The POOL_open call takes a structure of type SMAPOOL_Attrs for the POOL_open call. The elements in the structure are:

- numBufPools: Number of buffer pools.
- bufSizes: Array of sizes of the buffers in each buffer pools. The buffer sizes must be cache aligned.
- numBuffers: Array of number of buffers in each buffer pools.
- exactMatchReq: Flag indicating whether requested size is to be rounded to nearest available larger size in Pools or exact match has to be performed.

4.2.2 Exact match required

1. exactMatchReq specified as TRUE: With this configuration, error is returned if the exact size is not found configured.
2. exactMatchReq specified as FALSE: With this configuration, the highest buffer size next closest in size to the specified size to be allocated is returned. If the nearest higher size buffers are exhausted, POOL_alloc () call will return with DSP_EMEMORY or memory allocation failure.
    1. You can set exactMatchReq field in the SMAPOOL_Attrs while opening the pool to FALSE, and use a large buffer size for the configuring the pool (all allocations must be less than this size).
    2. Please note that the disadvantage of using exactMatchReq as FALSE is possible wastage of memory, since even a buffer of size 128 bytes may result in an allocation of size 1024 bytes if only buffers of 1024 bytes are configured in the pool.

4.2.3 Buffer configuration

To set the pool attributes, you need to know how many buffers that you need in the shared memory as well as their size. Depending on your application needs, you configure your pool according to the size and the number of the buffers required. You can also configure the pool to return the buffer only if an exact match size is configured or to return a buffer with a size which fits best to what has been asked.

4.2.4 Example

If you want to configure the pool (with exact match TRUE) to allocate 10 buffers of size 128, 10 buffers of size 512 and 10 buffers of size 2048 you may configure the pool as follows.

```
#define NUM_BUF_SIZES   3 /* 3 buffer sizes to be configured */
#define SAMPLE_POOL_NO  0 /* Pool no as in the config CFG_<PLATFORM>.c.

Uint32        numBufs [NUM_BUF_SIZES] ;
Uint32        size [NUM_BUF_SIZES] ;
SMAPOOL_Attrs poolAttrs ;

 . . .

if (DSP_SUCCEEDED (status)) {
    size    [0] = 128 ;
    numBufs [0] = 10 ;

    size    [1] = 512 ;
    numBufs [1] = 15 ;

    size    [2] = 2048 ;
    numBufs [2] = 5 ;

    poolAttrs.bufSizes      = (Uint32 *) &size ;
    poolAttrs.numBuffers    = (Uint32 *) &numBufs ;
    poolAttrs.numBufPools   = NUM_BUF_SIZES ;
    poolAttrs.exactMatchReq = TRUE ;

    /* Make the pool id  from pool no and dsp processor id . Applicable
     * GPP side only
     */
    poolId = POOL_makePoolId (ID_PROCESSOR ,SAMPLE_POOL_NO) ;
    status = POOL_open (poolId, &poolAttrs) ;
    if (DSP_FAILED (status)) {
        APP_Print ("POOL_open () failed. Status = [0x%x]\n", status) ;
    }
}
```

- The above is just a dummy representation of how to configure the POOL. In real world applications, this is more tuned to the application buffer size requirements.

In the above example:

1. Consider a scenario where exactMatchReq is TRUE. The application can successfully allocate 10 buffers of size 128, 15 buffers of size 512, and 5 buffers of size 2048. If you want to allocate a buffer of size 256, the above configuration will not support it and POOL_alloc will return error.
2. Consider a scenario where exactMatchReq is FALSE. The application can successfully allocate 10 buffers of size 128, 15 buffers of size 512, and 5 buffers of size 2048. as before. However, the difference is that an attempt to allocate a buffer of size 256 will result in the POOL_alloc () call returning a buffer of next larger size i.e. 512 if available. If buffer of size 512 is not available it will return DSP_EMEMORY or memory allocation failure.

4.2.5 Configuring multiple pools

4.2.5.1 *GPP side*

CFG_<platform>.c needs to be updated for multiple POOLS on GPP side. The parameters to be updated are:

1. Add other 'n' entries in LINKCFG_poolTable_00 with the same name. MEMENTRY i.e. the memory entry from which the pool will be configured and the POOLSIZE i.e. the size of the second pool can be configured as desired by the application.

```
STATIC CONST LINKCFG_Pool  LINKCFG_poolTable_00 [] =
{
    {
        "SMAPOOL",              /* NAME    : Name of the pool */
        (Uint32) 1,             /* MEMENTRY: Mem entry ID (-1 if not needed)*/
        (Uint32) 0x35000,       /* POOLSIZE: Size of pool (-1 if not needed)*/
        (Uint32) -1,            /* IPSID   : ID of the IPS used */
        (Uint32) -1,            /* IPSEVENTNO: IPS Event number for POOL */
        0x0,                    /* ARGUMENT1 : First Pool-specific argument */
        0x0                     /* ARGUMENT2 : Second Pool-specific argument*/
    },
    {
        "SMAPOOL",              /* NAME       */
        (Uint32) 1,             /* MEMENTRY   */
        (Uint32) 0x35000,       /* POOLSIZE   */
        (Uint32) -1,            /* IPSID      */
        (Uint32) -1,            /* IPSEVENTNO */
        0x0,                    /* ARGUMENT1  */
        0x0                     /* ARGUMENT2  */
    }
}
```

2. Update NUMPOOLS in LINKCFG_linkDrvObjects to 'n'.

```
STATIC CONST LINKCFG_LinkDrv  LINKCFG_linkDrvObjects [] =
{
    {
        "SHMDRV",               /* NAME: Name of the link driver */
        (Uint32) 100000000,     /* HSHKPOLLCOUNT  : Poll value for which */
                                *  handshake waits (-1 if infinite) */
        (Uint32) 1,             /* MEMENTRY: Mem entry ID (-1 if not needed)*/
        0,                      /* IPSTABLEID  : ID of the IPS table used */
        2,                      /* IPSENTRIES  : Number of IPS supported */
        0,                      /* POOLTABLEID : ID of the POOL table */
        2,                      /* NUMPOOLS    : Number of POOLs supported */
        0,                      /* DATATABLEID : ID of data driver table */
        1,                      /* NUMDATADRV  : Number of data drivers */
        0,                      /* MQTID       : ID of the MQT */
        0,                      /* RINGIOTABLEID: RingIO Table Id */
        0,                      /* MPLISTTABLEID: MpList Table Id */
        0                       /* MPCSTABLEID  : MPCS Table ID */
    }
} ;
```

- After configuring the pools, if application is using dynamic configuration, the GPP-side application must be rebuilt. If dynamic configuration is not used, the

DSP/BIOS™ LINK API library must be rebuilt, followed by the application rebuild.

4.2.5.2 *DSP side*

With BIOS 5.xx:

1. The global variable POOL_config must be configured as required by DSP/BIOS™.

```
POOL_Config POOL_config = {MESSAGE_Pools, 2} ;
```

2. The pools must be configured as required by DSP/BIOS™.

```
POOL_Obj MESSAGE_Pools [NUM_POOLS] =
{
    {
        &SMAPOOL_init,                  /* Init Function  */
        (POOL_Fxns *) &SMAPOOL_FXNS,  /* Pool interface functions */
        &MESSAGE_PoolParams [0],      /* Pool params */
        NULL                          /* Pool object: Set in pool impl. */
    },
    {
        &SMAPOOL_init,                  /* Init Function  */
        (POOL_Fxns *) &SMAPOOL_FXNS,  /* Pool interface functions */
        &MESSAGE_PoolParams [1],      /* Pool params */
        NULL                          /* Pool object: Set in pool impl. */
    }
}
```

- After configuring the pools, the DSP-side application must be rebuilt to generate the DSP executable.

4.3 POOL requirements for different DSP/BIOS™ LINK components

4.3.1 PROC

PROC component has no POOL requirements.

4.3.2 NOTIFY

NOTIFY component has no POOL requirements.

4.3.3 MPCS

Each MPCS has the following buffer requirements:

1. `MPCS_ShObj`: MPCS control structure size

Along with this, application must configure buffers as required according to application need for the protocol that uses MPCS.

4.3.4 MSGQ

One pool must be reserved for Message Queue transport. This pool is used by all applications that use MSGQ. This pool must configured messages as given below:

1. `ZCPYMQT_CTRLMSG_SIZE`: MSGQ control structure size. The number of buffers of this size required varies depending on the frequency with which `MSGQ_locate` is performed by the application. This message size is required for the MSGQ transport. The POOL ID configured for the MSGQ transport must have this buffer size configured.

The same or different pool(s) as used for the MQT can be used to satisfy the following other buffer requirements for messaging:

1. MSGQ_AsyncLocateMsg: MSGQ ASYNC locate call message size requirement, one for each ASYNC locate call. The number of buffers of this size required varies depending on the frequency with which MSGQ_locateAsync is performed by the application. This size is not required to be configured if application is not using MSGQ_locateAsync. The pool ID to be used internally for allocating these messages is provided within the MSGQ_LocateAsyncAttrs passed to the MSGQ_locateAsync call.
2. MSGQ_AsyncErrorMsg: MSGQ ASYNC error buffer size required if application wishes to handle asynchronous error messages by setting error handler MSGQ through the API MSGQ_setErrorHandler (). When an asynchronous error occurs within the MSGQ module, if the application has registered an error handler, an error message of this size is allocated, filled with async error details, and sent to the registered MSGQ. If no error handling MSGQ is registered, this size does not need to be configured. The pool ID to be used for this is passed to the MSGQ_setErrorHandler () call. Currently, the DSP/BIOS LINK MQT does not send any asynchronous error messages, since the shared physical link is lossless. However, this may be required for MQTs built over certain other possibly lossy physical connections.

Along with this, the application must configure message buffers according to application need. The considerations for allocating and reserving memory for message buffers are:

1. Each message buffer used by the application must have the fixed size MSGQ_MsgHeader as the first field in the message. The size of this structure must be included in the message size to be allocated by the application.
2. The sizes of the message buffers (including fixed header) must be a multiple of cache line size (if applicable for the platform). For example, for Davinci, the size of message buffer must be a multiple of 128 bytes.
3. The MSGQ_alloc () call takes the pool ID to be used for allocating the message buffers. The corresponding pool must be configured to support allocation of the required numbers of message buffers.
4. Message buffers of different sizes can be used within the same application as long as the generic constraints mentioned above are followed.

4.3.5 RingIO

Each RingIO has the following buffer requirements:

1. Data buffer: This size should be cache aligned. If foot buffer is configured, then the size required is (data buffer size + foot buffer size).
2. Attribute size buffer (if configured): This size should be cache aligned.
3. RingIO_ControlStruct: RingIO control structure size
4. MPCS_ShObj: MPCS control structure size

4.3.6 MPLIST

Each MPLIST has the following buffer requirements:

1. MPLIST_List: MPLIST control structure size.

Along with this, the application must configure buffers which will be the list elements according to application need. The considerations for allocating and reserving memory for buffers to be used with MPLIST are:

1. Each buffer used by the application on the MPLIST must have the fixed size `MPLIST_Header` as the first field in the message. The size of this structure must be included in the buffer size to be allocated by the application.
2. The sizes of the buffers (including fixed header) must be a multiple of cache line size (if applicable for the platform). For example, for Davinci, the size of buffer must be a multiple of 128 bytes.
3. Buffers of different sizes can be used within the same application with MPLIST as long as the generic constraints mentioned above are followed.

4.3.7 CHNL

The application needs to configure buffers for data transfer according to application need.

The considerations for allocating and reserving memory for data transfer buffers are:

1. The size of the data transfer buffers must be a multiple of cache line size (if applicable for the platform). For example, for Davinci, the size of data transfer buffer must be a multiple of 128 bytes.
2. The pool ID to be used for allocating the buffers is configured within the `CFG_<PLATFORM>.c` file within the `LINKCFG_DataDrv` object (field `POOLID`). This pool ID gets used internally when the buffers used for data transfer are allocated using the `CHNL_allocateBuffer ()` API. One or more buffers may be allocated for each channel using the `CHNL_allocateBuffer ()` API.
3. For each channel, buffers of a fixed size are used. Different channels can have different buffer sizes as long as they are less than the maximum buffer size supported by the data transfer driver.
4. The POOL configuration for the data driver must take buffer requirements for all channels into account.

4.4 POOL setup for multi process applications

There are two ways in which POOL can be configured for multi process applications

4.4.1 Opening all pools at system initialization time

The main process needs to understand the POOL requirements for the complete application and configure the POOL using `POOL_open ()` call accordingly.

In each other process which attaches to use DSPLINK, the application can call `POOL_open ()` with params as NULL. Any parameters that are provided are ignored for all calls subsequent to the first one for each pool ID. The application need not call `POOL_open ()` if it is not doing anything that uses a pool (RingIO, MSGQ, `POOL_alloc ()` etc.).

4.4.2 Opening pools dynamically

In this method, each application may open its own pool as required. All pools need not be opened at system initialization time, and the system integrator does not need to know pool requirements of all applications.

One pool must be reserved for Message Queue transport. This pool must be opened statically before `PROC_start ()` to ensure correct behavior on the DSP-side. All

applications using MSGQ must call POOL_open () for this pool ID to be allowed to use messaging.

To open pools dynamically, following procedure must be followed:

1. On the DSP-side, instead of specifying the actual pool configuration, use a dummy configuration. This ensures that BIOS boot-up in POOL module initialization does not hang waiting for the GPP-side pool to be opened.

```
/** ============================================================================
 *  @name   MESSAGE_DummyPoolFxns
 *
 *  @desc   Dummy pool functions to allow dynamic pool open as required
 *  ============================================================================
 */
POOL_Fxns MESSAGE_DummyPoolFxns =
{
    (POOL_Open)  SYS_zero,  /* return 0 so POOL_init will not fail */
    (POOL_Close) FXN_F_nop, /* have close do nothing */
    (POOL_Alloc) SYS_one,   /* have alloc return non-zero */
    (POOL_Free)  FXN_F_nop  /* have free do nothing */
} ;


/** ============================================================================
 *  @name   MESSAGE_Pools
 *
 *  @desc   Array of pools.
 *  ============================================================================
 */
POOL_Obj MESSAGE_Pools [NUM_POOLS] =
{
    {
        &FXN_F_nop,                 /* Init Function */
        &MESSAGE_DummyPoolFxns,     /* Pool interface functions */
        NULL,                       /* Pool params */
        NULL                        /* Pool obj: Set in pool impl. */
    }
} ;
```

2. After the GPP-side pool is opened (which may be after PROC_start ()), the application can notify the DSP-side (possibly through NOTIFY_notify (), since NOTIFY module does not require any pool). Note that the application must ensure that it does not attempt to use the POOL either directly or indirectly before the pool is opened on both GPP and DSP-sides.

3. When the notification is received from GPP that the pool has been opened, the DSP-side application can now provide the actual configuration parameters and open the pool. This is to ensure that the DSP-side application does not spin waiting for the pool to be opened by GPP-side. With this method, the DSP-side can simply wait on a semaphore for the GPP-side pool to be opened, and the notification callback posts this semaphore. The pool can be opened on DSP-side by:

```
SMAPOOL_Params MESSAGE_PoolParams [NUM_POOLS] ;

MESSAGE_PoolParams [poolId].poolId        = poolId ;
MESSAGE_PoolParams [poolId].exactMatchReq = TRUE ;
```

```
MESSAGE_Pools [poolId].initFxn = SMAPOOL_init ;
MESSAGE_Pools [poolId].fxns    = (POOL_Fxns *) &SMAPOOL_FXNS ;
MESSAGE_Pools [poolId].params  = &(MESSAGE_PoolParams [poolId]) ;
MESSAGE_Pools [poolId].object  = NULL ;

status = POOL_open (poolId, &(MESSAGE_PoolParams [poolId])) ;
```

- The code given above is only indicative. Similar changes would need to be done in the application to open pools dynamically.

5 RingIO

5.1 Overview

The RingIO component provides Ring Buffer based data streaming.

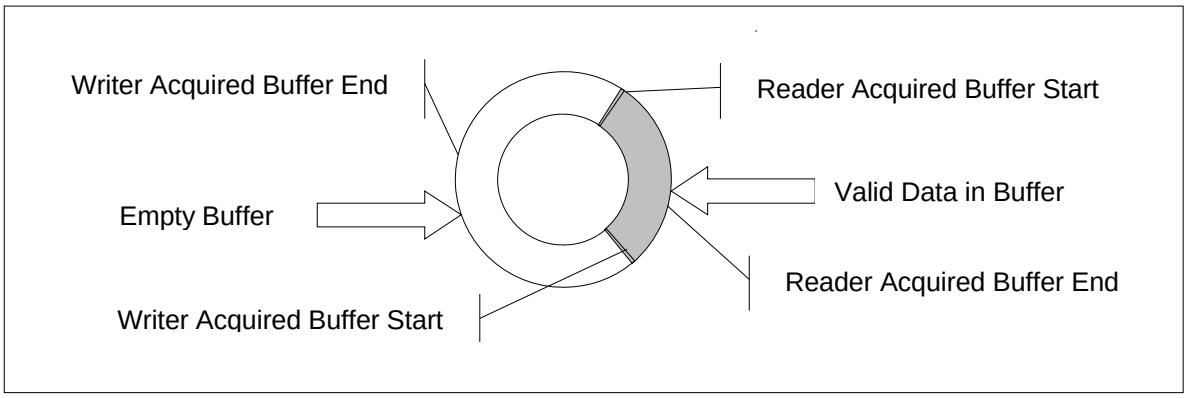

Figure 7. RingIO overview

The specific services provided by this module are:

1. Create a ring buffer created within the shared memory. The RingIO is identified by a unique name. The reader and writer of the ring buffer can be on different processors.
2. Writer and reader can open a handle to the RingIO in a specific mode, which can be used for all further accesses to the RingIO.
3. Writer can acquire empty regions of memory within the data buffer. The contents of the acquired region are committed to memory when the data buffer is released by the writer.
4. Reader can acquire regions of memory within the data buffer with valid data within them. On releasing the acquired region, the contents of this region are marked as invalid.
5. Writer and reader can operate completely asynchronously with each other.
6. The buffers are acquired in sequence. The size of released data need not match the sizes in which the data was acquired. Data is released to the buffer by specifying only the size to be released. The buffer pointer is not specified.
7. Attributes can be synchronous transferred with data. End of Stream (EOS), Time Stamps, Stream offset etc. are examples of such attributes and these can be associated with offsets in the ring buffer. Writer sets attributes, and reader gets the attributes. The attributes may be fixed or of variable size.
8. Cancel the acquired buffer.
9. Flush the contents of the ring buffer. The behavior of flush is different based on whether writer or reader is flushing the data/attributes, and also depends on the type of flush requested.
10. Writer and/or reader can register for notification with a callback function. The notification is received when certain specific conditions as required by the different notification types are met.
11. Helper functions to get information about the current state of the RingIO.

5.2 Generic features

- A client using RingIO is a single unit of execution. It may be a process or thread on the GPP or the DSP.
- The RingIO instance can be created between a client on the ARM and a client on the DSP or between two DSP clients.
- Either the reader or writer can create or delete the RingIO instance.
- The RingIO instance should be created in a shared memory region which can be accessed directly by both the reader and the writer.
- Both the reader and the writer need to open the RingIO instance and get a handle. Any data access on the RingIO instance should be made using these handles.
- Each RingIO can have a single writer client and a single reader client. A RingIO handle may not be shared between multiple clients on the GPP or DSP. For example, the following scenario is not permitted: One thread acquires from the RingIO, passes the buffer pointer to another thread, which then releases the buffer. This scenario is a multi-reader/writer scenario, which is not supported.
- Each RingIO instance is associated with a unique RingIO name. This RingIO name is specified while creating, opening and deleting the RingIO.
- The RingIO client can be closed only if there is no currently acquired data or attributes. If there is any unreleased data or attributes, they must be released or cancelled before the RingIO client can be closed.
- The RingIO can be deleted only when both reader and writer clients have successfully closed their RingIO clients.
- Each RingIO instance has an associated footer area, if configured. The foot-buffer can be configured to be of zero size if not required.
- The RingIO data and attribute buffer sizes must comply with any constraints imposed by the pool that they are specified to be allocated from. For example, for the Shared Memory Pool, the buffer sizes must be aligned to DSP cache line.

5.2.1 RingIO buffers

There are three types of RingIO buffers:

- Data buffer
- Attribute buffer
- Foot buffer

The size of the RingIO buffers depends on the application's need. The size of the buffers is specified while creating the RingIO, as part of RingIO creation attributes:

```
ringIOAttrs.dataBufSize = 0x40000 ;
ringIOAttrs.attrBufSize = 0x1000 ;
ringIOAttrs.footBufSize = 0x100 ;
```

- The RingIO footbuffer is required to be physically contiguous with the data buffer. Hence when specifying pool requirements for the buffers, a size of (dataBufSize + footBufSize) must be configured.

5.2.2 DSP cache-related information

- On the DSP-side, cache-related flags are provided to the writer and reader clients while opening the RingIO. These flags enable the user to get the maximum performance from the system and customize it for their own use. Separate cache flags are available for:
  - Control structures
  - Data buffer
  - Attribute buffer
- These flags indicate whether cache coherence is to be performed for the RingIO control structures, data buffer or attribute buffer. The flags need not be specified when opening the RingIO for the following application scenarios:
  - DSP-DSP RingIO
  - If RingIO control structures are specified to be placed into an internal memory pool, cache flag need not be specified for control structures.
  - If the RingIO data buffer is specified to be placed into an internal memory pool, cache flag need not be specified for data buffer.
  - If the RingIO attribute buffer is specified to be placed into an internal memory pool, cache flag need not be specified for attribute buffer.

5.3 Acquiring and releasing data

- The writer/reader client can acquire data buffers of any arbitrary size. RingIO does not maintain the acquired data as separate buffers, but as the complete acquired size.
  - Each buffer received from the acquire call is guaranteed to be a contiguous data buffer.
  - However, buffers received from multiple consecutive acquire calls may not be contiguous.
  - No assumption should be made that consecutively acquired buffers are contiguous in memory.
  - The writer/reader client can acquire multiple buffers and release the size completely, or in smaller chunks of varying sizes.
- The data is released into the RingIO by specifying the size to be released. Buffer pointers are not provided to the release call.
- As long as the size to be released does not exceed the total acquired size, the data can be released in any granularity. The sequence of release calls does not need to match the acquire calls.
- Cancel: Any acquired data that is not required can be cancelled back to the RingIO through the RingIO_cancel () API.
  - The cancel call removes all acquired but un-released data from the RingIO for the calling client.
  - In case of writer, any attributes that were set within this acquired but un-released region are also removed.
  - In case of reader, any attributes that were removed within the acquired region are replaced back into the RingIO.

5.3.1 Writer

- The writer writes data into the RingIO data buffer by first acquiring a contiguous data buffer, writing data into the acquired buffer, and then releasing the filled up data to the RingIO.
- The behavior of acquire varies depending on the NEED_EXACT_SIZE specified while opening the writer client. The NEED_EXACT_SIZE flag indicates whether the writer always needs buffers only of a specific size, and buffers of lesser size are not acceptable.
  - NEED_EXACT_SIZE is TRUE
    - If the requested empty size is not available within the RingIO as a contiguous data buffer, error is returned.
    - If the requested empty size is not available till the end of the RingIO buffer, but is available from the top of the buffer, a wraparound occurs, and a contiguous buffer is returned from the top of data buffer.
  - NEED_EXACT_SIZE is FALSE: If the requested buffer size is not available, RingIO returns the amount of empty contiguous data buffer that is available till the end of the data buffer, with a status code indicating this.
- Five different types of notification mechanisms are supported. Details of the notification types are present in later sections.
- The writer can flush the data that it has written into the RingIO in two different modes. In the case of hard-flush, all data and associated attributes present in the RingIO will be removed. In the case of soft-flush, all data and associated attributes after the first readable attribute will be flushed, and the attribute is also removed.

5.3.2 Reader

- The reader reads data from the RingIO data buffer by first acquiring a contiguous data buffer, reading data from the acquired buffer, and then releasing the empty buffer to the RingIO.
- The behavior of acquire varies depending on the NEED_EXACT_SIZE specified while opening the reader client. The NEED_EXACT_SIZE flag indicates whether the reader always needs buffers only of a specific size., and buffers of lesser size are not acceptable.
  - NEED_EXACT_SIZE is TRUE
    - If the requested valid size is not available within the RingIO as a contiguous data buffer, error is returned.
    - If the requested empty size is not available till the end of the RingIO buffer, but is available from the top of the buffer, the behavior varies depending on whether a foot-buffer has been configured.
      - If non-zero size foot-buffer is configured, the required amount of valid data is copied from the top of the data buffer into the foot-buffer (assuming foot-buffer size is sufficient). A contiguous data buffer is then returned to the user as requested. Further acquires will happen

from the specific offset from the top of the buffer. If foot-buffer size is not sufficient to return a contiguous data buffer of specified size, error is returned.

- If foot-buffer is not configured, error is returned in this case.

- NEED_EXACT_SIZE is FALSE: If the requested buffer size is not available, RingIO returns the amount of valid contiguous data buffer that is available till the end of the data buffer, with a status code indicating this. Foot-buffer is not used in this scenario.

§ Five different types of notification mechanisms are supported. Details of the notification types are present in later sections.

§ The reader can flush the data that is available from the RingIO in two different modes. In the case of hard-flush, all data and associated attributes present in the RingIO will be removed. In the case of soft flush, all data and associated attributes before the first readable attribute will be flushed

5.4 Attributes

5.4.1 Generic information

§ Attributes are used to communicate in-band information from the writer to the reader.

§ Typical attributes could be the EOS marker at the end of the stream that's being written, or an attribute to indicate changes in the stream's status.

§ Attributes can be of two types

- Fixed attributes: Fixed attributes have an attribute type and an optional parameter

- Variable attributes: Variable attributes can be provided a data buffer as payload data in addition to the attribute type and the optional parameter. The attributes are copy-based. The information in writer-provided buffer is copied into the attribute buffer. The size of provided buffer in variable attributes must be a multiple of 4 bytes.

5.4.2 Setting attributes

§ Attributes can be set by the writer only on a data buffer that has been acquired. This means that, if the writer has acquired a buffer of size x, attributes can be set at any of offset position between 0 and x (inclusive).

§ The only exception to the above rule is if the writer wishes to set an attribute when no data has been acquired. In this case, the writer can set attributes at the next write location i.e. offset 0 in the buffer that is going to be acquired. Attempts to set attributes at any other offset are ignored, and the attributes get set at offset 0.

§ When the writer writes attributes for the data buffer it has acquired, it should set attributes in the increasing order of buffer offsets. Setting attributes in any arbitrary order can lead to undefined behavior.

§ The writer commits attributes to the attribute buffer when the associated write buffer is released. Any attributes set when writer has no acquired data are released immediately.

5.4.3 Getting attributes

- The attributes written by the writer should be "read" before the reader can read any more data after the offset at which the attribute is set.
- If the reader could not read data due to presence of attributes at the current read location, an error code mentioning the presence of an attribute is returned.
- When a variable attribute is being read, a valid buffer must be provided to the getAttribute function. The attribute information is copied into this application buffer.
- Attributes are removed from the attribute buffer when the reader releases the data buffer that contains the associated attributes or the writer flushes valid data which will clear associated attributes.
- Fixed and Variable attributes can be set and received using different APIs. In case a fixed attribute get function is called when a variable attribute is present, an error code is returned informing of the presence of the variable attribute.

5.4.4 Constraints

Setting attributes when RingIO is in an incorrect state

An attempt to set an attribute shall fail with error `RINGIO_EWRONGSTATE`, if setting the attribute would fall into the reader region. This can happen for the following scenarios:

- The buffer is completely full. In this case, attribute can only be set at offset 0. But offset 0 falls into reader region.
- The buffer is completely acquired by the writer. Part or none of this buffer may have be released. Writer is attempting to set an attribute at the end of its acquired range. In this case, end of writer buffer is the same as beginning of reader buffer.

If the reader has acquired and released some data, resulting in its moving further such that its acquire start is not at the same location where writer may be able to set an attribute, the above conditions do not hold true, and the attribute is allowed to be set.

Ensuring the constraint:

If such a scenario occurs, the application must wait/poll till the reader moves ahead so that the attribute would not fall in its region. This can be done in two ways:

1. Wait on a semaphore for notification that it is safe to set the attribute: For this constraint, the reader release data is the trigger point. So if the writer can set notifier to be notified when reader releases data, writer can have a semaphore wait that gets posted when the notification comes.
    - Watermark needs to be set at 1 byte, since that's the minimum that's needed for reader to move forward.
    - Notification type must be `RINGIO_NOTIFICATION_HDWRFIFO_ONCE`, `RINGIO_NOTIFICATION_ALWAYS` or `RINGIO_NOTIFICATION_HDWRFIFO_ALWAYS`. This may interfere with any other notification mechanisms, since application may have a different threshold or notification type for the other generic activities. So it may

not be always possible to implement this type of notification mechanism for attribute buffer. If this method is not possible, then polling type of notification must be used.

2. Poll till `RingIO_setAttribute/RingIO_setvAttribute` returns success: It is preferable to have a small sleep in between successive calls to the `setattribute` APIs to ensure that the CPU is not loaded and other threads get a chance to run.

5.5 Foot-buffer

1. A foot-buffer is a buffer that is configured during creation of the RingIO. The size of the foot-buffer to be used, if any, is mentioned in the RingIO attributes specified while creating the RingIO.
2. The foot-buffer memory is reserved, contiguously starting from the end of the RingIO data buffer.
3. The foot-buffer comes into the picture only for RingIO reader. It is not used/written into by the RingIO writer.
4. For a RingIO reader, the following scenario causes foot-buffer to be used:
   - § Reader attempts to acquire a buffer size that is more than the contiguous size available till the end of the RingIO data buffer.
   - § Valid size is present from the top of the RingIO data buffer
   - § Foot-buffer is configured to be of a non-zero size.
   - § In this scenario, the RingIO_acquire call for reader decides the size of contiguous valid buffer available as a minimum of:
     - Total valid size
     - Contiguous buffer size available -- size till the end of RingIO data buffer + early end buffer (if any) + foot-buffer
     - Size till the first attribute that can be read by the reader

   Based on this, there is a memory copy from the top of the RingIO data buffer into the early end buffer & foot-buffer, and the pointer to the contiguous buffer is returned to the reader.

   After this, the reader acquire pointer will be reset to within the RingIO data buffer (from top), and subsequent acquires will be made from within the RingIO data buffer.
5. Without foot-buffer, application design in certain scenarios may get complicated. This is elaborated below in the use-case scenario.

5.5.1 Use Case Scenarios

5.5.1.1 *Scenario 1*

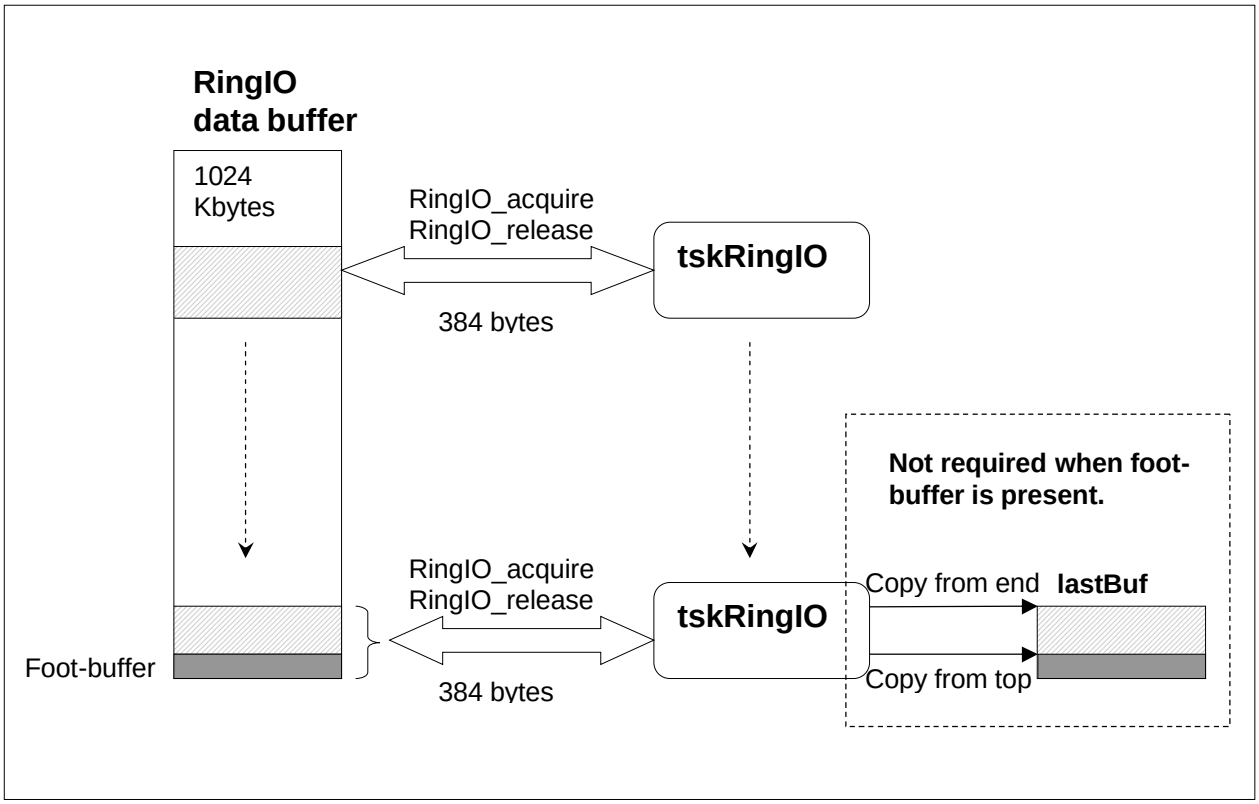

Figure 8. Foot-buffer Use-Case Scenario 1

Without foot-buffer

Consider a RingIO configured with data buffer size 1024K. No foot-buffer is configured.

1. Reader requires 384 bytes frame size. Note that 1024K is not a multiple of 384.
2. Writer acquires and releases large chunks of data as it is available, and fills up the data buffer.
3. The reader acquires and releases 384 bytes at a time.
4. When the reader reaches the end of the buffer, it has already used up 1048320 bytes (1024K – 256).
5. Now the reader needs another 384 bytes. However, only 256 bytes are available at the end of the buffer as a contiguous buffer.
6. The 256 bytes at the end of the data buffer cannot be ignored. The Reader needs to now allocate its own 384 byte buffer, acquire the 256 bytes from the RingIO data buffer, copy the 256 bytes into its own buffer. Then acquire another 128 bytes from the top of the data buffer, and copy it after the existing 256 bytes into its own buffer. This gives it the required 384 bytes.
7. This special implementation needs to be done by the application writer to ensure correct behavior.

With foot-buffer

When foot-buffer is copied, the complete special application implementation in step 6 above can be avoided.

In step 6:

The reader still makes a request for 384 bytes. Since only 256 bytes are available at the end, RingIO internally copies 128 bytes from the top of the buffer into the foot-buffer and provides a contiguous buffer to the reader.

When the reader releases the buffer, the internal RingIO data structures are updated to ensure that the next request goes to the start of the buffer (with 128 byte offset) instead of continuing into the foot-buffer.

Due to this, a minimal foot-buffer size of 128 bytes only is sufficient to ensure application simplicity.

Points to be considered

Only the required foot-buffer size of 128 bytes should be configured in above case. Configuring a larger foot-buffer will not result in additional efficiency. It will result in memory wastage.

5.5.1.2 *Scenario 2*

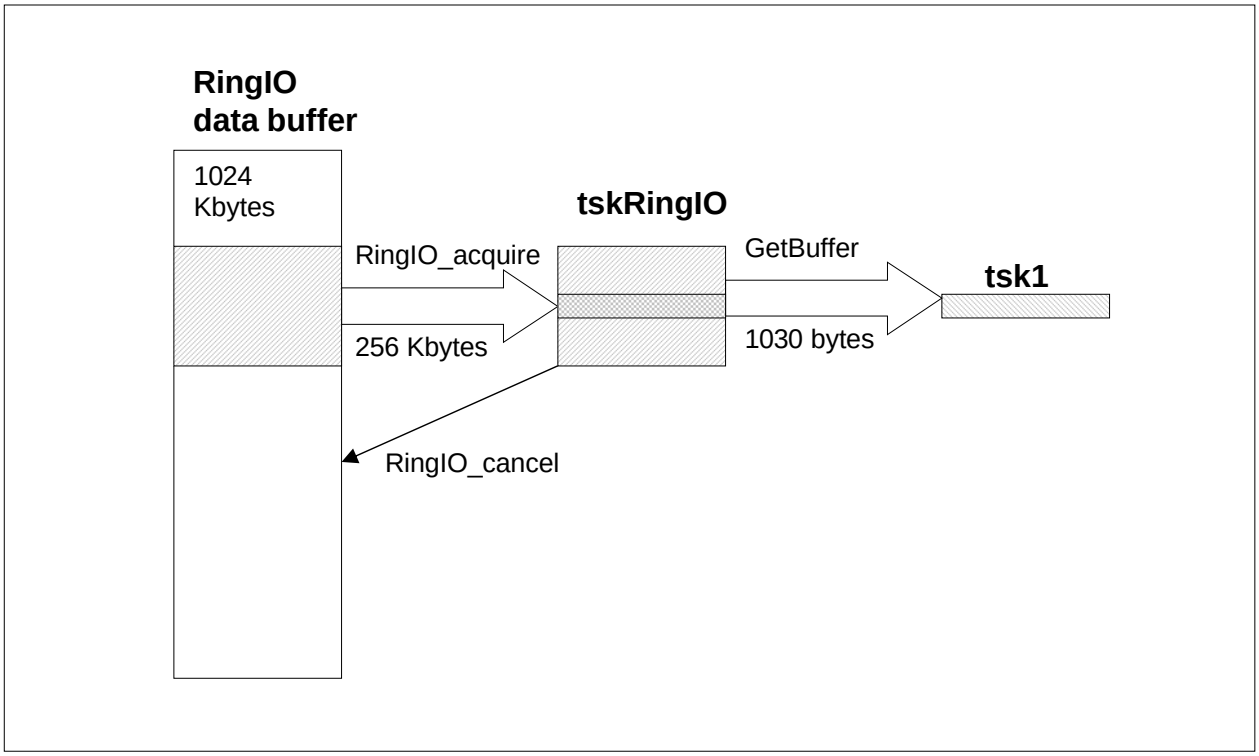

Figure 9. Foot-buffer Use Case Scenario 2

Without foot-buffer

Consider a RingIO data buffer of size 1024K. No foot-buffer is configured.

1. A task (tskRingIO) interacts with RingIO in reader mode to acquire and release buffers. It attempts to always acquire 256K buffers (ringIOBuf) with NEED_EXACT_SIZE false.

Other tasks (tsk1) interact with tskRingIO to satisfy their buffer needs. For example, tsk1 always asks for fixed size 1030 byte buffers. tsk1 does not interact with RingIO.

2. The tskRingIO buffer ringIOBuf provides a simple size management for its acquired buffer size. The size available within ringIOBuf is reduced whenever it provides a buffer to tsk1. When the size reduces to less than 1030 (e.g. 524) and tsk1 buffer get fails, tskRingIO releases the used buffer (size 261620) and cancels the remaining unused part (size 524) to the RingIO.
3. This works fine for the first few times tskRingIO acquires the 256K buffers, till it reaches the end of RingIO data buffer. Consider that the last cancel has now cancelled 524 bytes to the RingIO buffer, resulting in the reader's acquire start being at 524 bytes above its physical end. Now if tskRingIO attempts to acquire 256K, it will get only 524 bytes again of contiguous buffer till the end of RingIO data buffer, even though valid data may be available from the top of RingIO buffer.
4. The tsk1 buffer get again fails (524 < 1030), tskRingIO again cancels the buffer to RingIO, and this keeps happening in a loop, stalling the system.

With foot-buffer

In this scenario, foot-buffer is very useful. By configuring the foot-buffer of size 1030, this situation can be avoided.

In step 2, if tskRingIO cancels 524 to the RingIO data buffer, and sufficient valid size is available at the top of RingIO data buffer, RingIO_acquire for tskRingIO will result in copying valid data of size 1030 from top of buffer into foot-buffer. Then this buffer of size 1554 is returned to tskRingIO.

tsk1 buffer get passes, and tskRingIO cancels remaining amount (524) to the RingIO.

However, since the RingIO data buffer end boundary has now been crossed, the reader acquire start has been reset to 524 bytes from top of RingIO data buffer, and next acquire will return data from the top of the buffer.

Points to be considered

The following issues are seen if applications unnecessarily use a large foot-buffer:

The foot-buffer size to be used must be tuned to the application's needs. Using a larger foot-buffer size will not give any additional advantages to the system. On the other hand, it will degrade the system performance.

For example, if a 256K foot-buffer is used, assuming sufficient valid size, it will result in entire 256K buffer being copied from top of RingIO data buffer into the foot-buffer. This is unnecessary, and will degrade the performance, and zero-copy behavior is lost.

5.6 Notification

The notification mechanism as well as other configuration parameters for the notification can be set by the reader or writer through an API call to set the notifier. Parameters that can be configured include the notification type, watermark, callback function, and fixed parameter to the callback function.

Five different types of notification are supported:

5.6.1 RINGIO_NOTIFICATION_NONE

No notification is required.

5.6.2 RINGIO_NOTIFICATION_ALWAYS

5.6.2.1 *Description*

- The notification is enabled when an attempt to acquire data by the client has failed.
- Once enabled, the notification remains enabled till:
  - For writer client, empty data size falls below the watermark.
  - For reader client, valid data size falls below the watermark.

  At this point, the notification is disabled again. Only a RingIO_release call will disable the notification. RingIO_cancel and RingIO_flush will not disable the notification.
- Notifications are sent each time when the other client releases data, as long as the data size is above the watermark:
  - Empty data size for writer – This condition can be met in the functions RingIO_release and RingIO_flush.
  - Valid data size for reader
- Cancel call does not enable or disable notification.
- Flush call does not enable or disable notification. Flush called by the reader client may cause empty data size to fall above the watermark and cause a notification to be sent to the writer client.

5.6.2.2 *Examples*

Scenario 1: Reader notification

RingIO data buffer size = 1MB

Application requirements:

1. RingIO reader needs at least 16K valid buffer size to be able to start/continue its processing.
2. RingIO reader does not need to be notified as long as it is not acquiring any data. Writer may continue to release data, but only when reader has started acquiring data and has failed once, it needs to be notified. Till then, it is not interested in writer's data releases.
3. Once the reader's acquire has failed, it needs to be notified for each writer release as long as the valid buffer size is above its watermark.

Scenario:

1. Initial state is RingIO is empty.
2. RingIO reader sets notification for 16K watermark with RINGIO_NOTIFICATION_ALWAYS type.
3. Writer starts releasing data. 64K valid data is available in the RingIO. No notification is received for reader.

4. At some point, reader starts acquiring data. The initial acquires pass (e.g. 56K).
5. RingIO reader attempts acquire (e.g. 16K). Acquire fails. Notification gets enabled.
6. RingIO reader now waits for notification.
7. RingIO writer releases 16K. Notification is sent to reader.
8. RingIO reader wakes up and can now acquire data.
9. Writer releases another 8K. Notification is again sent to reader (valid data = 24K is above reader watermark of 16K).
10. Writer releases another 8K. Notification is again sent to reader (valid data = 32K is above reader watermark of 16K).
11. Reader acquires 16K. Valid data = 16K. Notification is still enabled. If writer releases any data, reader will get notified.
12. Reader acquires 8K. Valid data = 8K, which is below reader watermark of 16K. Notification gets disabled.
13. Writer releases 16K. Notification is not sent to reader (valid data = 24K is above reader watermark of 16K, but notification is disabled).
14. Now, notification gets enabled again only if reader attempt to acquire data fails. For example, if reader attempts to acquire 32K.

Scenario 2: Writer notification

RingIO data buffer size = 1MB

Application requirements:

- § RingIO writer needs at least 64K empty buffer size to be able to continue filling the RingIO buffer with valid data.
- § RingIO writer does not need to be notified as long as it is able to succesfully acquire any empty buffer. Reader may continue to release empty buffer, but only when writer acquire fails, it needs to be notified. Till then, it is not interested in reader's buffer releases.
- § Once the writer's acquire has failed, it needs to be notified for each reader release as long as the empty buffer size is above its watermark.

Scenario:

1. RingIO writer sets notification for 64K watermark with RINGIO_NOTIFICATION_ALWAYS type.
2. Writer and reader are acquiring and releasing data at their own processing speeds.
3. Writer is faster than reader. At some point, the RingIO data buffer gets filled up such that the empty buffer size falls below the 64K watermark set by writer.
4. Writer attempts acquire. Acquire fails. Notification gets enabled.
5. RingIO writer now waits for notification.

6. RingIO reader releases 64K buffer such that the empty buffer size goes above the 64K watermark set by writer. Notification is sent to writer. Empty buffer size = 64K.
7. RingIO writer wakes up and can now acquire empty buffer.
8. Reader releases another 8K. Notification is again sent to writer (empty buffer = 72K is above writer watermark of 64K).
9. Reader releases another 8K. Notification is again sent to writer (empty buffer = 80K is above writer watermark of 64K).
10. Writer acquires 16K. Empty buffer = 64K. Notification is still enabled. If reader releases any buffer, writer will get notified.
11. Writer acquires 8K. Empty buffer = 56K, which is below writer watermark of 64K. Notification gets disabled.
12. Reader releases 16K. Notification is not sent to reader (empty buffer = 72K is above writer watermark of 64K, but notification is disabled).
13. Now, notification gets enabled again only if writer attempt to acquire data fails. For example, if writer attempts to acquire 80K.

5.6.3 RINGIO_NOTIFICATION_ONCE

5.6.3.1 *Description*

- The notification is enabled when an attempt to acquire data by the client has failed.
- The notification is sent when the other client releases data, when the below condition is true:
  - For writer client, empty data size is above the watermark – This condition can be met in the functions RingIO_release and RingIO_flush.
  - For reader client, valid data size is above the watermark.

  As soon as the notification is sent, it is disabled.
- The notification is re-enabled, only when the first condition is met again (acquire attempt fails).
- Cancel call does not enable or disable notification.
- The notification is disabled only once the notification is sent by the other client.
- Flush call does not enable or disable notification. Flush called by the reader client may cause empty data size to fall above the watermark and cause a notification to be sent to the writer client.

5.6.3.2 *Examples*

Scenario 1: Reader notification

RingIO data buffer size = 1MB

Application requirements:

- RingIO reader needs at least 16K valid buffer size to be able to start/continue its processing.

- § RingIO reader does not require notification as long as valid data is above watermark. It needs notification only if its attempt to get required amount of data fails.

Scenario:

1. Initial state is RingIO is empty.
2. RingIO reader sets notification for 16K watermark with RINGIO_NOTIFICATION_ONCE type. Valid data size = 0K.
3. RingIO reader attempts acquire. Acquire fails. Notification gets enabled.
4. RingIO reader now waits for notification.
5. RingIO writer releases 64K. Notification is sent to reader. Valid data size = 64K.
6. RingIO reader wakes up and can now acquire data.
7. Reader acquires 16K data. Notification gets disabled because the acquire is successful. Valid data size = 48K.
8. Writer releases 8K data. Notification is not sent to reader because notification is disabled. This is even though valid data size = 64K is above reader watermark of 16K.
9. Reader acquires 56K buffer. Acquire is successful. Valid data size = 8K.
10. Reader attempts to acquire 16K buffer. Acquire fails. Notification is re-enabled.
11. Now if writer releases 8K buffer, valid data size = 16K matches reader watermark, and notification is sent to reader.
12. This continues ... On subsequent writer releases, notification will not be sent to reader. If reader acquire fails, then it can wait for notification again. It will get notified if writer releases enough buffer to go above the reader watermark for valid data size.

Scenario 2: Writer notification

RingIO data buffer size = 1MB

Application requirements:

- § RingIO writer needs at least 64K empty buffer size to be able to continue filling the RingIO buffer with valid data.
- § RingIO writer does not require notification as long as valid data is above watermark. It needs notification only if its attempt to get required amount of data fails.

Scenario:

1. RingIO writer sets notification for 64K watermark with RINGIO_NOTIFICATION_ONCE type.
2. Writer and reader are acquiring and releasing data at their own processing speeds.
3. Writer does not receive notification from reader even though empty buffer size is above watermark (64K).

4. Writer is faster than reader. At some point, the RingIO data buffer gets filled up such that the empty buffer size falls below the 64K watermark set by writer.
5. Writer attempts acquire. Acquire fails. Notification gets enabled.
6. RingIO writer now waits for notification.
7. RingIO reader releases 128K buffer. Notification is sent to writer. Empty buffer size = 128K.
8. RingIO writer wakes up and can now acquire empty buffer.
9. Writer acquires 32K buffer. Notification gets disabled because the acquire is successful. Empty buffer size = 96K.
10. Reader releases 8K data. Notification is not sent to reader because notification is disabled. This is even though empty buffer size = 104K is above writer watermark of 64K.
11. Writer acquires 48K buffer. Acquire is successful. Empty buffer size = 48K.
12. Writer attempts to acquire 64K buffer. Acquire fails. Notification is re-enabled.
13. Now if reader releases 16K buffer, empty buffer size = 64K matches writer watermark, and notification is sent to writer.
14. This continues ... On subsequent reader releases, notification will not be sent to writer. If writer acquire fails, then it can wait for notification again. It will get notified if reader releases enough buffer to go above the writer watermark for empty buffer size.

5.6.4 RINGIO_NOTIFICATION_HDWRFIFO_ALWAYS

5.6.4.1 *Description*

- Notifications are sent each time when the other client releases data, as long as the data size is above the watermark:
  - Empty data size for writer - This condition can be met in the function RingIO_release and RingIO_flush.
  - Valid data size for reader
- This notification is always enabled. Unlike RINGIO_NOTIFICATION_ALWAYS, this notification does not require buffer to get full/empty or acquire to fail to get enabled.
- Cancel call does not enable or disable notification.
- Flush call does not enable or disable notification. Flush called by the reader client may cause empty data size to fall above the watermark and cause a notification to be sent to the writer client.

5.6.4.2 *Examples*

Scenario 1: Reader notification

RingIO data buffer size = 1MB

Application requirements:

- RingIO reader needs at least 16K valid buffer size to be able to start/continue its processing.

- § RingIO reader needs to be notified for each writer release as long as the valid buffer size is above its watermark.

Scenario:

1. Initial state is RingIO is empty.
2. RingIO reader sets notification for 16K watermark with RINGIO_NOTIFICATION_HDWRFIFO_ALWAYS type.
3. Writer releases 8K valid data. Notification is not sent to reader because valid data size = 8K is less than reader's watermark (16K).
4. Writer releases another 8K valid data. Notification is sent to reader because valid data size = 16K matches reader's watermark.
5. Writer releases another 16K valid data. Notification is sent to reader because valid data size = 32K is more than reader's watermark.
6. Reader starts acquiring data. Reader acquires 24K. Valid data size = 8K falls below reader's watermark. Notification gets disabled.
7. Writer releases 4K valid data. Notification is not sent to reader because valid data size = 12K is less than reader's watermark.
8. Writer releases 8K valid data. Notification is sent to reader because valid data size = 24K is more than reader's watermark.
9. This continues ... as long as valid data size is equal or more than reader's watermark, every writer release sends notification to reader. Writer releases do not send notification to reader if valid data size is below reader's watermark even after the writer release.

Scenario 2: Writer notification

RingIO data buffer size = 1MB

Application requirements:

- § RingIO writer needs at least 64K empty buffer size to be able to continue filling the RingIO buffer with valid data.
- § RingIO writer needs to be notified for each reader release as long as the empty buffer size is above its watermark.

Scenario:

1. RingIO writer sets notification for 64K watermark with RINGIO_NOTIFICATION_HDWRFIFO_ALWAYS type.
2. Writer and reader are acquiring and releasing data at their own processing speeds.
3. Empty buffer size = 128K.
4. Reader releases 8K empty buffer. Notification is sent to writer because empty buffer size = 132K is more than writer's watermark (64K).
5. Writer is faster than reader. At some point, the RingIO data buffer gets filled up such that the empty buffer size falls below the 64K watermark set by writer. Empty buffer size = 32K.

6. Reader releases 8K empty buffer. Notification is not sent to reader because empty buffer size = 40K is less than writer's watermark.
7. Reader releases 24K empty buffer. Notification is sent to writer because valid data size = 64K matches the writer's watermark.
8. Reader releases another 8K empty buffer. Notification is sent to writer because valid data size = 72K is more than writer's watermark.
9. Writer starts acquiring the buffer. Writer acquires 24K. Empty buffer size = 48K falls below writer's watermark. Notification gets disabled.
10. Reader releases 8K empty buffer. Notification is not sent to writer because empty buffer size = 56K is less than writer's watermark.
11. Reader releases 16K empty buffer. Notification is sent to writer because empty buffer size = 72K is more than writer's watermark.
12. This continues ... as long as empty buffer size is equal or more than writer's watermark, every reader release sends notification to writer. Reader releases do not send notification to writer if empty buffer size is below writer's watermark even after the reader release.

5.6.5 RINGIO_NOTIFICATION_HDWRFIFO_ONCE

5.6.5.1 *Description*

This notification type will send a notification only once when a low watermark condition is satisfied and then it is disabled.

- Unlike RINGIO_NOTIFICATION_ONCE, this notification does not require buffer to get full/empty or acquire to fail to get enabled.
- The notification is sent when the other client releases data, when the below condition is true:
  - For writer client, empty data size is above the watermark - This condition can be met in the function RingIO_release and RingIO_flush.
  - For reader client, valid data size is above the watermark.

  As soon as the notification is sent, it is disabled.
- The notification is re-enabled when the data size crosses the watermark:
  - For writer client, empty data size falls below the watermark.
  - For reader client, valid data size falls below the watermark.
- Cancel call will affect the notification state. If the notification has been enabled earlier either because of a failed acquire call or a low watermark condition is satisfied, this notification will be disabled if the low watermark condition is no longer true.
  - The notification will be disabled when the data size crosses the watermark:
    1. For writer client, empty data size falls above the watermark.
    2. For reader client, valid data size falls above the watermark.
- Flush call will affect the notification state.

- For writer client, the notification will be disabled when the data size falls above the watermark i.e. empty data size is greater than the watermark.
- For reader client, the notification will be enabled when the data size falls below the watermark i.e. valid data size is lesser than the watermark.

5.6.5.2 *Examples*

Scenario 1: Reader notification

RingIO data buffer size = 1MB

Application requirements:

- § RingIO reader needs at least 16K valid buffer size to be able to start/continue its processing.
- § RingIO reader does not require notification as long as valid data is above watermark. It needs notification only if the valid data size falls below its watermark level.

Scenario:

1. Initial state is RingIO is empty.
2. RingIO reader sets notification for 16K watermark with RINGIO_NOTIFICATION_HDWRFIFO_ONCE type. Valid data size = 0K.
3. RingIO writer releases 8K valid data. Notification is not sent to reader. Valid data size = 8K.
4. Writer releases another 8K valid data. Notification is not sent to reader even though valid data size = 16K matches reader's watermark.
5. Writer releases another 16K valid data. Notification is not sent to reader even though valid data size = 32K is more than reader's watermark.
6. Reader starts acquiring data. Reader acquires 24K. Valid data size = 8K falls below reader's watermark. Notification gets enabled.
7. Writer releases 4K valid data. Notification is not sent to reader because valid data size = 12K is less than reader's watermark.
8. Writer releases 8K valid data. Notification is sent to reader because valid data size = 24K is more than reader's watermark. Then notification gets disabled.
9. This continues ... On subsequent writer releases, notification will not be sent to reader as long as the valid data size remains above the reader's watermark. When reader acquires results in valid data size falling below watermark, notification gets enabled again. As soon as one notification is sent to reader, it gets disabled again.

Scenario 2: Writer notification

RingIO data buffer size = 1MB

Application requirements:

- § RingIO writer needs at least 64K empty buffer size to be able to continue filling the RingIO buffer with valid data.

- § RingIO writer does not require notification as long as empty buffer size is above watermark. It needs notification only if the empty buffer size falls below its watermark level.

Scenario:

1. RingIO writer sets notification for 64K watermark with RINGIO_NOTIFICATION_HDWRFIFO_ONCE type.
2. Writer and reader are acquiring and releasing data at their own processing speeds.
3. Writer does not receive notification from reader even though empty buffer size is above watermark (64K).
4. Writer is faster than reader. At some point, the RingIO data buffer gets filled up such that the empty buffer size falls below the 64K watermark set by writer. At this point, notification gets enabled. Empty buffer size = 32K.
5. Reader releases 8K empty buffer. Notification is not sent to reader because empty buffer size = 40K is below reader's watermark.
6. Reader releases 32K empty buffer. Notification is sent to reader because empty buffer size = 72K is more than reader's watermark. Then notification is disabled.
7. Reader releases another 16K valid data. Notification is not sent to reader even though empty buffer size = 88K is more than reader's watermark.
8. Writer starts acquiring data. Writer acquires 48K. Empty buffer size = 40K falls below writer's watermark. Notification gets enabled.
9. Reader releases 16K valid data. Notification is not sent to reader because empty buffer size = 56K is less than writer's watermark.
10. Reader releases 8K valid data. Notification is sent to reader because empty buffer size = 64K matches the writer's watermark. Then notification gets disabled.
11. This continues ... On subsequent reader releases, notification will not be sent to writer as long as the empty buffer size remains above the writer's watermark. When writer acquire results in empty buffer size falling below watermark, notification gets enabled again. As soon as one notification is sent to writer, it gets disabled again.

6 Multi-DSP support

6.1.1 Overview

DSPLink supports multiple DSPs connected to the master GPP processor. With this feature, GPP side process/application can communicate with the multiple DSPs connected to the GPP over heterogeneous physical links. Example reference ports have also be included for multi-DSP configurations

1. Linux PC connected to two DM6437 devices over PCI
2. DRA44x connected to external DM6437 over VLYNQ

6.2 Features

6.2.1 Configuration of DSPLink for Multi-DSP.

DSPLink static build configuration allows users to configure the DSPLINK for multi DSP usage. With the static build configuration, users can select the number of DSPs, supported OS on each DSP etc. Refer to user guide for the details.

6.2.2 Linux PC connected to multiple DM6437 devices over PCI

DSPLink provides support for Linux PC connected to multiple DM6437devices over PCI.

Application on GPP can communicate with all the DSPs or any DSP that is configured in the system. Application on any DSP can communicate with GPP and also can communicate with the other DSPs via GPP. Note that DSP side application can not directly communicate the other DSPs but it can send the information to GPP and GPP can transfer the information to other DSP.

6.2.3 DRA44x connected to external DM6437 over VLYNQ

DSPLink provides support for DRA44x connected to DM6437 device over VLYNQ.

Application on GPP can communicate with all the DSPs (DM6437GEM and DRA44xGEM) or any DSP that is configured in the system. Application on any DSP can communicate with GPP and also can communicate with the other DSPs via GPP. Note that DSP side application can not directly communicate the other DSPs but it can send the information to GPP and GPP can transfer the information to other DSP.

6.2.4 Configuration changes

6.2.4.1 *Dynamic configuration*

New dynamic configuration files

The dynamic configuration has been enhanced for multi-DSP support. Previously, only two dynamic configuration files were required for each platform:

- CFG_<PLATFORM>.c
- CFG_<GPPOS>.c

With multi-DSP support, this has now changed into a need for four or more dynamic configuration files. The current CFG_<PLATFORM>.c file has been split up into separate configuration files for the GPP and DSP:

- CFG_<GPP>.c

- o CFG_<DSPn>_<PHYLINK>.c (These files may be one or more based on the number of DSPs to be used in the system configuration)
- o CFG_system.c: This file contains the current configured system architecture, and is generated by the static build configuration script.
- o CFG_<GPPOS>.c: This file contains the GPP OS related configurations i.e. signal that needs to handled by link. This file is unchanged from previous release.

The CFG_system.c file is generated within the GPP temporary folder. If a temporary folder location is not specified when running the build configuration script, the default location is the same as in previous releases.

Macros for easier modifications to dynamic configuration

To enable users to easily modify the DSPLink dynamic configuration for most commonly changed fields, the following enhancements have been made:

1. On changes in number of entries in any table, the corresponding value for number of entries in the configuration now gets automatically updated. For example, on adding a new memory table entry, the MEMENTRIES field in DSP object gets updated automatically.
2. Macros have been provided for most commonly changed fields such as base addresses of memory sections, memory entry IDs and handshake poll count (set to -1 during debugging of DSP-side for infinite wait).
3. Even if ordering of the memory entries is changed (due to moving them or adding/removing new entries), the memory entry ID used by DSPLink modules for their control needs remains current by modification only in the macro for shared memory entries number.

Change in name of top-level configuration object for DSP

The top-level configuration structure LINKCFG_Config is now a generated structure, and includes configurations for the GPP as well as all DSPs. When using dynamic configuration, applications must only create DSP and GPP configurations, but the system configuration must get generated only through the static build configuration script, since it will get overwritten whenever the dsplinkcfg.pl script is run.

6.2.5 Build changes

For legacy single-DSP users, there is no change in build process for GPP or DSP. Refer user guide to user guide to use the DSPLINK in legacy mode.

6.2.5.1 *Common*

Path for generated files

It is now possible to maintain a golden DSPLink installation by providing a different path for all generated files. This path can be specified while running the static build configuration script dsplinkcfg.pl. If a path is not specified, files are generated in the same folders as in previous releases.

Removal of platform variant concept

Variant concept has been removed from DSPLink. All devices are now mentioned by their full names. This has been done to ensure that porting to a different device in DSPLink will not require any changes in generic DSPLink, such as moving a header from generic platform folder into variant-specific folder.

Also, all zero-copy and dma-copy implementations have been moved out of platform-specific folders. Due to this, device porting effort is now limited to specific folders within the DSPLink directory tree.

6.2.5.2 *GPP-side*

None.

6.2.5.3 *DSP-side*

Scripts for multi-DSP build

For multi-DSP configurations, `multimake.bat` and `multimake.sh` scripts are generated during the static build configuration step. These script files can be used to build DSP executables for all DSPs in the system in a single step. The files get generated into `$(DSPLINK)/etc/host/scripts/[Linux | msdos]`.

Generated DSP executable name

The DSP executable is now generated in a folder having the processor ID of the DSP appended to it.

6.2.6 GPP-side changes

6.2.6.1 *Changes in applications*

If using single-DSP configuration, applications need to take the following into consideration:

- If passing `PROC_Attrs` to `PROC_attach ()`, the `dspCfgPtr` field must be set to NULL. A garbage (un-initialized) value in this field shall no longer be accepted and can cause a system crash. If NULL is being passed as attributes to `PROC_attach ()`, no change is required.
- If dynamic configuration is to be used in multi-processing/multi-application scenario, `PROC_setup ()` (and correspondingly `PROC_destroy ()`) must now be called in all processes to pass the new dynamic configuration information. If this is not done, the other processes shall only get default configuration information, and updated dynamic configuration shall not be available in their user space. This may cause non-deterministic results.
- `POOL_getPoolId ()` API signature has been changed to take an additional `procId` as the first parameter.

If using multi-DSP configuration, applications must ensure the following:

- DSP dynamic configuration for each DSP must be provided to DSPLink as part of `PROC_Attrs` provided to `PROC_attach ()` API for that DSP. This ensures that it becomes possible to reconfigure one of the executing DSPs by detaching & attaching to it with a different dynamic configuration. This can be done without disturbing the execution of the other DSP.

- The GPP-side POOL IDs used for communication with the DSP must now be generated using a new API: `POOL_makePoolId`. By default, a single DSP configuration is used, and processor ID of 0 is assumed for the DSP. Hence, in single-DSP configuration, this change is not needed. However, in multi-DSP configuration, it is essential to identify the DSP with which the POOL is shared. Hence, the `POOL_makePoolId` macro can be used to generate the ID which is to be used for all POOL operations.

6.2.6.2 *Include path changes*

The include paths for GPP-side have been modified to include two folders:

- sys: Contains all system and device specific header files
- usr: Contains all user include files, which are device-independent

Most applications would only need to include header files in the usr folder. The generated `<COMPONENT>_includes.TXT` file would now contain the updated include paths. This file can be used by any users that are not using the DSPLink build system for their applications.

6.2.6.3 *New compiler defines*

The `<COMPONENT>_defines.TXT` file generated during build would contain all new compiler defines to be used for building DSPLink and applications. This can be used as a reference if a non-DSPLink based build system is being used.

6.2.7 DSP-side application changes

The DSP-side of DSPLink is fully backward compatible with the previous release. No changes are required to applications.

Application configuration in multi-DSP configuration must ensure that Message Queue transports are created as needed considering IDs of all processors in the system. This is not specific to DSPLink, but a basic DSP/BIOS MSGQ configuration requirement for multi-DSP usage.

7 Multi-application and multi-process support

7.1 Overview

Multiple applications/processes on the GPP may wish to use the services provided by DSPLINK to control and communicate with the DSP. A few possible methods are available to support multiple processes or multiple applications using DSP/BIOS™ LINK. This is applicable for operating systems such as Linux that support multi-processing.

3. Multiple independent applications using DSP/BIOS™ LINK services
4. Multiple processes within a single application.
5. Multiple threads within the processes

7.2 Features

7.2.1 Multiple independent applications

Multi-application support with DSP/BIOS™ LINK has the following features:

1. An application can be written to execute singly using DSPLINK to control and communicate with the DSP.
2. The same application can be used without any changes in the applications source code, to run simultaneously along with another application also using DSPLINK. The only consideration to be used while writing the application, is that the DSPLINK resources (e.g RingIO/MSGQ names) used by the applications must be unique for the system.
3. The applications use the same integrated DSP executable containing DSP-side content required for all the co-existing GPP-side applications.
4. If multiple different applications using DSPLINK are running on the target processor, a crash in one of these does not affect the execution of the other application.

- When multiple applications use DSPLINK, they must ensure that they pass the same dynamic configuration pointer during PROC_setup. They must also ensure that they use the same DSP executable to be loaded with PROC_load.

7.2.1.1 *Example*

Two applications contain source as follows:

```
PROC_setup (...) ;
PROC_attach (...) ;
POOL_open (poolId, poolParams) ;
PROC_load (..., dspExec, ...) ;
PROC_start (...) ;
MSGQ_transportOpen (...) ;

/* Application-specific code */

MSGQ_transportClose (...) ;
PROC_stop (...) ;
POOL_close (...) ;
PROC_detach (...) ;
```

```
PROC_destroy (...) ;
```

Both the applications can start-up and run independently if run singly. They can also start-up and run independently if run at the same time.

The behavior seen by the applications shall be the same irrespective of the sequence in which the calls actually get made to DSPLINK. An overview of the activities occurring in each API, depending on the sequence in which it gets called, is given below. It may not be necessary that the first occurrence for all APIs occurs only for the first application.

API	First occurrence	Second occurrence
PROC_setup	Sets up GPP-side of DSPLINK	No activity. Does not result in actually allocating any resources for DSPLINK.
PROC_attach	Performs all activities required to be able to access the DSP resources from this process.	Performs all activities required to be able to access the DSP resources from this process.
POOL_open	Configures the specified pool with the specified parameters	If the same pool is opened, it is made available to the process. No change is made in the pool configuration and the parameters are ignored.
PROC_load	Loads the specified DSP executable on the DSP.	If the same DSP executable is specified, the DSP state is not changed, and the executable is not actually loaded on the DSP.
PROC_start	Starts the DSP executing from its entry point.	The DSP state is not changed, and this call does not result in actually starting the DSP execution.
MSGQ_transportOpen	Opens the MSGQ transport	The MSGQ transport is not actually opened.
MSGQ_transportClose	Does not actually close the MSGQ transport	The MSGQ transport is closed.
PROC_stop	Does not actually stop the execution of the DSP, since it is still being used by the second application.	Stops execution of the DSP and places it in reset.
POOL_close	Does not result in actually closing the pool. Only makes the pool unavailable to this process.	Closes the pool and makes it unavailable to any process/DSP.
PROC_detach	Releases all resources that were acquired for this	Releases all resources that were acquired for this process

	process in PROC_attach.	in PROC_attach.
PROC_destroy	No activity. Does not result in freeing any resources in DSPLINK.	Releases all allocated resources on the GPP-side of DSPLINK. Following this, no further calls can be made to DSPLINK APIs.

- For additional information about return codes from PROC APIs and the behavior of each API, please refer to the enhanced multi-process support design document (LNK_157_DES) available with the release.

7.2.2 Multiple processes within a single application

When multiple processes are used within a single application, the following are the salient features of this scenario:

1. There is one system integration application, which initially sets up DSPLINK. It opens all pools as are required by the system. It also loads and starts execution of the DSP executable.
2. This application forks out different processes that perform different independent activities.
3. Each process that needs to use DSPLINK attaches to the required DSP using PROC_attach ().
4. Each process also indicates the pool that it wishes to use by calling POOL_open. It may pass NULL as the pool parameters, since the pool was already opened by the system integrator.
5. Each process may also open additional pools as required only for that process.
6. The process may perform data transfers with the DSP using any DSPLINK components as required.
7. When it has completed its processing, it closes its handle to the pools it had opened by calling POOL_close.
8. It also detaches from the DSP by calling PROC_detach.
9. Only after all processes have completed their activities, the system integrator performs system shutdown by stopping the DSP execution, performing final close of the POOLs and destroying the DSPLINK driver.

7.2.3 Multiple threads within the processes

When multiple threads are used within the processes, the following are the salient features of this scenario:

1. The setup of DSPLINK is done by the processes as required. This is similar to both scenarios 1 and 2.
2. Each thread that needs to use DSPLINK can directly start using the DSPLINK component services as required. It must not call PROC_attach or POOL_open.
3. It can open and use any MSGQs, CHNLs, or other resources as required.
4. When the thread has finished processing, it can close the resources that it had allocated, and simply exit. It does not need to call POOL_close or PROC_detach.

5. On Linux, if signals are being used for cleanup processing, all threads that do not wish to catch the termination signals must mask the signals to ensure that they do not die.

8 Dos and Don't's for writing applications using DSP/BIOS LINK

8.1 Dos

8.1.1 Always check the return status of any DSPLink API call.

DSP/BIOS Link provides macros like DSP_SUCCEEDED and DSP_FAILED which can be used to determine if the API has succeeded or failed.

These API's can be used to check or return status for all protocols except RingIO. In RingIO the user needs to check each status return type explicitly as the application might need to interpret and evaluate application behavior between different success code.

This macro cannot be used in the following manner

```
if (DSP_SUCCEEDED (MSGQ_Open (msgqName, &msgq, NULL))) {
    ...
}
```

Though the argument 'x' is used only once in the statement as it appears in the program, the macro expansion can result in invoking 'x' multiple times, if it is a function.

Here, MSGQ_Open () may get invoked multiple times, resulting in undesired behavior. Hence, this usage must be replaced by the following:

```
status = MSGQ_Open (msgqName, &msgq, NULL) ;
if (DSP_SUCCEEDED (status)) {
    ...
}
```

8.1.2 Use the software dependencies with correct versions as stated in release notes.

Each DSP/BIOS Link release documents the dependencies against which it has been validated. Some of the features required by DSPLink may depend on the versions of the dependencies. The behavior of DSPLink may not be as expected if the dependencies are incorrect.

8.2 Don'ts

8.2.1 Do not use names with size equal or greater than 32 characters.

DSP/BIOS Link stores all names for e.g. RingIO names, MSGQ names in an array with size as 32 characters. If you name your MSGQ or RingIO with a size equal or larger than 32 characters width it might lead to system issues.

8.2.2 Do not call any DSP/BIOS Link API from a registered callback function

No DSP/BIOS Link API should be called from a callback function registered through the RingIO or NOTIFY module. On DSP-side or on operating systems such as PrOS, the callback functions are run from ISR context and must not perform any operations that may take a lock or block, which is done by most DSPLink APIs. Minimum functionality must be used in the callback functions, most often limited to posting a semaphore on which application is waiting, posting SWI etc.