USER GUIDE

DSP/BIOS™ LINK

LNK 058 USR

Version 1.65

Document Template Version 1.4

This page has been intentionally left blank.

This page has been intentionally left blank.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

1 Purpose

DSP/BIOS™ LINK is foundation software for the inter-processor communication across the GPP-DSP boundary. It provides a generic API that abstracts the characteristics of the physical link connecting GPP and DSP from the applications. It eliminates the need for customers to develop such link from scratch and allows them to focus more on application development[1].

This software can be used across platforms:

§ Using SoC (System on Chip) with GPP and one DSP.

§ With discrete GPP and DSP.

As the name suggests, DSP/BIOS™ is expected to be running on the DSP. No specific operating system is mandated to be running on the GPP. It is released on a reference platform for a set of reference operating systems. The release package contains full source code to enable the customers to port it to their specific platforms and/ or operating systems.

Depending on the supported platform and OS, DSP/BIOS™ LINK provides a subset of the following services to its clients:

§ Basic processor control

§ Shared/synchronized memory pool across multiple processors

§ Notification of user events

§ Mutually exclusive access to shared data structures

§ Linked list based data streaming

§ Data transfer over logical channels

§ Messaging (based on MSGQ module of DSP/BIOS)

§ Ring buffer based data streaming

The following physical mechanism may be used for messaging

§ Zero Copy Messaging

A typical application may not require all services provided by DSP/BIOS™ LINK. Also, a typical application may use only one mechanism for transferring messages between GPP and DSP. To enable this capability, DSP/BIOS™ LINK can be scaled at compile time to choose only the required components.

This document provides necessary information for users to get started with the basic concepts of DSP/BIOS™ LINK.

2 Text Conventions

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

[1] Applications differentiate the products. The application developers would prefer to focus on the application rather than the IPC mechanism.

q	This bullet indicates additional information.

3 Terms & Abbreviations

CCS	Code Composer Studio
IPC	Inter Processor Communication
DSPLink	DSP/BIOS™ LINK

4 References

1.	LNK_041_DES	Zero Copy Link Driver design document
2.	LNK_076_DES	Buffer Pools design document
3.	LNK_040_DES	DSP Executable Loader design document
4.	LNK_137_DES	Dynamic Configuration design document.
5.	LNK 133 DES	MPCS design.
6.	LNK 010 DES	Processor Manager design document.
7.	LNK 012 DES	Link Driver design document.
8.	LNK 015 DES	Test Suite design document.
9.	LNK 024 DES	OS Adaptation Layer for Linux.
10.	LNK 031 DES	Messaging Component design document.
11.	LNK 082 DES	Pool design document.
12.	LNK 096 DES	OS Adaptation Layer for PROS.
13.	LNK 128 DES	IPS & Notify
14.	LNK 129 DES	RINGIO.
15.	LNK 131 DES	MPLIST design.
16.	LNK 157 DES	Enhanced Multi-process support design
17.	LNK 182 DES	Multi-DSP Design
18.	LNK 181 DES	MMU Dynamic entry support (OMAP)

WHERE TO BEGIN?

5 Available Documents

A.1. Platform Specific

These documents are specific to the supported platform.

1.	INSTALLATION GUIDE	InstallGuide_[platform].pdf
	This document provides information to install DSP/BIOS™ LINK on the development host and setup the development platform. The platform can be OMAP2530, OMAP3530 or DM6437 etc.	
2.	OS Adaptation Layer for Linux	LNK_024_DES.pdf
	This document describes the overall design and architecture of the OS Adaptation Layer (OSAL) of DSP/BIOS™ Link for Linux.	
3.	OS Adaptation Layer for PrOS	LNK_096_DES.pdf
	This document describes the overall design and architecture of the OS Adaptation Layer (OSAL) of DSP/BIOS™ Link for PrOS.	

A.2. Generic

These documents are generic. They do not contain any information that is specific to any platform or the operating system running on the GPP.

1.	RELEASE NOTES	ReleaseNotes.pdf
	This document provides information on the current release [Version 1.65].	
2.	USER GUIDE	UserGuide.pdf
	The current document. This document provides information to get started on DSP/BIOS™ LINK.	
3.	PROGRAMMER'S GUIDE	ProgrammersGuide.pdf
	This document provides information enabling users to write applications using DSP/BIOS™ LINK.	
4.	PROCESSOR MANAGER	LNK_010_DES.pdf
	This document describes the detailed design of the Processor Manager component.	
5.	LINK DRIVER	LNK_012_DES.pdf
	This document describes the detailed design of the Link Driver component.	
6.	SHARED MEMORY PROCESSOR COPY LINK DRIVER	LNK_019_DES.pdf
	This document explains the design of link driver for data communication between the GPP and DSP for OMAP5910/5912 using shared memory.	

7.	MESSAGING USING MSGQ This document describes the detailed design of the messaging component utilizing MSGQ module of DSP/BIOS™.	LNK_031_DES.pdf
8.	ZERO COPY LINK DRIVER This document describes the detailed design of the Zero Copy Link Driver for shared memory based architectures.	LNK_041_DES.pdf
9.	RING IO This document describes the detailed design of the Ring Buffer Based data streaming component.	LNK_129_DES.pdf
10.	MPLIST This document describes the design and interface definition of linked list based transport mechanism between GPP and DSP.	LNK_131_DES.pdf
11.	MPCS This document describes the design and interface definition of the multi-processor critical section component.	LNK_133_DES.pdf
12.	BUFFER POOLS This document describes the detailed design of the different fixed-size buffer based pools provided with DSPLink.	LNK_076_DES.pdf
13	PORTING GUIDE Provides recommendations and guidelines for the developers to port DSP/BIOS™ LINK to a different GPP OS, a different platform or a different physical link.	LNK_017_PRT.pdf
14.	DSP Executable Loader Design This document describes the overall design and architecture of the Loader used to parse and load DSP binaries for DSP/BIOS™ LINK. It lists the interfaces exposed by the loader and also describes the overall design for implementation of these interfaces.	LNK_040_DES.pdf
15.	Dynamic configuration Design This document describes the overall design and architecture of the dynamic configuration used to build DSP/BIOS™ LINK.	LNK_137_DES.pdf
16	Enhanced multiprocess support This module provides the design for enhanced multi-process support within DSPLink. This allows multiple applications/processes to use DSPLink independently, and without being aware of each other.	LNK_157_DES.pdf
17	Multi DSP Design This document describes the overall design of MULTI-DSP DSPLink.	LNK_182_DES.pdf
18.	MMU Dynamic entry support (OMAP) This document describes the overall design and architecture of the MMU dynamic entry creation.	LNK_181_DES.pdf

SOFTWARE ARCHITECTURE

6 Overview

The software architecture of DSP/BIOS™ LINK is shown in the diagram below:

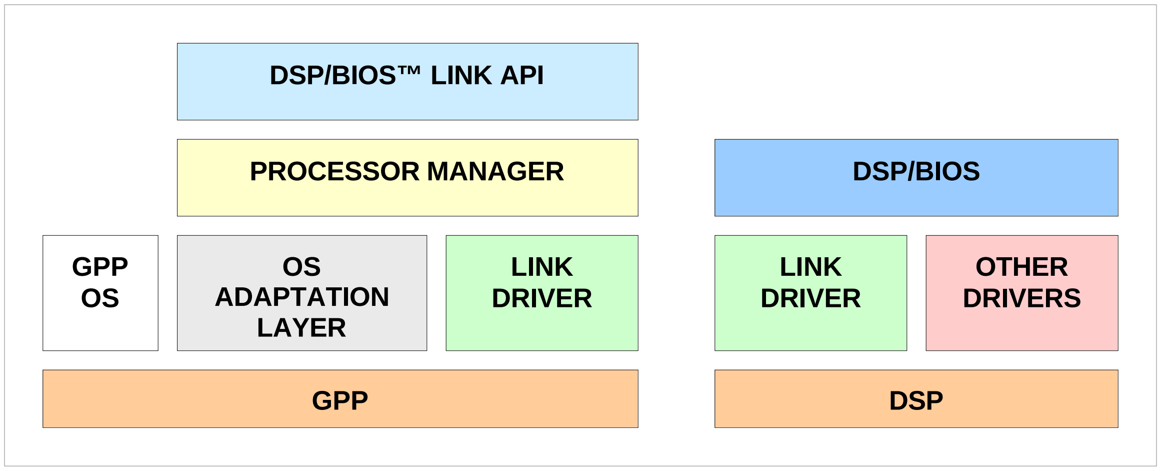


Figure 1. Software architecture of DSP/BIOS™ LINK

6.1 On the GPP side

On the GPP side, a specific OS is assumed to be running.

The OS ADAPTATION LAYER encapsulates the generic OS services that are required by the other components of DSP/BIOS™ LINK. This component exports a generic API that insulates the other components from the specifics of an OS. All other components use this API instead of direct OS calls. This makes DSP/BIOS™ LINK portable across different operating systems.

The LINK DRIVER encapsulates the low-level control operations on the physical link between the GPP and DSP. This module is responsible for controlling the execution of the DSP and data transfer using defined protocol across the GPP-DSP boundary.

The PROCESSOR MANAGER maintains book-keeping information for all components. It also allows different boot-loaders to be plugged into the system. It builds exposes the control operations provided by the LINK DRIVER to the user through the API layer.

The DSP/BIOS™ LINK API is interface for all clients on the GPP side. This is a very thin component and usually doesn't do any more processing than parameter validation. The API layer can be considered as 'skin' on the 'muscle' mass contained in the PROCESSOR MANAGER and LINK DRIVER.

The thin API layer allows easy partition of DSP/BIOS™LINK across the user kernel boundary on specific operating systems e.g. Linux. Such partition may not be necessary on other operating systems.

6.2 On the DSP side

Here, the LINK DRIVER is one of the drivers in DSP/BIOS™. This driver specializes in communicating with the GPP over the physical link.

The communication (data/ message transfer) is done using the DSP/BIOS™ modules- SIO/ GIO/ MSGQ. There are specific DSP/BIOS™ LINK API on the DSP for the other modules RingIO, MPCS, MPLIST, NOTIFY, POOL.

7 Key Components

7.1 PROC

This component represents the DSP processor in the application space. PROC is an acronym for 'processor'.

This component provides services to:

§ Initialize the DSP & make it available for access from the GPP.

§ Load code on the DSP.

§ Start execution from the run address specified in the executable.

§ Read from or write to DSP memory.

§ Stop execution.

§ Additional platform-specific control actions.

§ Get DSP address for given symbol.

In the current version, only one processor is supported. However, the APIs are designed to support multiple DSPs and hence they accept a `processorId` argument to support this future enhancement.

7.2 POOL

This component provides APIs for configuring shared memory regions across processors. It also provides APIs for synchronizing the contents of buffer as seen by the two CPU cores.

These buffers are used by other modules from DSP/BIOS Link for providing inter-processor communication functionality. These can also be used by applications for implementing their own protocol for data streaming if desired.

The specific services provided by this module are:

1. Configure the shared memory region through open & close calls.
2. Allocate and free buffers from the shared memory region.
3. Translate address of a buffer allocated to different address spaces (e.g. GPP to DSP)
4. 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™.

7.3 NOTIFY

This component allows applications to register for notification of events occurring on the remote processor and send event notification to the remote processor.

It allows applications to register a callback function with an associated parameter for events that occur on remote processors.

It enables applications to send specific event notification to remote processors. The applications can also send an optional value with the event.

The NOTIFY component enforces a priority on event notifications. The priority is defined by the event number, with lower event number indicating higher priority.

The applications can also un-register their event callback functions at runtime if such event notification is no longer required.

7.4 MPCS

This component allows applications to achieve mutually exclusive access to shared data structures through a multi-processor critical section (MPCS) between GPP and DSP.

Applications may need to define their own data structures in memory that can be accessed by multiple processors. Such data structures can be used for communicating pieces of information between the processors. However, applications need to ensure mutually exclusive access to such data structures between multiple processors, and multiple tasks on each processor, to ensure consistency of data. To enable such scenarios, the MPCS component is provided to support this functionality.

In a multiprocessor system having shared access to a memory region, a multi-processor critical section between GPP and DSP can be implemented. In cases where a shared memory region does not exist, the module internally performs the synchronization required to provide the protection required by the MPCS component.

The MPCS component provides APIs to create and delete instances of the MPCS. Each instance of the MPCS is identified by a system-wide unique string name. Every client that needs to use an MPCS must get a handle to the MPCS by calling an API to open it. A corresponding API to close the MPCS handle is used when the client no longer needs to use the MPCS.

APIs to enter and leave the critical section specified by the MPCS object handle are also provided.

If provided by the user, the memory required for the MPCS object must be allocated from a pool accessible across the processors. Alternatively, if no memory is provided during creation of the object, the pool ID specified is used to internally allocate the MPCS object.

7.5 MPLIST

This component provides a doubly-linked circular linked list based transport mechanism between GPP and DSP.

On the devices where a shared memory region exists between GPP and DSP, this module implements the linked-list in the shared memory region. In cases where a shared memory region does not exist, the module internally maintains coherence between linked lists on the remote processors.

This component provides APIs to create and delete instances of the MPLIST. Each instance of the MPLIST is identified by a system-wide unique string name. Every client that needs to use an MPLIST must get a handle to the MPLIST by calling an API to open it. A corresponding API to close the MPLIST handle is used when the client no longer needs to use the MPLIST.

The MPLIST component provides APIs to place an element at the end of list, and remove an element from the front of the list. It also allows applications to insert a buffer before an existing element in the list, and remove any specified list element from the list. An API to check if the list is empty is also provided. In addition, APIs are also provided to traverse the list by getting a pointer to the first element in the list and the element after a specified element.

7.6 CHNL

This component represents a logical data transfer channel in the application space. CHNL is responsible for the data transfer across the GPP and DSP. CHNL is an acronym for 'channel'.

A channel (when referred in context of DSP/BIOS™ LINK) is:

§ A means of transferring data across GPP and DSP.

§ A logical entity mapped over a physical connectivity between the GPP and DSP.

§ Uniquely identified by a number within the range of channels for a specific physical link towards a DSP.

§ Unidirectional. The direction of a channel is decided at run time based on the attributes passed to the corresponding API.

Multiple channels may be multiplexed on single physical link between the GPP and DSP depending upon the characteristics of the link & associated link driver.

The data being transferred on the channel does not contain any information about the source or destination[2]. The consumer and producer on either side of the processor boundary must establish the data path explicitly.

This component follows the issue-reclaim model for data transfer. As such, it mimics the behavior of issue-reclaim model of the SIO module in DSP/BIOS™. This model is briefly summarized in the appendix of this document.

7.7 MSGQ

This component represents queue based messaging. It is an acronym for 'message queue'.

This component is responsible for exchanging short messages of variable length between the GPP and DSP clients[3]. It is based on the MSGQ module in DSP/BIOS™.

The messages are sent and received through message queues.

A *reader* gets the message from the queue and a *writer* puts the message on a queue. A message queue can have only one reader and many writers. A task may read from and write to multiple message queues.

The client is responsible for creating the message queue if it expects to receive messages. Before sending the message, it must 'locate' the queue where message is destined.

7.8 RING IO

This component provides Ring Buffer based data streaming.

This component allows creation of a ring buffer created within the shared memory. The reader and writer of the ring buffer can be on different processors.

[2] The contents of data buffer are not interpreted during the data transfer operations.
[3] The unit of execution on the GPP depends upon the GPP OS.

The RingIO component provides the ability for the writer to 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.

The RingIO component provides the ability for the reader to 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.

Each RingIO instance can have a single reader and a single writer.

The RingIO component also supports APIs for enabling synchronous transfer of attributes 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.

8 Source Code Layout

The top-level source code layout is shown in the diagram below:

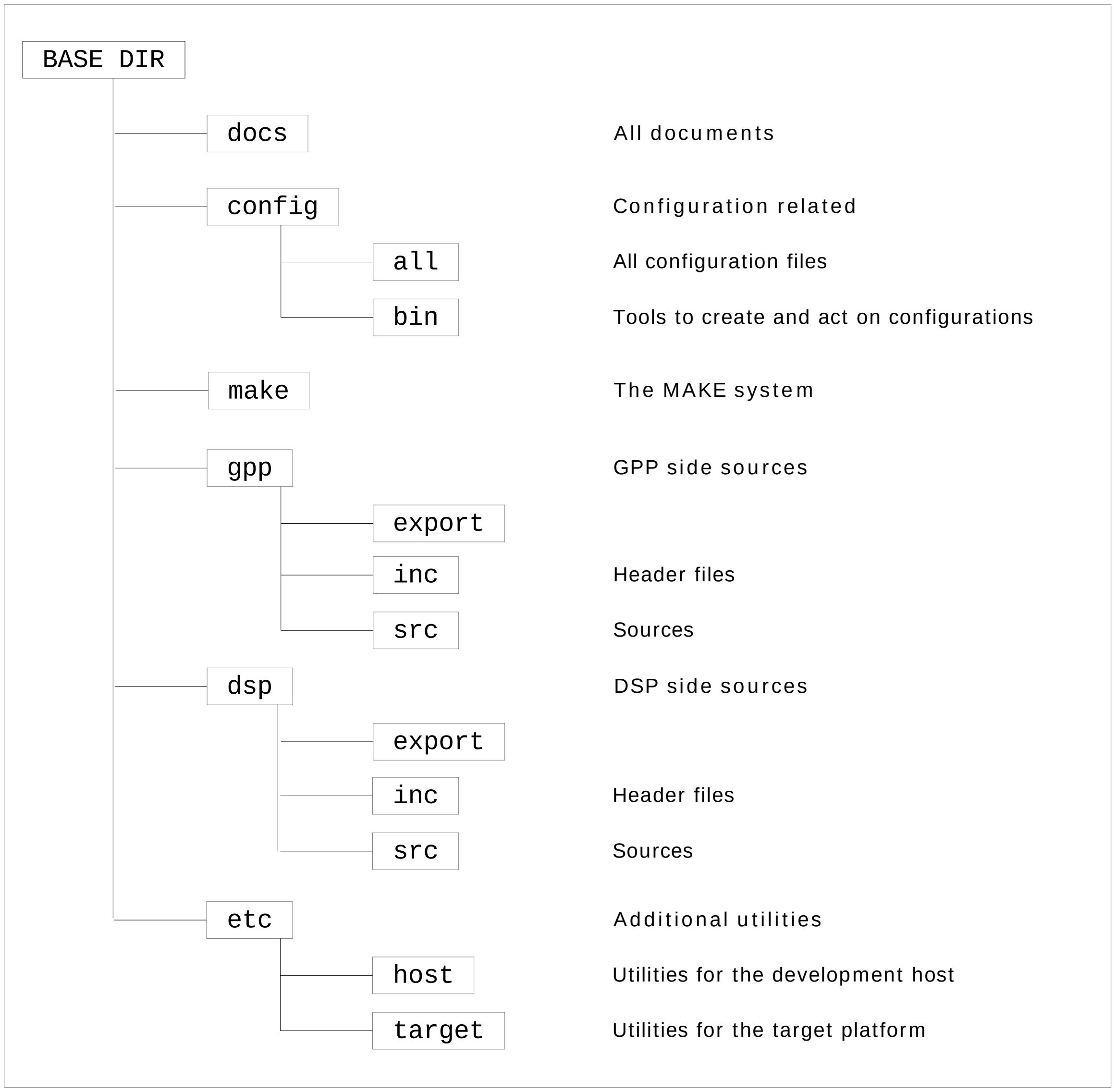
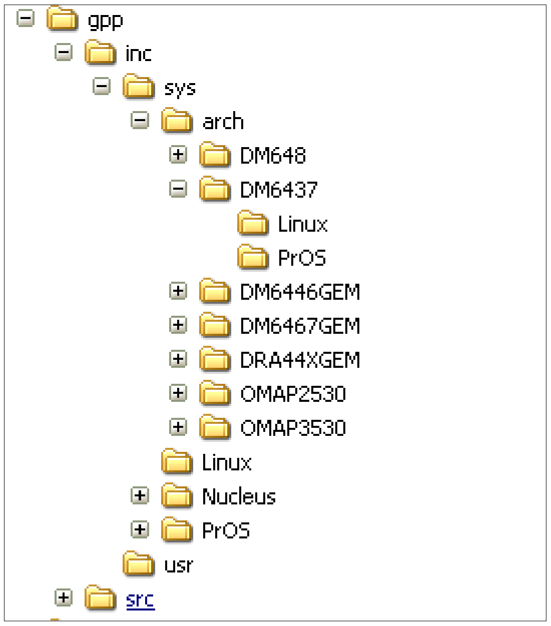


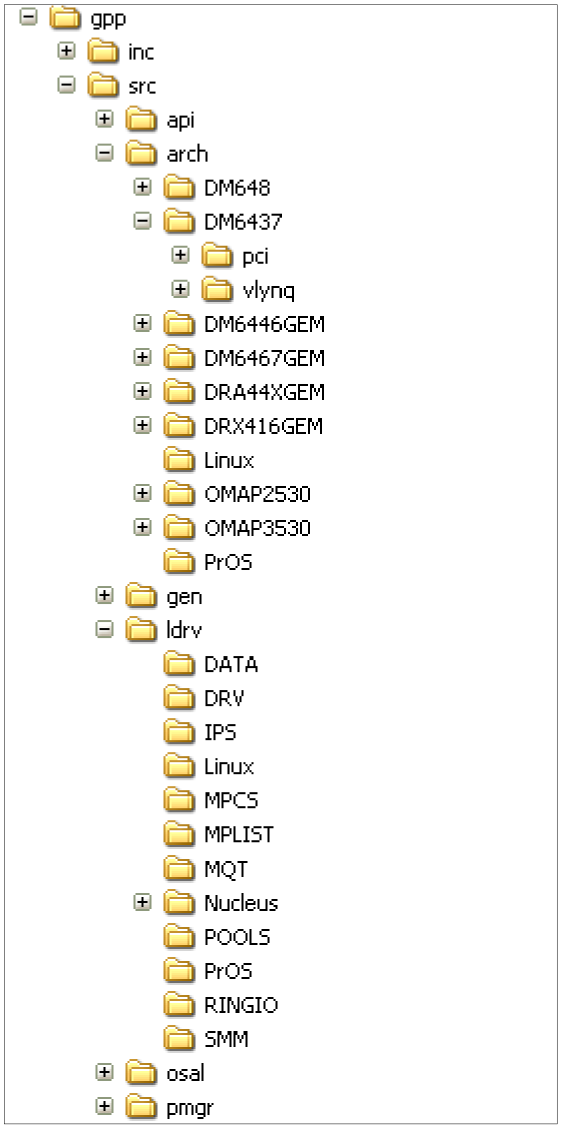
Figure 2. Top level view of directory structure

8.1 GPP side sources

The directory structure for the sources in the GPP side is shown below:



```
⊟ gpp
    ⊟ inc
        ⊟ sys
            ⊟ arch
                ⊞ DM648
                ⊟ DM6437
                    Linux
                    PrOS
                ⊞ DM6446GEM
                ⊞ DM6467GEM
                ⊞ DRA44XGEM
                ⊞ OMAP2530
                ⊞ OMAP3530
                Linux
            ⊞ Nucleus
            ⊞ PrOS
            usr
    ⊞ src
```



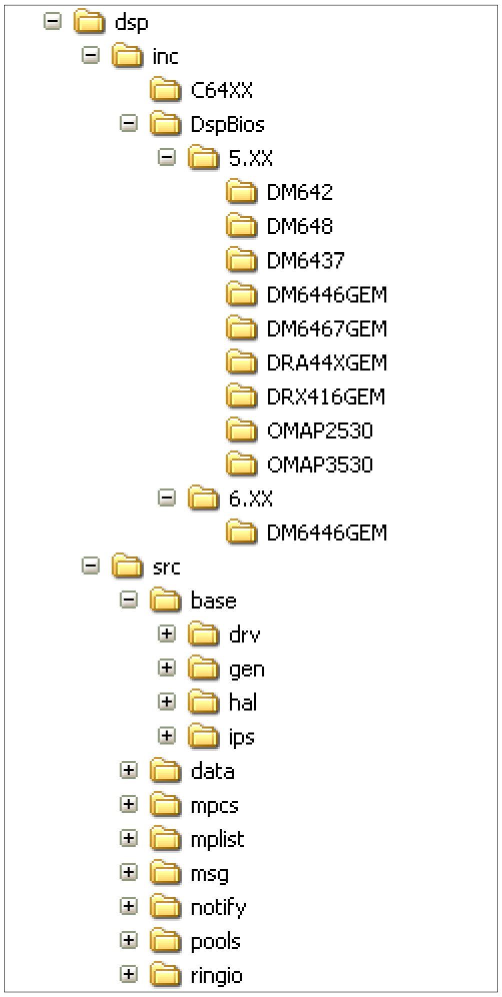
```
⊟ gpp
    ⊞ inc
    ⊟ src
        ⊞ api
        ⊟ arch
            ⊞ DM648
            ⊟ DM6437
                ⊞ pci
                ⊞ vlynq
            ⊞ DM6446GEM
            ⊞ DM6467GEM
            ⊞ DRA44XGEM
            ⊞ DRX416GEM
              Linux
            ⊞ OMAP2530
            ⊞ OMAP3530
              PrOS
        ⊞ gen
        ⊟ ldrv
              DATA
              DRV
              IPS
              Linux
              MPCS
              MPLIST
              MQT
            ⊞ Nucleus
              POOLS
              PrOS
              RINGIO
              SMM
        ⊞ osal
        ⊞ pmgr
```

Figure 3. Directory structure for GPP side sources

- All directories in "LDRV" may have sub-directories for platform-specific files. This is not shown explicitly in above diagram.

8.2 DSP side sources

The directory structure for the sources in the DSP side is shown below:



```
⊟ dsp
    ⊟ inc
        C64XX
        ⊟ DspBios
            ⊟ 5.XX
                DM642
                DM648
                DM6437
                DM6446GEM
                DM6467GEM
                DRA44XGEM
                DRX416GEM
                OMAP2530
                OMAP3530
            ⊟ 6.XX
                DM6446GEM
    ⊟ src
        ⊟ base
            ⊞ drv
            ⊞ gen
            ⊞ hal
            ⊞ ips
        ⊞ data
        ⊞ mpcs
        ⊞ mplist
        ⊞ msg
        ⊞ notify
        ⊞ pools
        ⊞ ringio
```

Figure 4. Directory structure for DSP side sources

q When compiled through the DSP/BIOS™ Link build system the subdirectories in “src” tree generates up to eight libraries. The “base” directory contains the sources for generating the base dsplink.lib that is needed for both data transfer and messaging. This part implements the basic driver functionality. The “data” directory contains the sources for generating the data transfer-specific library dsplinkdata.lib. The “msg” directory contains the sources for generating the messaging-specific library dsplinkmsg.lib. The “ringio” directory contains the sources for generating the ringio-specific library dsplinkringio.lib. The “mplist” directory contains the sources for generating the mplist-specific library dsplinkmplist.lib. The “mpcs” directory contains the sources for generating the mpcs-specific library dsplinkmpcs.lib. The “pool” directory contains the sources for generating the pool-specific library dsplinkpool.lib. The “notify” directory contains the sources for generating the notify -specific library dsplinknotify.lib.

- All directories in “base”, “data”, “msg”, “ringio” “mpcs”, “notify”, “pools” and “mplist” may have sub-directories for platform-specific files. This is not shown explicitly in above diagram.

8.3 Make-system Organization

The directory structure for the common make-system for building GPP as well as DSP sources is shown below.

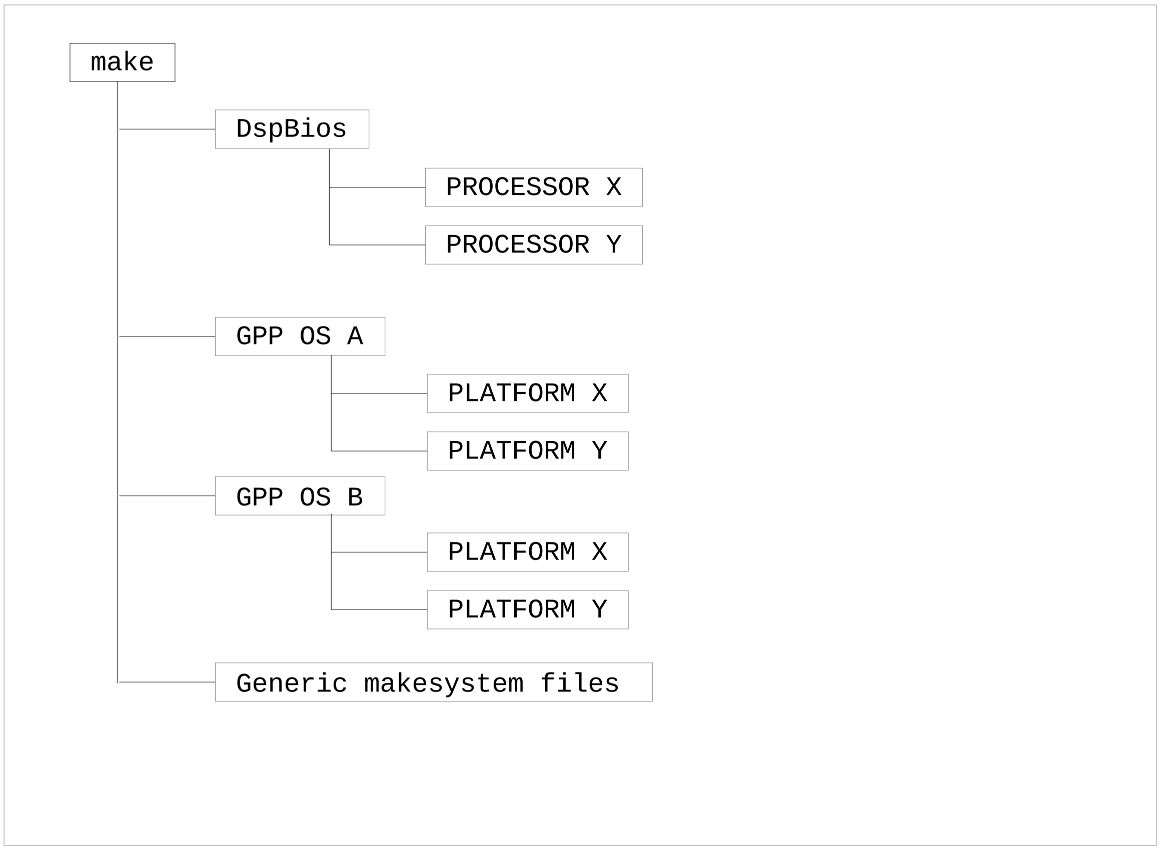


Figure 5. Directory structure for the make-system

BUILD PROCEDURE

Two types of development hosts can be used for building the DSPLink sources. If the target GPP OS is Linux, the GPP-side sources must be built on a Linux development host.

When working with PrOS, a windows development host must be used for building the GPP-side sources.

For building the DSP-side sources, either a Linux or Windows development host may be used.

9 Customizing and configuring the build environment

Before building the DSPLink sources, the DSPLink build system needs to be customized and configured for the user's build environment and target application.

This involves the following three major activities:

1. DSPLink make system customization
2. Setup the build environment
3. DSPLink build configuration

9.1 DSPLink make system customization

The DSPLink make system provided with the release assumes the build environment setup as documented in the Installation Guide. To customize the make system to the user's build environment, some files may need to be modified.

9.1.1 Operating System distribution file

DSPLink supports build for the Linux and Windows development hosts. The different platforms may support different variants or versions of the target operating system. In addition, the tool-chain used for building the sources may differ based on the selected platform.

To support this, the make system provides a separate distribution file for the Operating System distribution being used. Typically, there is one distribution file for each platform-OS combination. For more details on distribution files, please refer to the section 32.2.5 on "Supporting a new distribution".

The distribution file for a specific platform-OS combination can be found within the DSPLink installation at:

```
$(DSPLINK)/make/<$(GPPOS) | $(DSPOS)>
```

For example, the distribution file for the Davinci platform for Linux is:

```
$(DSPLINK)/make/Linux/davinci_mvlpro5.0.mk
```

The distribution file for the DSP-side for the Davinci platform for Linux is:

```
$(DSPLINK)/make/DspBios/c64xxp_5.xx_linux.mk
```

The configuration values within this distribution file can be modified to customize the make system for the user build environment. Some of the common values that may need to be modified are:

For GPP-side distribution file:

`BASE_BUILDOS`	Base directory for the GPP operating system

For DSP-side distribution file:

`BASE_INSTALL`	Base directory for the installed tools and operating system.
`BASE_SABIOS`	Base directory for the DSP operating system
`BASE_CSL`	Base directory for the Chip Support Library (if required for the platform)

For both GPP and DSP-side distribution file:

`BASE_CGTOOLS`	Base directory for the code generation tool chain
`STD_CC_FLAGS`	Standard build flags for the compiler
`STD_AR_FLAGS`	Standard build flags for the archiver
`STD_LD_FLAGS`	Standard build flags for the linker

In addition, there may be some configuration values within the distribution file that are specific to the selected platform-OS combination.

9.1.2 System Tools configuration file

For each target operating system, based on whether the build environment is Linux or Windows-based, the DSPLink make system configures the system tools and system calls through a specific file `systools.mk` present in:

```
$(DSPLINK)/make/<$(GPPOS) | $(DSPOS)>
```

This file may also need to be customized for the user build environment.

Typically, the following configuration values may need to be modified:

`BASE_PERL`	Base directory for the PERL installation

9.2 Setup the build environment

Scripts are provided to setup the necessary environment variables required by DSPLink:

DSPLINK	Defines the root for DSP/BIOS LINK installation.
PATH	Appends the path to include scripts provided in the installation.

9.2.1 Linux development host

1. Set up necessary environment variables.

```
$ source ~/dsplink/etc/host/scripts/Linux/dsplinkenv
```

2. The modified environment variables are displayed.

```
===========================================================
The environment for DSP/BIOS LINK development has been set:
DSPLINK  = /home/<user>/dsplink
PATH    += /home/<user>/dsplink/etc/host/scripts/Linux
===========================================================
```

- The above command assumes that you have installed DSP/BIOS™ Link in your home directory on the development host. If this is not the case, the script must be updated to reflect the location where the product is installed.
- The above command assumes that you are using *tcsh* shell. If you are using *bash* shell, an equivalent script '*dsplinkenv.bash*' is shipped with the release package that can be used.
- This command can be included in the '.rc' file corresponding to your shell in which case you will no longer need to execute the 'dsplinkenv' script.

9.2.2 Windows development host

1. Set up necessary environment variables.

```
L:> dsplink\etc\host\scripts\msdos\dsplinkenv.bat
```

2. The modified environment variables are displayed.

```
==========================================================
The environment for DSP/BIOS LINK development has been set:
DSPLINK  = L:\dsplink
PATH    += L:\dsplink\etc\host\scripts\msdos
==========================================================
```

- In general, it is advisable to use a Linux host for Linux-based development. However, if a Windows development host is being used for this, the following points need to be considered:
  - On a Windows development host, if using PC-based Linux development environment such as mvcyg4.0, only GPP-side of DSPLink can be built using this shell.
  - For building DSP-side, an MSDOS shell should be used. On this shell, it must be ensured that cygwin is not in the path, else errors will be seen during build.

9.3 DSPLink build configuration

The build configuration script must be executed to configure DSPLink for the various parameters such as platform, GPP OS, build configuration etc.

The build configuration for DSP/BIOS™ LINK is an interactive process. The generated configuration file is appropriately included during the build process. The build configuration depends upon the environment variable – DSPLink set by execution of the script mentioned earlier.

9.3.1 Linux development host

The build configuration can be initiated by executing the command: `dsplinkcfg`

1. Execute the build configuration perl script.

```
$ perl ~/dsplink/config/bin/dsplinkcfg.pl
```

2. This script expects command line options to configure the DSPLink. These command line options are described in this section.

9.3.2 Windows development host

The build configuration can be initiated by executing the command: `dsplinkcfg.bat`

1. Execute the build configuration script.

   ```
   L:> perl dsplink\config\bin\dsplinkcfg.pl
   ```

2. This script expects command line options to configure the DSPLink. These command line options are described in this section.

9.3.3 Command Line Options

9.3.3.1 Platform

This option directs the configure script to configure DSPLink for the provided platform.

Usage	--platform=<PLATFORM ID>
Example	--platform=DAVINCI, DSPLink is configure for Davinci Platform.

O If no option is provided, configure script displays help message with listing all supported platforms:

```
***************** ERROR !!! ***************************

Please provide a valid Platform!
Following platform are supported currently:

ID-->DAVINCI
        DaVinci SoC - C64P DSP interfaced directly to ARM9
ID-->DAVINCIHD
        DaVinciHD SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->JACINTO1
        Jacinto SoC version 2 - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->JACINTO2
        Jacinto SoC Version2 - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->LINUXPC
        Linux Box (PC) with PCI based cards
        This platform supports multi DSP (PCI cards) architecture
ID-->OMAP2530
        Omap2530 SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->OMAP3530
        Omap3530 SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
```

```
ID-->DA8XX
        DA8XX SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
ID-->OMAPL1XX
        OMAP-L1XX SoC - C64P DSP interfaced directly to ARM9
        This platform does not supports multi DSP scenario
Provided:
Example: --platform=DAVINCI or --platform=<ID>
```

On successful condition it will display messages as below:

```
==========================================================
Chosen configuration is as follows:

Chosen platform:
        Identifier:     DAVINCI
        Description:    DaVinci SoC - C64P DSP interfaced directly to
ARM9
```

9.3.3.2 *Number of DSPs*

This option directs the configure script to configure DSPLink for the desired number of DSPs.

Usage	--nodsp=<Number of DSPs>
Example	--nodsp=1.

O If no option is provided, configure script displays help message as below:

```
****************** ERROR !!! ***************************
Please provide a valid number of DSPs!
Please provide number of DSPs in the system
Provided:
Example: --nodsp=2
```

On successful condition, it will display messages as follows:

```
        No of DSPs:     1
```

9.3.3.3 *DSPPhysical Interface.*

This option directs the configure script to configure the DSPLINK for the desired DSP and also makes the chosen DSP uses the desired physical interface. For example user can choose DM6437 VLYNQ interface with Jacinto1 system, where DM6437 can support two different physical interfaces, PCI and VLYNQ.

Usage	--dspcfg_0==<DSPID><PHY ID>

Example	--dspcfg_0=DM6446GEMSHMEM. Here _0 denotes phy for first DSP.

O If no option is provided, configure script displays help message as below:

```
***************** ERROR !!! **************************
Please provide a valid DSP for DSP0 with a valid Physical Interface
combination!
Following DSP & Physical interface (PHY) combinations are supported
byDM6446GEM:
<ID>-->DM6446GEMSHMEM
        Shared Memory Physical Interface
Provided:
Example: --dspcfg_0==<ID> or --dspcfg_0==DM6446GEMSHMEM
```

On successful condition, it will display messages as follows:

```
Chosen combination for DSP0:
          Identifier             :     DM6446GEM
          DSP Description        :     On-Chip DSP of DaVinci SoC
          Physical Interface (PHY):     DM6446GEMSHMEM
          PHY Description        :     Shared Memory Physical Interface
```

9.3.3.4 DSP OS.

This option directs the configure script that the chosen DSP uses the desired DspBios interface. For example user can choose DspBios5 or DspBios6 on DaVinci Platform.

Usage	--dspos_0=<DSP OS ID>
Example	--dspos_0=DSPBIOS5XX. Here _0 denotes DSP OS for first DSP.

O If no option is provided, configure script displays help message as below:

```
***************** ERROR !!! **************************
Please provide a valid DSP OS!
Following DSP OS are supported by DM6446GEM with Shared Memory Physical
Interface:
<ID>-->DSPBIOS5XX
        DSP/BIOS (TM) Version 5.XX
<ID>-->DSPBIOS6XX
        DSP/BIOS (TM) Version 6.XX
Provided:
Example: --dspos_0=<ID> or --dspos_0=DSPBIOS5XX
```

On successful condition, it will display messages as follows:

```
Chosen DSP OS for DSP0:
          Identifier:   DSPBIOS5XX
          Description:  DSP/BIOS (TM) Version 5.XX
```

9.3.3.5 GPP OS.

This option directs the configure script to configure DSPLink for desired GPP OS.

Usage	--gppos=<GPP OS ID>
Example	--gppos=MVL4G. Configures to use, Montavista Pro 4.0 Linux with GLibc system.

O If no option is provided, configure script displays help message as below:

```
***************** ERROR !!! **************************
Please provide a valid GPP OS!
Following GPP OS are supported by selected DSPs:
<ID>-->MVL5U
        Montavista Pro 5.0 Linux + uCLibc Filesystem
<ID>-->MVL5G
        Montavista Pro 5.0 Linux + gLibc Filesystem
<ID>-->WINCE
WinCE OS 6.0
Provided:
Example: --gppos=<ID> or --gppos=MVL5G
```

On successful condition, it will display messages as follows:

```
Chosen GPP OS for DSP(s):
        Identifier:     MVL5G
        Description:    Montavista Pro 5.0 Linux + gLibc Filesystem
```

9.3.3.6 DSPLink Components

This option directs the configure script to configure DSPLink for the desired components, this option provides scalability.

Usage	--comps=<component string>
Example	--comps=ponslrmc. Configures DSPLink with all components.

O If no option is provided, configure script displays help message as below:

```
***************** ERROR !!! **************************
Please provide valid components!
Following COMPONENTs  are supported by MVL5G:
        [P]ROC Component
        P[O]OL Component
        [N]OTIFY Component
        MPC[S] Component
        MP[L]IST Component
        [R]INGIO Component
        [M]SGQ Component
        [C]HNL Component
```

```
Provided:
Example: --comps=ponslrmc
```

On successful condition, it will display messages as follows:

```
Chosen Components for DSPLink:
        USE_PROC        = 1
        USE_NOTIFY      = 1
        USE_POOL        = 1
        USE_MPCS        = 1
        USE_MPLIST      = 1
        USE_RINGIO      = 1
        USE_MSGQ        = 1
        USE_CHNL        = 1
```

9.3.3.7 *Filesystem*

This option directs the configure script to configure DSPLink for desired filesystem. User must provide this option only on platform which supports multiple filesystems, for example, Jacinto support PSEUDO or PrFile.

Usage	--fs=<FILESYSTEM ID>
Example	--fs=PSEUDOFS.

O If no option is provided, configure script displays help message as below:

```
****************** ERROR !!! ***************************
Please provide a valid filesystem!
Following filesystems are supported by GPP OS (PROS):
<ID> --> PSEUDOFS
        Read user guide to compile a Pseudo filesytem
        and how to build it with dsplink
<ID> --> PRFILEFS
        Read PrFile guide for further details
Provided:
Example: --fs=PSEUDOFS or --fs=<ID>
```

On successful condition, it will display messages as follows:

```
Chosen Filesystem for GPP OS:
        Identifier:     PRFILEFS
        Description:    Read PrFile guide for further details
```

9.3.3.8 *Legacy Support*

This option directs the configure script to enable legacy support, for multi-DSP DSPLink modules were upgrade, application written for older version of DSPLink will break if legacy support is not enable. This is optional options, default is no legacy support.

Usage	--legacy=1
Example	--legacy=1

O If no option is provided, configure script displays help message as below:

```
****************** ADVICE !!! ***************************
To enable legacy support use option: --legacy=1
```

9.3.3.9 *Trace*

This option directs the configure script to enable tracing. This is an optional argument, if not provided assumes disable state.

Usage	--trace=1|0
Example	--trace=1. Enables the trace.

O If no option is provided, configure script displays help message as below:

```
****************** ADVICE !!! ***************************
To enable trace use option: --trace=1
```

On successful condition, it will display messages as follows:

```
Trace : 1
```

9.3.3.10 *DSP SWI-TSK mode configuration*

This option directs the configure script to enable DSP SWI mode or DSP TSK mode. This is an optional argument, if not provided assumes DSP SWI mode enabled.

Usage	--DspTskMode=1
Example	--DspTskMode=1. Enables the DSP TSK mode.

O If no option is provided, configure script displays help message as below:

```
****************** ADVICE !!! ***************************
To enable DSP TSK mode select: --DspTskMode=1
Provided:
Assuming DSP SWI mode enabled and continuing...
```

On successful condition, it will display messages as follows:

```
Enabling DSP TSK Mode !!
```

9.3.3.11 *dspdma*

This option directs the configure script to enable DMA instead of memory copy for data transfer between host and DSP. This option is valid for pci platforms (LINUXPC) only. This is an optional argument, if not provided assumes memory copy by default.

Usage	--dspdma=1

Example	--dspdma=1. Enables the DMA.

O If no option is provided, configure script displays help message as below:

```
***************** ADVICE !!! ***************************
To enable usage of dsp edma use option: --dspdma=1
```

On successful condition, it will display messages as follows:

```
Enabling option to use DSP EDMA instead of default memcpy!!
```

9.3.3.12 *GPP Temporary & Export Path*

Using this option all GPP temporaries/Libraries and Binaries are generated at desired path.

Usage	--gpp_temp=<path>
Example	--gpp_temp=/home/skull/dsplink_temp/gpp.

O If no option is provided, configure script displays help message as below:

```
***************** ADVICE !!! ***************************
Binaries for GPP can be generated at preferred location
For example: --gpp_temp=/home/dsplink/gpp/bin
```

9.3.3.13 *DSP Temporary & Export Path*

Using this option all DSP temporaries/Libraries and Binaries are generated at desired path.

Usage	--dsp0_temp=<path>
Example	--dsp0_temp=/home/skull/dsplink_temp/dsp_0. Here note prefix '0' tells that path for first DSP.

O If no option is provided, configure script displays help message as below:

```
***************** ADVICE !!! ***************************
Binaries for DSP can be generated at preferred location
For example: --dsp0_temp=/home/dsplink/dsp<#>/bin
```

9.4 Additional steps for XDCtools-based configuration users

If users are integrating DSPLink into their systems using XDCtools-based configuration (e.g. using Codec Engine), there are 2 more steps required before the Build Configuration step is complete.

```
cd into the $(DSPLINK)/dsp directory and run:
$ $(XDC_INSTALL_DIR)/xdc clean
$ $(XDC_INSTALL_DIR)/xdc .interfaces
```

```
cd into the $(DSPLINK)/gpp directory and run:
$ $(XDC_INSTALL_DIR)/xdc clean
$ $(XDC_INSTALL_DIR)/xdc .interfaces
```

These two steps prepare the dsplink.dsp and dsplink.gpp XDC packages for consumption by the XDC config tooling.

10 Build the sources

The GPP and DSP-side DSPLink sources, sample applications and test-suite can be built using the common make system provided with the DSPLink release. The make system supports building the sources on Linux or Windows development host.

Please refer to the section 32 on “Understanding The MAKE System” for additional generic details about the make system.

10.1 Linux development host

`make` is used for building the sources on a Linux development host.

It can be invoked from the shell within the base directory of the sources to be built:

```
make –s [TARGET] [VERBOSE=1]
```

- The '-s' option can be used to build silently. Please refer to make documentation for other options.

The TARGET can be one of the following:

`all`	Make all build variants. [Default]
`debug`	Build DEBUG variant.
`release`	Build RELEASE variant.
`clean`	Delete all intermediate and output files.
`clobber`	Delete all directories created during build process.
`targets`	Build the target (.o/.ko) file from the intermediate object files.
`exports`	Export the specified file to a pre-defined location.

10.1.1 GPP-side build

Build the sources

1. Change to the source directory:

```
$ cd ~/dsplink/gpp/src
```

2. Start the build process:

```
$ make –s [debug | release]
```

3. Upon successful completion of build, the kernel module and user library shall be created in the following directories:

```
$(DSPLINK)/gpp/export/BIN/<GPP OS>/<PLATFORM>/DEBUG
$(DSPLINK)/gpp/export/BIN/<GPP OS>/<PLATFORM>/RELEASE
Or
$(GPPTEMPATH)/gpp/export/BIN/<GPP OS>/<PLATFORM>/DEBUG
$(GPPTEMPATH)/gpp/export/BIN/<GPP OS>/<PLATFORM>/RELEASE
```

Build the samples

1. Change to the source directory:

```
$ cd ~/dsplink/gpp/src/samples
```

2. Start the build process. Based on the selected build configuration, some or all of the samples may be built.

```
$ make –s [debug | release]
```

3. Upon successful completion of build, the sample application executables shall be created in the following directories:

```
$(DSPLINK)/gpp/export/BIN/<GPP OS>/<PLATFORM>/DEBUG
$(DSPLINK)/gpp/export/BIN/<GPP OS>/<PLATFORM>/RELEASE
Or
$(GPPTEMPATH)/gpp/export/BIN/<GPP OS>/<PLATFORM>/DEBUG
$(GPPTEMPATH)/gpp/export/BIN/<GPP OS>/<PLATFORM>/RELEASE
```

q To build a single sample application, change to the source directory of the specific sample in the first step above..

10.1.2 DSP-side build

Build the sources

1. Change to the source directory:

```
$ cd ~/dsplink/dsp/src
```

2. Start the build process:

```
$ make –s [debug | release]
```

3. Upon successful completion of build, the DSP libraries shall be created in the following directories:

```
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP PROCID>\DEBUG
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP PROCID>\RELEASE
Or
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP PROCID>\DEBUG
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP PROCID>\RELEASE
```

q Based on the scalability configuration selected, the DSP/BIOS™ LINK libraries generated are:

q dsplink.lib: Generic DSPLink base library required for using all components supported within DSPLink.

q dsplinkdata.lib: DSPLink library required for data transfer.

q dsplinkmsg.lib: DSPLink library required for messaging.

q dsplinkringio.lib: DSPLink library required for RingIO.

q dsplinkmplist.lib: DSPLink library required for MPLIST.

q dsplinkmpcs.lib: DSPLink library required for MPCS

- dsplinknotify.lib: DSPLink library required for NOTIFY.
- dsplinkpool.lib: DSPLink library required for POOL.

Build the samples

1. Change to the source directory:

```
$ cd ~/dsplink/dsp/src/samples
```

2. Start the build process. Based on the selected build configuration, some or all of the samples may be built.

```
$ make –s [debug | release]
```

3. Upon successful completion of build, the sample executable shall be created in the following directories:

```
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\DEBUG
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\RELEASE
Or
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
PROCID>\DEBUG
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
PROCID>\RELEASE
```

- To build a single sample application, change to the source directory of the specific sample in the first step above.

10.2 Windows development host

`gmake` is used for building the sources on a Windows development host.

It can be invoked from the shell within the base directory of the sources to be built:

```
gmake –s [TARGET] [VERBOSE=1]
```

- The '-s' option can be used to build silently. Please refer to gmake documentation for other options.
- Please refer to the Install Guide for the specific platform for details on how to build using Platform builder for Wince GPP OS. This is the recommended way to build the DSPLink for WinCE.

The TARGET can be one of the following:

`all`	Make all build variants. [Default]
`debug`	Build DEBUG variant.
`release`	Build RELEASE variant.
`clean`	Delete all intermediate and output files.

`clobber`	Delete all directories created during build process.
`targets`	Build the target (.o/.ko) file from the intermediate object files.
`exports`	Export the specified file to a pre-defined location.

10.2.1 GPP-side build

Build the sources

1. Change to the source directory:

```
L:> cd dsplink\gpp\src
```

2. Start the build process:

```
L:\dsplink\gpp\src> gmake –s [debug | release]
```

3. Upon successful completion of build, the user library shall be created in the following directories:

```
$(DSPLINK)\dsplink\gpp\export\BIN\<GPP OS>\<PLATFORM>\DEBUG
$(DSPLINK)\gpp\export\BIN\<GPP OS>\<PLATFORM>\RELEASE
Or
$(GPPTEMP PATH)\gpp\export\BIN\<GPP OS>\<PLATFORM>\DEBUG
$(GPPTEMP PATH)\gpp\export\BIN\<GPP OS>\<PLATFORM>\RELEASE
```

Build the samples

1. Change to the source directory:

```
L:> cd dsplink\gpp\src\samples
```

2. Start the build process. Based on the selected build configuration, some or all of the samples may be built.

```
L:\dsplink\gpp\src\samples> gmake –s [debug | release]
```

3. Upon successful completion of build, the sample application executables shall be created in the following directories:

```
$(DSPLINK)\dsplink\gpp\export\BIN\<GPP OS>\<PLATFORM>\DEBUG
$(DSPLINK)\gpp\export\BIN\<GPP OS>\<PLATFORM>\RELEASE
Or
$(GPPTEMP PATH)\gpp\export\BIN\<GPP OS>\<PLATFORM>\DEBUG
$(GPPTEMP PATH)\gpp\export\BIN\<GPP OS>\<PLATFORM>\RELEASE
```

q To build a single sample application, change to the source directory of the specific sample in the first step above.

10.2.2 DSP-side build

Build the sources

1. Change to the source directory:

```
L:> cd dsplink\dsp\src
```

2. Start the build process:

```
L:\dsplink\dsp\src> gmake –s [debug | release]
```

3. Upon successful completion of build, the DSP libraries shall be created in the following directories:

```
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\DEBUG
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\RELEASE
Or
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
PROCID>\DEBUG
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
PROCID>\RELEASE
```

- Based on the scalability configuration selected, the DSP/BIOS™ LINK libraries generated are:
  - dsplink.lib: Generic DSPLink base library required for using all components supported within DSPLink.
  - dsplinkdata.lib: DSPLink library required for data transfer.
  - dsplinkmsg.lib: DSPLink library required for messaging.
  - dsplinkringio.lib: DSPLink library required for RingIO.
  - dsplinkmplist.lib: DSPLink library required for MPLIST.
  - dsplinkmpcs.lib: DSPLink library required for MPCS.
  - dsplinknotify.lib: DSPLink library required for NOTIFY.
  - dsplinkpool.lib: DSPLink library required for POOL.

Build the samples

1. Change to the source directory:

```
L:> cd dsplink\dsp\src\samples
```

2. Start the build process. Based on the selected build configuration, some or all of the samples may be built.

```
L:\dsplink\dsp\src\samples> gmake –s [debug | release]
```

3. Upon successful completion of build, the sample executable shall be created in the following directories:

```
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\DEBUG
$(DSPLINK)\dsplink\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<D
SP PROCID>\RELEASE
Or
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
PROCID>\DEBUG
$(DSPTEMP PATH)\dsp\export\BIN\DSPBIOS\<PLATFORM>\<DEVICE>_<DSP
```

```
PROCID>\RELEASE
```

- To build a single sample application, change to the source directory of the specific sample in the first step above.

11 Scalability

Depending on application needs, DSPLink can be configured to scale out undesired components. Subsets of the below configurations are supported for different platform-OS configurations.

COMPONENTS and their dependencies	Select The Configuration
Basic DSP boot-loading and control capability.	PROC
NOTIFY – This includes PROC and NOTIFY component.	NOTIFY
POOL – This includes PROC, MPCS along with POOL component.	POOL
MPCS - This includes PROC, POOL along with MPCS component.	MPCS
MPLIST - This includes PROC, POOL, MPCS along with MPLIST component.	MPLIST
RINGIO - This includes PROC, POOL, MPCS, NOTIFY along with RINGIO component.	RINGIO
MSGQ - This includes PROC, POOL, MPCS, and MPLIST along with MSGQ component.	MSGQ
CHNL - This includes PROC, POOL, MPCS, and MPLIST along with CHNL component.	CHNL

- Select the appropriate option while executing the build configuration for DSPLink (*'dsplinkcfg'*). These are applicable for the GPP side as well as DSP side source build.

12 TYPICAL APPLICATION FLOW

This section provides overview of the typical steps involved in the following phases of each component:

Due to the dependency between the components, it is possible that Initialization of a component may depend upon the Execution of another. Application programmers must consider these dependencies when writing their applications.

13 INITIALIZATION

This section provides an overview of various steps involved in initialization phase of each component. These steps ensure that all necessary resources are allocated and appropriately initialized.

13.1 PROC

13.1.1 Typical sequence

1. Do the basic initialization of the component.

   This initialization sequence extends to the lower level components and populates the necessary data structures.

2. Attach to the specific DSP for communication.

   In the process, the lower level components initialize the hardware interfacing the DSP to make it accessible to the GPP.

3. Load an executable on the DSP. This executable contains the application intended to run on the DSP.

The client that attaches first to a DSP becomes the owner of the DSP. Such ownership model is required .so that another client doesn't cause undesirable side affects e.g. stop the DSP/ load another executable etc.

13.1.2 APIs used

1. `PROC_setup ()`
2. `PROC_attach ()`
3. `PROC_load ()`

13.2 POOL

13.2.1 Typical sequence

1. Open the pool from which data buffers or messages are to be allocated.

   The default pools shipped with DSP/BIOS™ LINK are:

   1. SMAPOOL: For zero-copy buffers allocated from memory with shared access across processors.
   2. BUFPOOL: For fixed-size buffers.

   The buffers can be allocated and freed from the pool from the ISR and DPC context.

- Each pool must be initialized only once. However, multiple pools (of differing IDs) using the same pool function table can be configured using the static configuration tool. Each of these pools must be initialized once.

13.2.2 APIs used

1. `POOL_open ()`

13.3 NOTIFY

13.3.1 Typical sequence

1. Register a callback function for notification of the required event from the remote processor. A fixed parameter can be optionally specified during registration, which is received with the callback function when an event notification is received.

13.3.2 APIs used

1. `NOTIFY_register ()`

13.4 MPCS

13.4.1 Typical sequence

1. Create an MPCS instance identified with a system-wide unique name. If memory for the shared MPCS object is not provided by the user, it is allocated based on the POOL IDs provided as part of the attributes.
2. Open the MPCS identified by name to get a handle to the critical section, that can be used for further calls to the MPCS component.

13.4.2 APIs used

1. `MPCS_create ()`
2. `MPCS_open ()`

13.5 MPLIST

13.5.1 Typical sequence

1. Create an MPLIST instance identified with a system-wide unique name. If memory for the shared MPLIST object is not provided by the user, it is allocated based on the POOL IDs provided as part of the attributes.
2. Open the MPLIST identified by name to get a handle to the list that can be used for further calls to the MPLIST component.

13.5.2 APIs used

1. `MPLIST_create ()`
2. `MPLIST_open ()`

13.6 CHNL

13.6.1 Typical sequence

1. Create the channel for data transfer across the GPP and DSP.
2. Allocate the buffer(s) to be used for transferring the data across the channel.
3. Prime the buffers before initiating the data transfer.

The applications must decide on the channels to be used for data transfer. The channel must be opened in appropriate directions on the GPP and DSP to allow the transfer to take place.

13.6.2 APIs used

1. `CHNL_create ()`
2. `CHNL_allocateBuffer ()`

13.7 MSGQ

13.7.1 Typical sequence

1. Open a message queue. All messages destined for the client will be added to this queue.
2. Open a message queue where the asynchronous error messages will be queued.

   This queue can be same as the one created in the previous step.
3. Open the transport towards the DSP to be used for messaging.
4. Locate the remote queue for sending messages by name.

Step 4 may require a response from the DSP. It should, therefore, be executed only after the DSP is running.

The message queues are identified through system-wide unique names.

13.7.2 APIs used

1. `MSGQ_transportOpen ()`
2. `MSGQ_open ()`
3. `MSGQ_setErrorHandler ()`
4. `MSGQ_locate ()`

13.8 RING IO

13.8.1 Typical sequence

1. Create a RingIO identified with a system-wide unique name. The data buffer, attribute buffer, control structure and lock structure are allocated based on the POOL IDs provided as part of the attributes.
2. Open the RingIO in Reader or Writer mode. This returns a client-specific handle to the application, which is used for further calls to the RingIO

component.

3. If required, set the notifier function to be used for the RingIO client. Based on the notification type specified, and whether the RingIO has been opened in Reader or Writer mode, the notification is called when the watermark for full or empty buffer is reached.

13.8.2 APIs used

1. `RingIO_create ()`
2. `RingIO_open ()`
3. `RingIO_setNotifier ()`

14 EXECUTION

This section provides an overview of various steps involved in execution phase of each component.

14.1 PROC

14.1.1 Typical sequence

1. Start execution of the executable that was loaded earlier on the DSP.

   Once the DSP is executing, there isn't much expected from the PROC component.
2. Read from DSP memory.
3. Write to DSP memory.
4. Once the application completes, the execution is stopped.

14.1.2 Relevant APIs

1. `PROC_start ()`
2. `PROC_read ()`
3. `PROC_write ()`
4. `PROC_stop ()`

14.2 POOL

14.2.1 Typical sequence

1. Allocate a buffer from the pool.
2. If required, translate the allocated buffer between different address spaces (user, kernel, physical, DSP).
3. Free the buffer previously allocated from the pool.

14.2.2 APIs used

1. `POOL_alloc ()`
2. `POOL_translateAddr ()`
3. `POOL_free ()`

14.3 NOTIFY

14.3.1 Typical sequence

1. Send a notification of an event to the remote processor, along with an optional payload value.
2. Receive notification of an event from the remote processor. The callback function is invoked with the fixed parameter specified during registration, and a variable payload value received with the event.

14.3.2 APIs used

1. `NOTIFY_notify ()`

14.4 MPCS

14.4.1 Typical sequence

1. Enter the critical section specified by its handle to get exclusive access to the shared structure(s) protected by the MPCS.
2. After performing the required processing on the shared data structure(s) protected by the MPCS, leave the MPCS and make it available to the other processes/processor.

14.4.2 APIs used

1. `MPCS_enter ()`
2. `MPCS_leave ()`

14.5 MPLIST

14.5.1 Typical sequence

1. Place a buffer allocated from the pool at the end of the linked list.
2. Remove a buffer from the head of the linked list.
3. If required, check whether the list is empty.
4. If required, insert a buffer before an existing element in the list.
5. If required, remove the specified element from the list by unlinking it.
6. If required, get a pointer to the first element in the list.
7. If required, get a pointer to the element after the specified element in the list.

14.5.2 APIs used

1. `MPLIST_putTail ()`
2. `MPLIST_getHead ()`
3. `MPLIST_isEmpty ()`
4. `MPLIST_insertBefore ()`
5. `MPLIST_removeElement ()`

6. `MPLIST_first ()`
7. `MPLIST_next ()`

14.6 CHNL

14.6.1 Typical sequence

1. Issue allocated buffer(s) to the channel(s) created earlier. Usually:
   - A primed buffer is issued on an output channel to be received by the remote client on the DSP.
   - An empty buffer is issued on an input channel to receive the data issued by remote client on the DSP.
2. Reclaim a buffer on the channel to which a buffer was issued in the previous step. This is a synchronous operation i.e. the execution of the client is blocked until the IO operation is successful (or a timeout occurs).

14.6.2 Relevant APIs

1. `CHNL_issue ()`
2. `CHNL_reclaim ()`

14.7 MSGQ

14.7.1 Typical sequence

1. Allocate a message using the pool.
2. Send the message to the message queue.
3. Receive the message from the message queue.
4. Get the handle to the source message queue from the received message. This message queue handle can be used for replying to the received message.
5. If required, get information from a message, or set some information within the message, or get information about a message queue.
6. Free the message.

q Step 4 is only required if replying to a received message is desired.

14.7.2 Relevant APIs

1. `MSGQ_alloc ()`
2. `MSGQ_put ()`
3. `MSGQ_get ()`
4. `MSGQ_getSrcQueue ()`
5. `MSGQ_getMsgId ()`, `MSGQ_getMsgSize ()`, `MSGQ_setMsgId ()`, `MSGQ_getDstQueue ()`, `MSGQ_setSrcQueue ()`, `MSGQ_isLocalQueue ()`
6. `MSGQ_free ()`

14.8 RING IO

14.8.1 Typical sequence

1. Acquire a buffer from the RingIO.
2. If the RingIO is opened in Writer mode, if required by the application, set a fixed or variable attribute at an offset within the acquired buffer region. If the RingIO is opened in Reader mode, and if an attribute is present at the present read offset, get the fixed or variable attribute.
3. If required, cancel the previous acquire.
4. If the RingIO is opened in Writer mode, write into the empty buffer acquired, and release the size of buffer that has been initialized. If the RingIO is opened in Reader mode, read from the full buffer acquired, and release the size of buffer that has been read.
5. If required, get information about the current status of the RingIO client.

14.8.2 Relevant APIs

1. `RingIO_acquire ()`
2. `RingIO_setAttribute (), RingIO_getAttribute (), RingIO_setvAttribute (), RingIO_getvAttribute ()`
3. `RingIO_cancel ()`
4. `RingIO_release ()`
5. `RingIO_flush()`
6. `RingIO_getValidSize (), RingIO_getEmptySize (), RingIO_getAcquiredOffset (), RingIO_getAcquiredSize (), RingIO_getWatermark ()`

15 FINALIZATION

This section provides an overview of various steps involved in finalization phase of each component. These steps ensure that all resources allocated in earlier phases are appropriately freed.

15.1 PROC

15.1.1 Typical sequence

1. Detach from the DSP.

   If the client was the owner of the DSP (i.e. was the first to attach to the DSP) then it also finalizes the DSP.
2. Free the resources allocated in the initialization phase.

15.1.2 Relevant APIs

1. `PROC_detach ()`
2. `PROC_destroy ()`

The PROC component must be the last one to be finalized by the application.

15.2 POOL

15.2.1 Typical sequence

1. Close the pool

15.2.2 Relevant APIs

1. `POOL_close ()`

15.3 NOTIFY

15.3.1 Typical sequence

1. Unregister the callback function with the fixed parameter specified during registration. After this, no further notifications for the event are received.

15.3.2 APIs used

1. `NOTIFY_unregister ()`

15.4 MPCS

15.4.1 Typical sequence

1. Close the handle to the MPCS obtained earlier. After this, no further calls can be made to the run-time MPCS APIs.
2. Delete the MPCS instance created earlier.

15.4.2 APIs used

1. `MPCS_close ()`
2. `MPCS_delete ()`

15.5 MPLIST

15.5.1 Typical sequence

1. Close the handle to the MPLIST obtained earlier. After this, no further calls can be made to the run-time MPLIST APIs.
2. Delete the MPLIST instance created earlier.

15.5.2 APIs used

1. `MPLIST_close ()`
2. `MPLIST_delete ()`

15.6 CHNL

15.6.1 Typical sequence

1. Free the buffer(s) allocated on the in the initialization step.
2. Delete the channel.

15.6.2 Relevant APIs

1. `CHNL_freeBuffer ()`
2. `CHNL_delete ()`

15.7 MSGQ

15.7.1 Typical sequence

1. Release the remote message queue.
2. Close the remote transport.
3. Close the local message queue.

15.7.2 Relevant APIs

1. `MSGQ_release ()`
2. `MSGQ_transportClose ()`
3. `MSGQ_close ()`

15.8 RING IO

15.8.1 Typical sequence

1. Close the RingIO.
2. Delete the RingIO.

15.8.2 Relevant APIs

1. `RingIO_close ()`
2. `RingIO_delete ()`

SAMPLE APPLICATIONS

16 LOOP

16.1 Overview

This sample illustrates basic data streaming concepts in DSP/BIOS™ LINK. It transfers data between a task running on GPP and another task running on the DSP

On the DSP side, this application illustrates use of TSK with SIO and SWI with GIO.

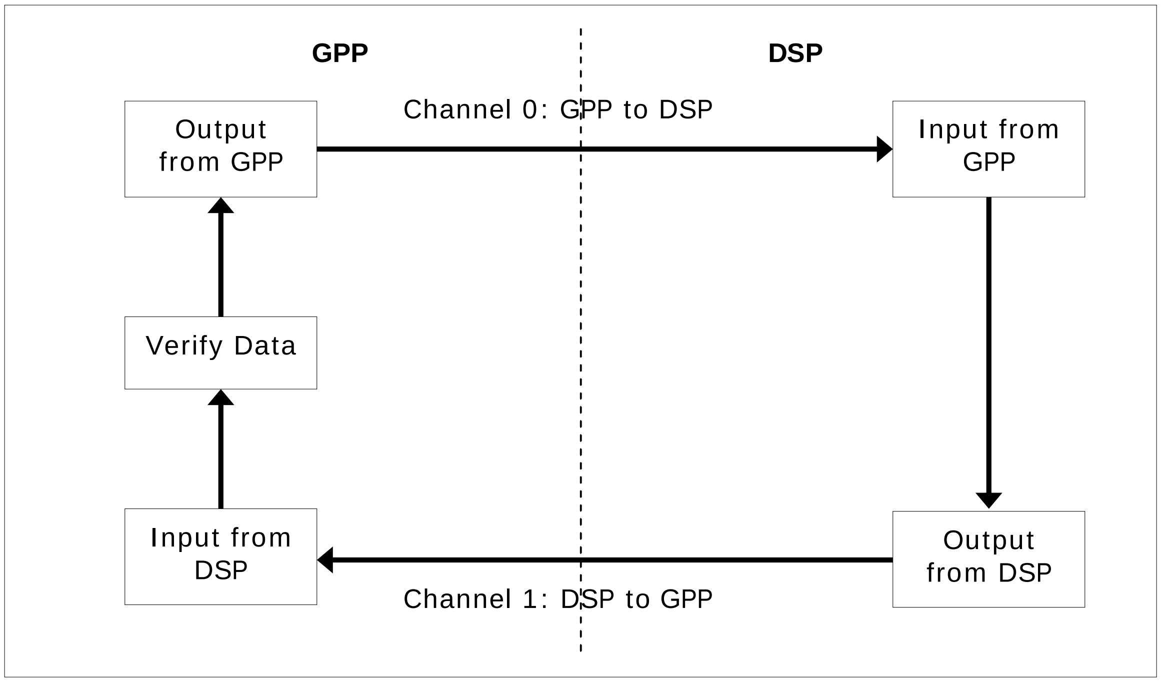


Figure 6. Data flow in the sample application – LOOP

16.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
2. It opens the pool to be used for allocating the data transfer buffer(s).
3. It loads DSP executable (`loop.out`) on the DSP.
4. It creates channels CHNL_ID_INPUT and CHNL_ID_OUTPUT for data transfer.
5. It allocates and primes buffer(s) of specified size for data transfer on these channels.

EXECUTION

1. The client starts the execution on DSP.
2. It fills the output buffer with sample data.

3. It then issues the buffer on CHNL_ID_OUTPUT and waits to reclaim it. The reclaim is specified to wait forever.
4. The completion of reclaim operation indicates that the buffer has been transferred across the physical link.
5. It issues an empty buffer on CHNL_ID_INPUT and waits to reclaim it. The reclaim is specified to wait forever.
6. Once the buffer is reclaimed, its contents are compared with those of the buffer issued on CHNL_ID_OUTPUT. Since this is a loop back application the contents should be same.
7. The client repeats the steps 3 through 6 for number of times specified by the user.
8. It stops the DSP execution.

FINALIZATION

1. The client frees the buffers allocated for data transfer.
2. It deletes the channels CHNL_ID_INPUT and CHNL_ID_OUTPUT.
3. It closes the pool.
4. It detaches itself from DSP and destroys the PROC component.

16.1.2 On the DSP side

16.1.2.1 Using TSK with SIO

INITIALIZATION

1. The client task `tskLoop` is created in the function `main ()`.
2. The pool to be used for allocating the data transfer buffers is configured in main () with the buffer size and number of buffers to be allocated.
3. This task creates SIO channels for data transfer - INPUT_CHANNEL and OUTPUT_CHANNEL.
4. It allocates and primes the buffer(s) for to be used for data transfer.

EXECUTION

1. The task issues an empty buffer on INPUT_CHANNEL and waits to reclaim it. The reclaim is specified to wait forever.
2. It then issues the same buffer on OUTPUT_CHANNEL and waits to reclaim it. The reclaim is specified to wait forever.
3. The completion of reclaim operation indicates that the buffer has been transferred across the physical link.
4. These steps are repeated until the number of iterations passed as an argument to the DSP executable is completed.

FINALIZATION

1. The task frees the buffers allocated for data transfer.
2. It deletes the SIO channels INPUT_CHANNEL and OUTPUT_CHANNEL.

16.1.2.2 Using SWI with GIO

INITIALIZATION

1. In the function `main ()`, GIO channels for data transfer - INPUT_CHANNEL and OUTPUT_CHANNEL are created.
2. The pool to be used for allocating the data transfer buffers is configured in main () with the buffer size and number of buffers to be allocated.
3. A SWI object is created for doing the data transfer. One of the attributes for the SWI object is the callback function `loopbackSWI`. This function is called when the SWI is posted on completion of READ and WRITE requests on the GIO channels.
4. The buffers for to be used for data transfer are allocated and primed.

EXECUTION

1. To initiate the data transfer a READ request on the input buffer is submitted on the INPUT_CHANNEL.
2. Once the SWI is posted, contents of input buffer are copied to the output buffer.
3. The empty input buffer is reissued onto the input channel and the filled buffer is issued onto the output channel.
4. The SWI is posted again after the completion of both requests.
5. Steps 2 to 4 continue till the time GPP application is issuing buffers.

FINALIZATION

In the sample, the SWI is continuously posted due to READ and WRITE requests. So it would never reach the finalization. The finalization sequence, however, would be:

1. The buffers allocated for data transfer are freed.
2. The GIO channels INPUT_CHANNEL and OUTPUT_CHANNEL are deleted.

16.1.3 Invoking the application

The loop sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./loop.out`
Buffer Size	`1024`
Number of iterations	`10000`
DSP Processor Id	`0`

q The sample can be executed for infinite iterations by specifying the number of iterations as 0.

q DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier.

17 MESSAGE

17.1 Overview

This sample illustrates basic message transferring concepts in DSP/BIOS™ LINK. It transfers messages between a task running on GPP and another task running on the DSP.

On the DSP side, this application illustrates use of TSK and SWI with MSGQ.

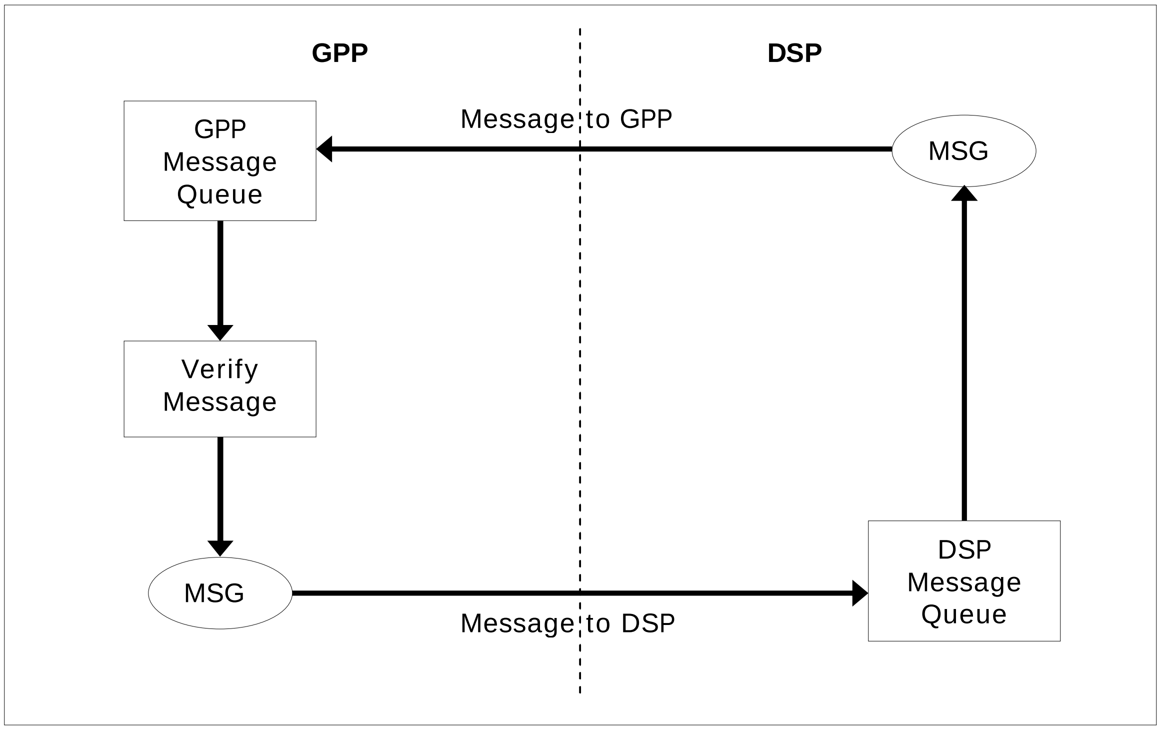


Figure 7. Message flow in the sample application – MESSAGE

17.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP.
2. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
3. It opens the pool required for allocating the messages, depending on the physical link to be used for the data driver.
4. It then opens a message queue identified by a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It loads DSP executable (`message.out`) on the DSP
7. The client starts the execution on DSP.
8. It then opens the remote transport.

9. It then attempts to locate the queue opened on the DSP side. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not opened), it sleeps for some time and tries to locate the queue again.

EXECUTION

1. The client tries to get a message on the local queue. The get operation is specified to wait forever.
2. On receiving the message, it verifies the validity of contents of the received message.
3. It then sends the same message back to the DSP message queue.
4. Once the message is received, its contents are compared with the sequence number, which is incremented every time a get is successful.
5. The client repeats the steps 2 through 4 for number of times specified by the user, or infinitely if so specified.
6. For the case when finite number of iterations is specified, it frees the message that was received for the last get operation.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It closes the remote transport.
3. It stops the DSP execution.
4. It resets the error handler which was set in the create phase.
5. It closes the local message queue.
6. It closes the pool.
7. It detaches itself from DSP.
8. It destroys the PROC component.

17.1.2 On the DSP side

17.1.2.1 Using TSK with MSGQ

INITIALIZATION

1. The pool to be used for messaging is configured statically through the global `POOL_config` variable.
2. The MSGQ component is configured statically through the global `MSGQ_config` variable.
3. The client task tskMessage is created in the function `main ()`.
4. It opens a message queue with a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It then attempts to locate the queue opened on the GPP side. Locate is specified to wait forever. If the Locate call was unsuccessful (GPP queue still not opened), it sleeps for some time and tries to locate the queue again. The locate operation is synchronous.

EXECUTION

1. The task allocates a message from the pool.
2. It sends this message to the GPP message queue located earlier.
3. The task then tries to get a message on the local queue. The get operation is specified to wait forever.
4. These steps are repeated for number of iterations specified by the user, or infinitely, if so specified.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It resets the error handler which was set in the create phase.
3. It closes the local message queue.

17.1.2.2 Using SWI with MSGQ

INITIALIZATION

1. The `SWIMESSAGE_create ()` is called from the function `main ()`.
2. A SWI object is created for doing the message transfer. One of the attributes for the SWI object is the callback function `messageSWI`.
3. It opens a message queue with a specific name on the local processor. The SWI object is used as the notification object for messages received on the message queue. This ensures that the SWI is posted each time a message is received on the message queue.
4. It sets the above-opened queue as the error handler.
5. It finally posts the SWI to be used for the execution phase of the application.
6. It attempts to locate the queue created on the GPP side. This Locate operation is asynchronous.

EXECUTION

1. When the message SWI is posted for the first time, the client attempts to locate the message queue opened on the GPP-side. The Locate operation is asynchronous.
2. The message SWI is posted whenever a message is received on the DSP message queue. The first message received indicates completion of the asynchronous locate request. The asynchronous locate message has message ID `MSGQ_ASYNCLOCATEMSGID`. On receiving this message, the SWI function sets its handle for the GPP message queue for sending messages to it, and frees the received message.
3. It then allocates a new message and sends it to the GPP message queue to initiate the message transfer.
4. Each subsequent time that the SWI is posted indicates that a new message is received. The SWI sends the same message back to the GPP message queue.
5. In case the ID of the received message is `MSGQ_ASYNCERRORMSGID`, it indicates that an error occurred. The error type is identified through the contents of the received error message.

FINALIZATION

In the message sample, the SWI is continuously posted whenever a message is ready on the local message queue. So it would never reach the finalization. The finalization sequence, however, would be:

1. The client releases the remote message queue on the DSP side.
2. It resets the error handler which was set in the create phase.
3. It closes the local message queue.

17.1.3 Invoking the application

The message sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./message.out`
Number of iterations	`10000`
DSP Processor id	`0`

- The sample can be executed for infinite iterations by specifying the number of iterations as 0.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

.

18 SCALE

18.1 Overview

This sample illustrates a combination of data streaming and messaging concepts in DSP/BIOS™ LINK. It transfers data between a task running on GPP and another task running on the DSP and sends messages from GPP to DSP.

On the DSP side, this application illustrates use of TSK with SIO & MSGQ, and SWI with GIO & MSGQ.

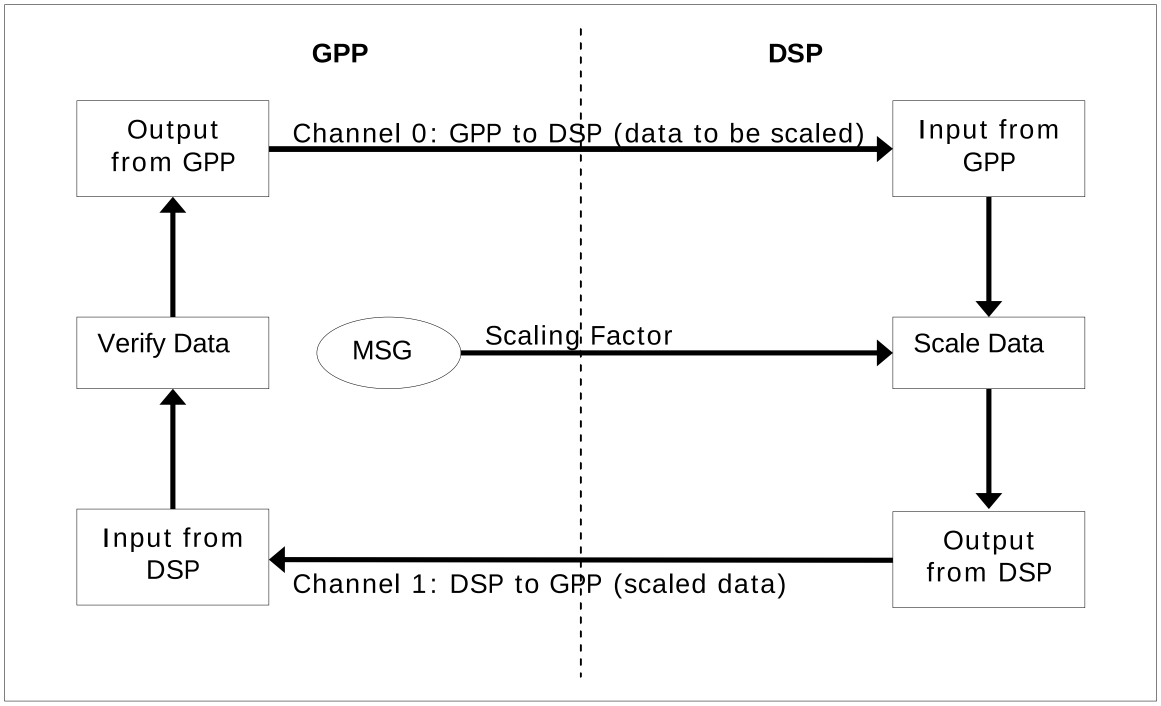


Figure 8. Data and message flow in the sample application – SCALE

18.1.1 On the GPP side

INITIALIZATION

1. The client calls APIs required for making the DSP accessible.
2. It opens the pool required for allocating the messages, depending on the physical link to be used for the data driver.
3. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
4. It loads DSP executable (`scale.out`) on the DSP
5. It creates channels CHNL_ID_INPUT and CHNL_ID_OUTPUT for data transfer.
6. It allocates and initializes buffer(s) of specified size for data transfer on these channels.
7. The client starts the execution on DSP.
8. It then opens the remote transport.

EXECUTION

1. It attempts to locate the MSGQ created on the DSP side. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not created), it sleeps for some time and tries to locate the queue again.
2. It issues the buffer on CHNL_ID_OUTPUT and waits to reclaim it. The reclaim is specified to wait forever.
3. The completion of reclaim operation indicates that the buffer has been transferred across the physical link.
4. It issues an empty buffer on CHNL_ID_INPUT and waits to reclaim it. The reclaim is specified to wait forever.

5. Once the buffer is reclaimed, its contents are compared with those of the buffer issued on CHNL_ID_OUTPUT. The DSP-side application is initialized with a scaling factor, which it uses to scale the data.
6. Every 100 iterations of data transfer, the client sends a message to the DSP-side MSGQ with a new scaling factor within it. Following this, all further buffers received from the DSP are expected to contain the scaled data.
7. The client repeats the steps 2 through 7 for number of times specified by the user.
8. The client releases the remote message queue on the DSP side.

FINALIZATION

1. The client closes the remote transport.
2. It then stops the DSP execution.
3. The client frees the buffer(s) allocated for data transfer.
4. It deletes the channels CHNL_ID_INPUT and CHNL_ID_OUTPUT.
5. It then closes the pool.
6. It detaches itself from DSP.
7. It closes the local transport.
8. Finally, it destroys the PROC component.

18.1.2 On the DSP side

18.1.2.1 Using TSK with SIO and MSGQ

INITIALIZATION

1. The client task tskScale is created in the function `main ()`.
2. The pool required for allocating the messages and data buffers is configured as required by the application, depending on the physical link to be used for the data driver.
3. This task creates SIO channels for data transfer - INPUT_CHANNEL and OUTPUT_CHANNEL.
4. It allocates and initializes the buffer to be used for data transfer.
5. It then opens a message queue identified by a specific name on the local processor.

EXECUTION

1. The task issues an empty buffer on INPUT_CHANNEL and waits to reclaim it. The reclaim operation is specified to wait forever.

5. The task tries to get a message on the local queue. The get operation is specified with no timeout. This results in returning a message if it is already available on the specified MSGQ.
6. If a message is available, the new scaling factor is extracted from it. This scaling factor is used to multiply the contents of the buffer received from the GPP.

2. It then issues the scaled buffer on OUTPUT_CHANNEL and waits to reclaim it. The reclaim operation is specified to wait forever.

3. The completion of reclaim operation indicates that the client on the GPP has received the buffer.
4. These steps are repeated until the number of iterations passed as an argument to the DSP executable is completed.

FINALIZATION

1. In its delete phase, the task first deletes the local message queue.
2. It then deletes the SIO channels INPUT_CHANNEL and OUTPUT_CHANNEL.
3. The task frees the buffer allocated for data transfer.

18.1.2.2 Using SWI with GIO and MSGQ

INITIALIZATION

1. SWISCALE_create is called from the function `main ()`.
2. The pool required for allocating the messages and data buffers is configured as required by the application, depending on the physical link to be used for the data driver.
3. It then creates two GIO channels for data transfer - INPUT_CHANNEL and OUTPUT_CHANNEL.
4. Two SWI objects are created, one for doing data transfer (dataSWI), and the other for message transfer (msgSWI). The data SWI function is called when the SWI is posted on completion of READ and WRITE requests on the data channel. The message SWI is posted whenever a message is received.
5. The buffers to be used for data transfer are then allocated and initialized.
6. It then opens a message queue identified by a specific name on the local processor (DSP).

EXECUTION

1. To initiate the data transfer a READ request on the input buffer is submitted on the INPUT_CHANNEL.
2. Once the SWI is posted, contents of input buffer are scaled by the current scaling factor and transferred to the output buffer.
3. The empty input buffer is reissued onto the input channel and the filled buffer is issued onto the output channel.
4. The SWI is posted again after the completion of both requests.
5. Whenever a message is received on the created MSGQ, the message SWI is posted. This SWI checks if a message is available by attempting to get a message with no timeout specified. If present, the new scaling factor is extracted from the message, and saved. This scaling factor is used for scaling all data buffers received from that time onwards.
6. Steps 1 to 5 continue till the time GPP application is issuing buffers.

FINALIZATION

In the sample, the data SWI is continuously posted due to READ and WRITE requests. Similarly, the message SWI is continuously posted whenever a message is ready on the local message queue. So they would never reach the finalization phase. The finalization sequence, however, would be:

1. The data and message SWIs are deleted.
2. The local message queue is deleted.
3. The GIO channels INPUT_CHANNEL and OUTPUT_CHANNEL are deleted.
4. The buffers allocated for data transfer are freed.

18.1.3 Invoking the application

The scale sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./scale.out`
Buffer Size	`1024`
Number of iterations	`10000`
DSP Processor Id	`0`

q The sample can be executed for infinite iterations by specifying the number of iterations as 0.

q DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier.

19 READWRITE

19.1 Overview

This sample illustrates large buffer transfer through direct writes to and reads from DSP memory. It transfers a large size data buffer between the GPP and DSP using `PROC_Read ()` and `PROC_Write ()` API's and tasks running on the DSP

On the DSP side, this application illustrates use of TSK with MSGQ.

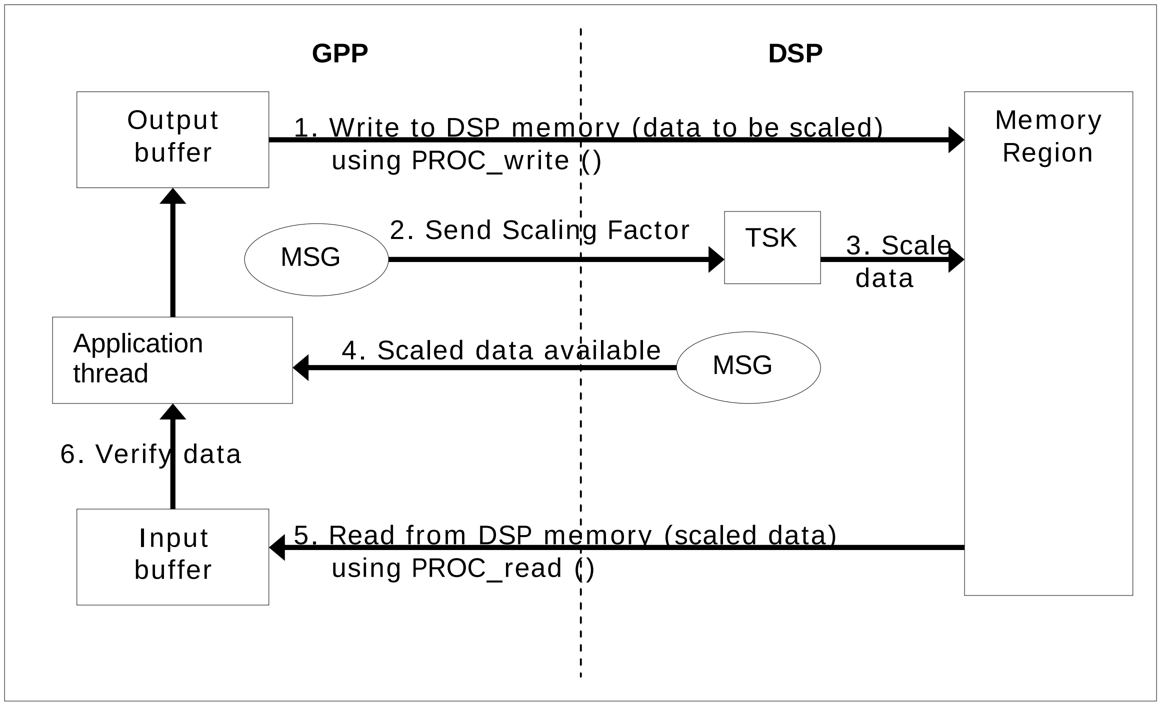


Figure 9. Data and message flow in the sample application – READWRITE

19.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP.
2. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
3. It opens the pool required for allocating the messages, depending on the physical link to be used for the data driver.
4. It then opens a message queue identified by a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It loads DSP executable (readwrite.out) on the DSP
7. The client starts the execution on DSP.
8. It then opens the remote transport.
9. It then attempts to locate the queue opened on the DSP side. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not opened), it sleeps for some time and tries to locate the queue again.

EXECUTION

1. The client allocates a buffer of the required size for both input buffer and output buffer.
2. The client primes the data regions to allow for data integrity check to ensure data transfer has happened correctly.
3. It writes the data buffer to the DSP using `PROC_Write ()` API.

4. The client writes a message to inform the DSP that the data buffer has been written on the DSP.
5. The client sends a message to the DSP-side MSGQ with a new scaling factor within it. Following this, all further buffers received from the DSP are expected to contain the scaled data
6. It then waits for a message from the DSP that will confirm it has the written data.
7. The client then reads from the DSP region using the `PROC_Read ()` API.
8. This is followed by a data integrity check to ensure validity of buffer contents written from the GPP to the DSP and read by the GPP from the DSP.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It closes the remote transport.
3. It stops the DSP execution.
4. It resets the error handler which was set in the create phase.
5. It closes the local message queue.
6. It closes the pool.
7. It detaches itself from DSP.
8. It destroys the PROC component.

19.1.2 On the DSP side

19.1.2.1 Using TSK with MSGQ

INITIALIZATION

1. The pool to be used for messaging is configured statically through the global POOL_config variable.
2. The MSGQ component is configured statically through the global `MSGQ_config` variable.
3. The client task `tskReadWrite` is created in the function `main ().`
4. It opens a message queue with a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It then attempts to locate the queue opened on the GPP side. Locate is specified to wait forever. If the Locate call was unsuccessful (GPP queue still not opened), it sleeps for some time and tries to locate the queue again. The locate operation is synchronous.

EXECUTION

1. The task tries to get a message on the local queue which will inform that the data has been written by the GPP. The get operation is specified to wait forever.
2. If a message is available, the new scaling factor is extracted from it. This scaling factor is used to multiply the contents of the buffer received from the GPP and the resulting values are written to a different region.
3. It then issues a message saying that the scaled buffer is ready for data transfer.

4. These steps are repeated for number of iterations specified by the user if so specified.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It resets the error handler which was set in the create phase
3. It closes the local message queue.

19.1.3 Invoking the application

The readwrite sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./readwrite.out`
DSP address	`2414804992`
Buffer Size	`1024`
Number of iterations	`10000`
DSP Processor Id	`0`

- The sample can be executed for infinite iterations by specifying the number of iterations as 0.
- The DSP address mentioned above is for the Davinci platform. This needs to be specified as a valid DSP address for all platforms.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

20 MAPREGION

20.1 Overview

This sample illustrates direct pointer access to the DSP memory region over PCI through DSP/BIOS™ LINK. This sample application is supported only on the DM642_PCI platform. The following diagram shows the behavior of the application.

The DSP-side for the sample is the same as the readwrite sample.

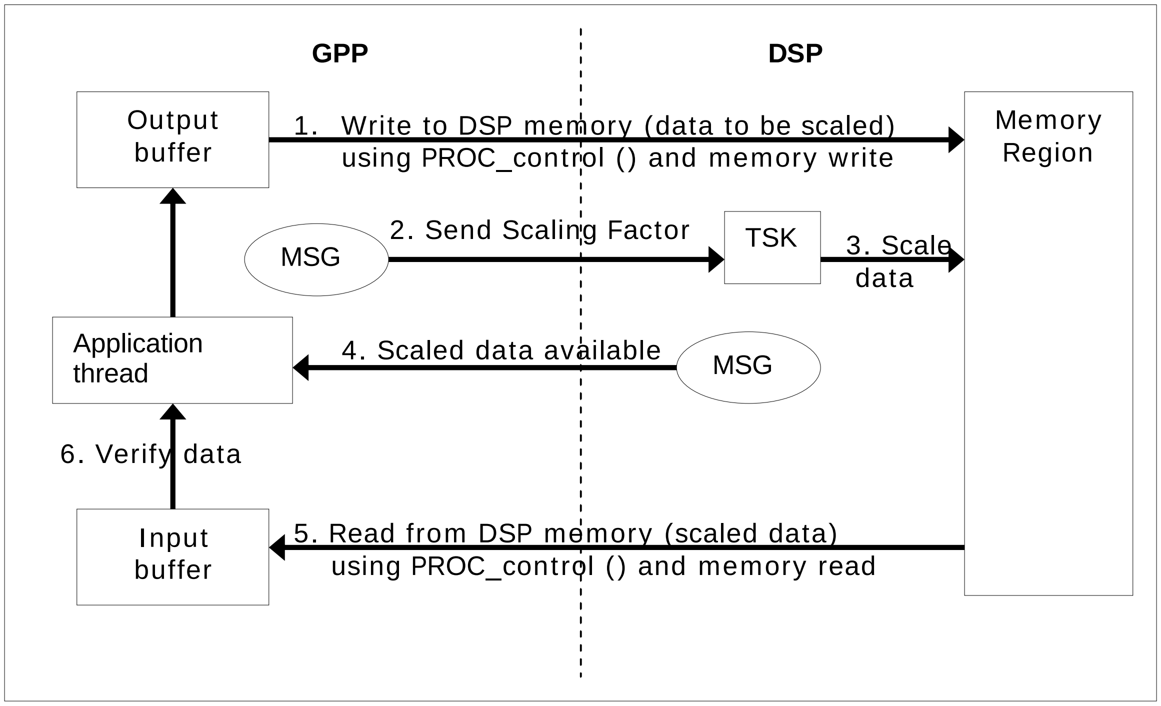

GPP
DSP
Output buffer
1. Write to DSP memory (data to be scaled) using PROC_control () and memory write
Memory Region
MSG
2. Send Scaling Factor
TSK
3. Scale data
Application thread
4. Scaled data available
MSG
6. Verify data
Input buffer
5. Read from DSP memory (scaled data) using PROC_control () and memory read

Figure 10. Data and message flow in the sample application – MAPREGION

20.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP.
2. It then attaches to the DSP identified by `DSP PROCESSOR IDENTIFIER`.
3. It opens the pool required for allocating the messages, depending on the physical link to be used for the data driver.
4. It then opens a message queue identified by a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It loads DSP executable (`readwrite.out`) on the DSP
7. The client starts the execution on DSP.
8. It then opens the remote transport.
9. It then attempts to locate the queue opened on the DSP side. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not opened), it sleeps for some time and tries to locate the queue again.

EXECUTION

1. The client gets control of buffers in the DSP memory region of the required size for both input buffer and output buffer using `PROC_Control ()` API.
2. The client primes the data regions to allow for data integrity check to ensure data transfer has happened correctly.

3. It writes the data buffer to the DSP using the pointer which gives it direct access to the input buffer.
4. It then relinquishes control of the DSP memory areas using the `PROC_Control ()` API.
5. The client sends a message to the DSP-side MSGQ with a new scaling factor within it. Following this, all further buffers received from the DSP are expected to contain the scaled data.
6. It then waits for a message from the DSP that will confirm it has the written data.
7. The client then again gets control of the input buffer in the DSP memory region of required size using the `PROC_Control ()` API. It then reads the contents of the memory using the direct pointer access obtained.
8. It then relinquishes control of the DSP memory area using the `PROC_Control ()` API.
9. This is followed by a data integrity check to ensure validity of buffer contents written from the GPP to the DSP and read by the GPP from the DSP.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It closes the remote transport.
3. It stops the DSP execution.
4. It resets the error handler which was set in the create phase.
5. It closes the local message queue.
6. It closes the pool.
7. It detaches itself from DSP.
8. It destroys the PROC component.

20.1.2 On the DSP side

The DSP-side used is the same as the readwrite sample.

20.1.3 Invoking the application

The mapregion sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./readwrite.out`
Buffer Size	`1024`
Number of iterations	`10000`
DSP Processor Id	`0`

- The sample can be executed for infinite iterations by specifying the number of iterations as 0.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

21 RING_IO

21.1 Overview

This sample illustrates the usage of the RingIO component in DSP/BIOS™ LINK to stream data between the GPP and DSP using two RingIO instances. It transfers data between application (thread/process) running on the GPP and another task running on the DSP. In Linux, this application runs as a set of processes or a set of thread s in a process. In PrOS, it runs as a set of tasks.

In subsequent sections each thread/process/task in the application is treated as a client.

On the DSP side, this application illustrates use of TSK with RingIO.

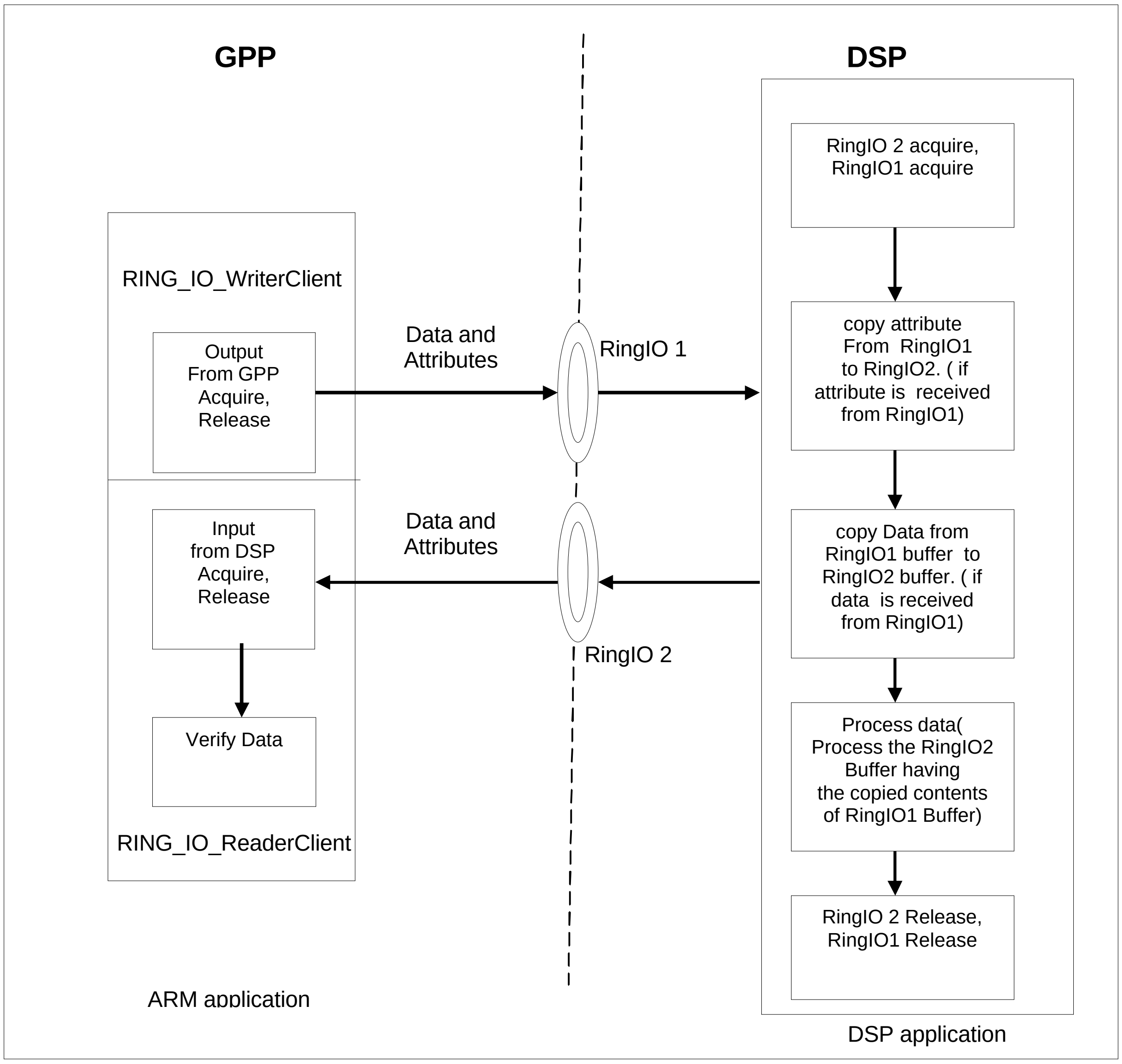

GPP
DSP
RING_IO_WriterClient
Output From GPP Acquire, Release
Data and Attributes
RingIO 1
Input from DSP Acquire, Release
Data and Attributes
RingIO 2
Verify Data
RING_IO_ReaderClient
ARM application
RingIO 2 acquire, RingIO1 acquire
copy attribute From RingIO1 to RingIO2. (if attribute is received from RingIO1)
copy Data from RingIO1 buffer to RingIO2 buffer. (if data is received from RingIO1)
Process data(Process the RingIO2 Buffer having the copied contents of RingIO1 Buffer)
RingIO 2 Release, RingIO1 Release
DSP application

Figure 11. Data flow in the sample application – RING_IO

21.1.1 On the GPP side

INITIALIZATION

1. The GPP application calls APIs required for making the DSP accessible.
2. It initializes the RingIO component for the DSP. (PROC_setup internally does this)
3. It opens the pool required for allocating the RingIO data buffers, attribute buffers, control Structures and lock objects.
4. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
5. It loads DSP executable(ringio.out) on to the DSP
6. It starts the execution on DSP.
7. The GPP client application creates the RingIO (RINGIO1) to be used for

   sending data to the DSP with the GPP as the writer.
8. It then creates two clients one to send data and/or attributes to DSP (GPP RingIO Writer) and another one to read data from the DSP (GPP RingIO reader).

EXECUTION

GPP side application has two clients (RING_IO_WriterClient and RING_IO_ReaderClient) running to send and receive data to/from DSP. The RING_IO_WriterClient sends data to DSP and the RING_IO_ReaderClient receives data from DSP.

RING_IO_WriterClient

1. This client opens the RINGIO1 (created by the GPP) in write mode to read data from DSP.

2. It (GPP RingIO writer) sets the notifier for the Writer with the specific watermark value of the buffer size used for data transfer. Pointer to a semaphore is passed to the notifier function. The notifier function post the semaphore passed to it, resulting in unblocking the application which would be waiting on it.

2. It inserts an attribute (RINGIO_DATA_START) in to RINGIO1 to indicate the start of the data transfer.

3. It sends a force notification to unblock RINGIO1 reader (DSP) and to allow it to read data from the RingIO.

4. It sets a variable attribute before acquiring any buffer. This variable attribute payload contains size, action, factor fields.
   - § Size is the size of the received data (in bytes) that needs to be considered by DSP for processing based on the action and the factor fields.
   - § Action tells the DSP what action needs to be taken on the received data (i.e. multiply or division).
   - § Factor holds the other operand used in processing the received data by DSP.

5. It acquires and initializes the RINIGIO1 buffer .Then it releases the buffer.

6. If buffer is not available, the application waits on a semaphore, which will be posted by the notification function registered for RINIGIO1 writer with watermark equal to the buffer size.

7. Steps 4 to 6 are repeated for the number of bytes specified.

8. After finishing the data transfer, it inserts an attribute (RINGIO_DATA_END) indicating end of data transmission from GPP. It also sends a force notification to the RINGIO1 reader (DSP).This force notification allows DSP to come out of blocked state, if it is waiting for data notification. Because finally we are sending only the attribute and not data.

9. It deletes the created semaphore

10. It closes the RingIO1 opened in write mode and exits.

RING_IO_ReaderClient

1. This client opens the RINGIO2 in read mode to read data from DSP.
2. It (GPP RingIO reader) sets the notifier for the reader (RINGIO2) with the specific watermark value of zero. Pointer to a semaphore is passed to the notifier function. The notifier function post the semaphore passed to it, resulting in unblocking the reader task which would be waiting on it.
3. It waits on semaphore to receive a start notification from the DSP.
4. After receiving notification from the RINGIO2 writer (i.e. DSP), it tries to get the start attribute (RINGIO_DATA_START). If the start attribute is received, reader task starts reading data.
5. It acquires data buffer in read mode from the RINGO2 and verifies the contents based on the variable attribute received prior to this acquire call. This task always tries to acquire the full buffer and gets what is available in the RINGIO2. If nothing is available, it waits on a semaphore for notification.
6. Step 4 is performed repeatedly until it receives end of data transfer attribute (RINIGIO_DATA_END) from DSP.
7. It deletes the created semaphore
8. It closes the RingIO2 opened in read mode and exits.

FINALIZATION

1. It deletes the RingIO1 created by the GPP-side.
2. It then stops the DSP execution.
3. It then closes the pool.
4. It finalizes the RingIO component for the DSP.
5. It detaches itself from DSP.
6. Finally, it destroys the PROC component.

21.1.2 On the DSP side

INITIALIZATION

1. DspLink is initialized in the main ().
2. Then the client task tskRingIo is created in the function main ().
3. This task creates the RingIO (RINGIO2) to be used for sending data to the GPP with the DSP as the writer.
4. It then opens the RINGIO2 (created by the DSP) in writer mode with need exact flag set.
5. It also waits till open call for RINGIO1 in reader mode is successful with need exact flag not set. If it is able to open the RingIO1, then the RingIO1 has been created by the GPP.

EXECUTION

Dsp client application performs the following in the execute phase.

1. The task sets the notifier for the Writer and Reader with the specific watermark value of the buffer size used for data transfer. Pointer to a semaphore is passed to each notifier function. The notifier functions post the semaphore passed to it, resulting in unblocking the application which would be waiting on it.
2. The task then waits on a semaphore (RINGIO1 reader semaphore) for a notification.
3. If it gets the notification, it tries to get data transfer start attribute (RINGIO_DATA_START) from RINGIO1.
4. If it is able to get the data transfer start attribute (RINGIO_DATA_START), it inserts the same attribute in to RINGIO2.
5. It acquires the buffer from RINGIO2 and then from RINGIO1.
6. If it fails to acquire the buffer either from the RINGIO1 or from the RINGIO2,
7. It waits on a corresponding semaphore for the notification.
8. If it is able to acquire the buffers, it copies the data contents from the RINGIO1 buffer to RINGIO2 buffer based on the received RINGIO1 buffer size.
9. It processes the RINGIO2 buffer contents based on the variable attribute (contains action, data size and factor), received from the RINGIO1.
10. It also sets the same variable attribute in to RINGIO2 buffer at zero offset.
11. Then it releases the input buffer (RINGIO1 buffer).
12. It releases the output buffer (RINGIO2 buffer) of size equal to the RINGIO1 received buffer size and cancels the remaining RINGIO2 buffer.
13. Steps 5 to 11 are repeated until it gets the end of data transfer attribute (RINGIO_DATA_END) from the RINGIO1.
14. It cancels the RINGIO2 buffer which is acquired and sets the end of data transfer attribute (RINGIO_DATA_END) attribute in to RINGIO2.

FINALIZATION

1. In its delete phase, it closes the RingIO2 opened in writer mode.
2. It deletes the RingIO2 created by the DSP.
3. It then closes the RingIO1 opened in reader mode.
4. Finally, the task frees all the temporary buffers allocated.

21.1.3 Invoking the application

The RING_IO sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./ringio.out`
RingIO Data Buffer Size	`10240`
Number of Bytes to transfer	`10240`
DSP Processor Id	`0`

- The sample creates two RingIOs with the data buffer size equal to RingIO Data Buffer Size specified through the command line arguments. The minimum value that can be specified is 1024 Bytes. The maximum value depends on the size of the memory configured for DSPLink.
- The RingIO Data buffer size can be given between 1k bytes to 200k bytes with the Default memory configuration provided with the link.
- The sample can be executed infinitely by specifying Number of Bytes to transfer as zero.
- By default sample runs in multithread mode. To run the sample in multi process mode, define RINGIO_MULTI_PROCESS flag in $DSPLINK\gpp\src\samples\ring_io\Linux\COMPONENT file and build the sample. This multi process mode is applicable only for Linux.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

22 MP_LIST

22.1 Overview

This sample illustrates the usage of the MPLIST component in DSP/BIOS™ LINK to stream data between the GPP and DSP using multi-processor list instance. It transfers data between a task running on GPP and another task running on the DSP.

On the DSP side, this application illustrates use of TSK with MPLIST.

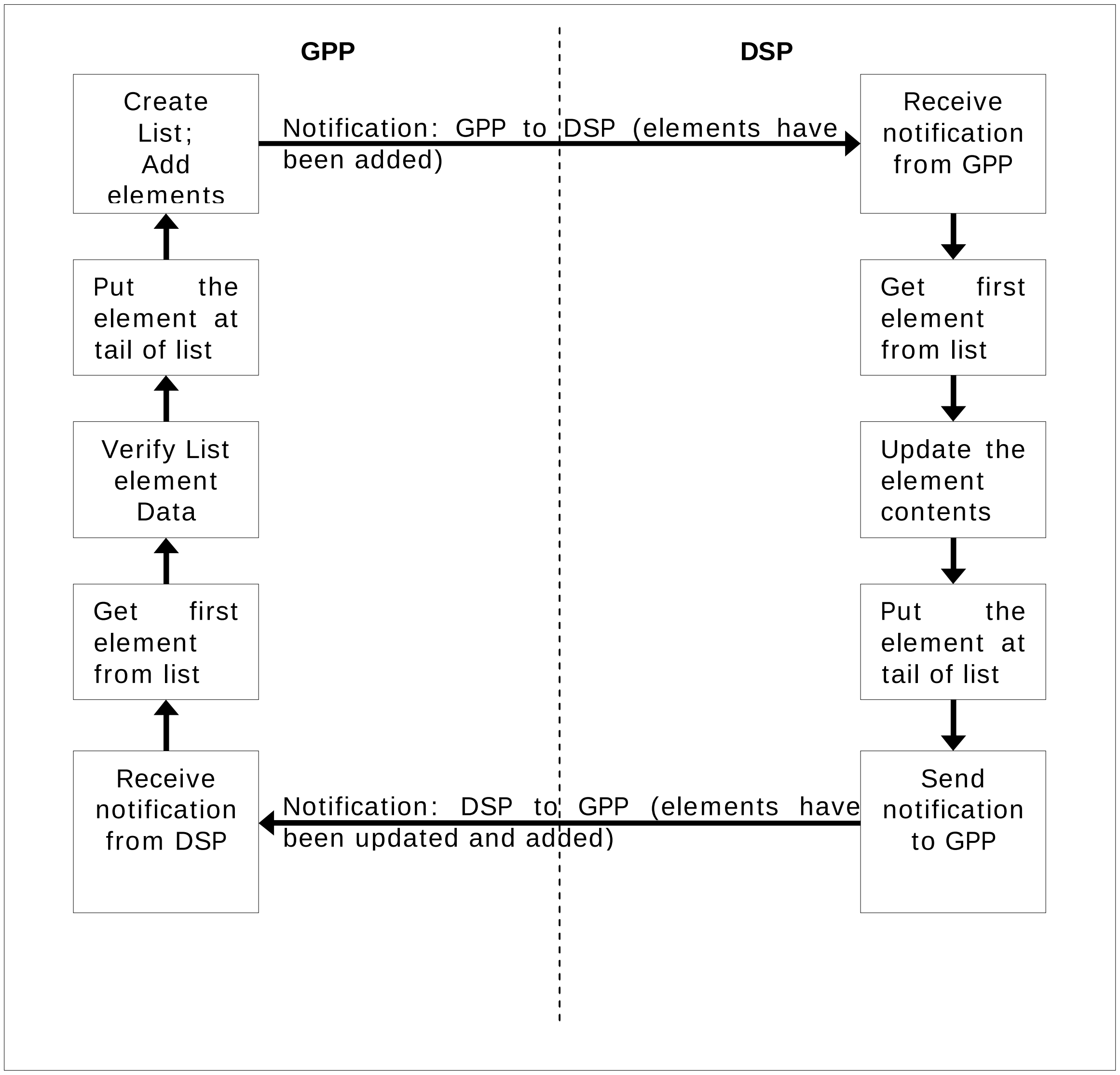

GPP
DSP
Create List; Add elements
Notification: GPP to DSP (elements have been added)
Receive notification from GPP
Put the element at tail of list
Get first element from list
Verify List element Data
Update the element contents
Get first element from list
Put the element at tail of list
Receive notification from DSP
Notification: DSP to GPP (elements have been updated and added)
Send notification to GPP

Figure 12. Data flow in the sample application – MP_LIST

22.1.1 On the GPP side

INITIALIZATION

1. The client calls APIs required for making the DSP accessible.
2. It initializes the MPLIST component for the DSP.
3. It opens the pool required for allocating the MPLIST data structures including the list itself and the number of elements as specified by the user.
4. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
5. It loads DSP executable (`mplist.out`) on the DSP
6. It creates the GPPMPLIST (MPLIST instance) to be used for sending data to the DSP.
7. The client starts the execution on DSP.

8. The client sets up notification which will enable the GPP to know when the DSP has modified the list elements.

EXECUTION

1. The client allocates memory for the MPLIST and the number of list elements as specified by the user.
2. The client gets a handle to the created MPLIST.
3. The GPP sets the list element using the iteration number and its position in the list. The list element is then added to the tail of the list. This is done for the number of elements as specified by the user.
4. After the elements have been added to the list, the GPP sends a token notification to the DSP indicating the same.
5. It then waits for notification for DSP indicating that the DSP has finished modifying the elements in the shared list.
6. After receiving notification it verifies that the modification done by the DSP is a function of the iteration number and the list position.
7. Steps 2 to 6 are repeated for the number of iterations specified.

FINALIZATION

1. The client closes the MPLIST opened in reader mode.
2. It then deletes the MPLIST instance.
3. It then stops the DSP execution.
4. It then closes the pool.
5. It detaches itself from DSP.
6. Finally, it destroys the PROC component.

22.1.2 On the DSP side

INITIALIZATION

1. The client task `tskMpList` is created in the function `main ()`.
2. This task initializes the MPLIST, NOTIFY and MPCS components.
3. It then opens the MPLIST instance (GPPMPLIST) created by the GPP. This enables it to get a handle with which it can perform the list operations.
4. It then registers a notification for the event callback to know when the GPP has added all list elements to the list.

EXECUTION

1. The task waits for notification from the GPP side which will tell that all list elements have been added to the list.
2. After the notification is received, the DSP pops the head off the list. It modifies the data structure within the list element by setting a value data element which is a function of the iteration number data element and the position in the list data element. It then adds the element to the tail of the list. This is done for all elements in the list.
3. After completing the modification, the DSP send a notification to the GPP-side indicating the same.

4. Steps 1 to 3 are repeated for the number of iterations specified.

FINALIZATION

1. In the delete phase, it closes the MPLIST handle.
2. It also un-registers notification for the event callback..
3. In its delete phase, the task first frees all the temporary buffers allocated.

22.1.3 Invoking the application

The MP_LIST sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./mplist.out`
Number of iterations	`10000`
Number of elements	`100`
DSP processor Id	`0`

q DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

23 MPCSXFER

23.1 Overview

This sample illustrates data transfer between the GPP and DSP through a basic mechanism of shared buffers with mutually exclusive access protection. It uses the MPCS component to provide the access protection for shared buffers allocated using the POOL component. Synchronization between the GPP and DSP-side application is done using the NOTIFY component.

On the DSP side, this application illustrates the use of TSK with the MPCS, POOL and NOTIFY components.

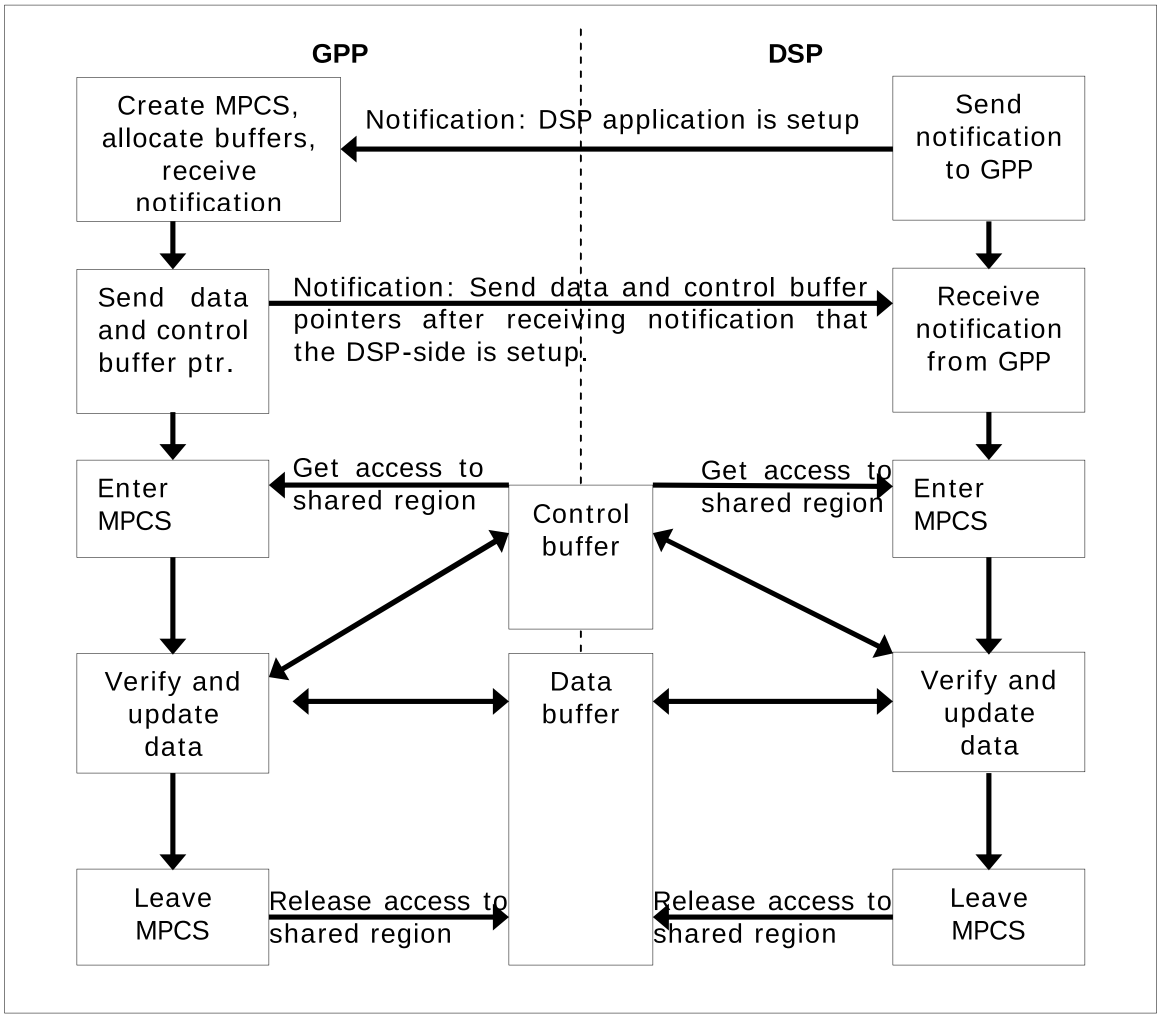

GPP
DSP
Create MPCS, allocate buffers, receive notification
Notification: DSP application is setup
Send notification to GPP
Send data and control buffer ptr.
Notification: Send data and control buffer pointers after receiving notification that the DSP-side is setup.
Receive notification from GPP
Enter MPCS
Get access to shared region
Control buffer
Get access to shared region
Enter MPCS
Verify and update data
Data buffer
Verify and update data
Leave MPCS
Release access to shared region
Release access to shared region
Leave MPCS

Figure 13. Data flow in the sample application – MCPSXFER

23.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP.
2. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
3. It opens the pool required for allocating the shared MPCS object, control and data buffer.
4. It then allocates the control and data buffers and translates their addresses to DSP address space to be sent to the DSP.
5. It creates the MPCS object to be used for protecting the control and data buffers and opens it.
6. It then initializes the control buffer contents and writes back the buffer through POOL to synchronize the buffer contents.
7. It creates a semaphore to be used to wait for notification from the DSP, and registers for notification of the event used by the application.
8. It then loads DSP executable (`mpcsxfer.out`) on the DSP

9. The client starts the execution on DSP.
10. It then waits on the semaphore. When the semaphore is posted, it indicates that the DSP application has completed its setup.
11. It then sends events to the DSP with the DSP addresses of the control and data buffers as payload.

EXECUTION

1. The client tries to get access to the shared control and data buffers by entering the MPCS used to provide mutually exclusive access to the buffers.
2. The contents of the control and data buffers are invalidated to synchronize their contents across processors.
3. If the control buffer contents indicate that the DSP had updated the control and data buffers, their contents are verified against the expected values. In this case, or if the control buffer indicates empty buffer, the contents of control and data buffers are modified to indicate that the GPP has updated them.
4. If the control buffer contents indicate that the GPP was the last to update them, the client sleeps for a few microseconds to simulate some other processing that can be done in this duration.
5. The contents of the control and data buffers are written back to synchronize their contents across processors.
6. Then the client releases control of the buffers by leaving the MPCS.
7. The client repeats the steps 1 through 6 for number of times specified by the user, or infinitely if so specified.

FINALIZATION

1. The client stops the DSP execution.
2. It unregisters the notification for events from the DSP and deletes the semaphore that was created to wait for the notification.
3. It closes the handle to the MPCS object and deletes it.
4. It then frees the pool memory that was allocated for the control and data buffers.
5. It closes the pool.
6. It detaches itself from DSP.
7. It destroys the PROC component.

23.1.2 On the DSP side

INITIALIZATION

1. The pool to be used for data transfer using the MPCS component is configured statically through the global `POOL_config` variable.
2. The client task `tskMpcsXfer` is created in the function `main ()`.
3. It opens the MPCS object with a specific name on the local processor.
4. It registers for notification of the event used by the application.
5. It then sends an event notification to the GPP to indicate that it has completed its setup.

6. It then waits for the event callback from the GPP-side to post the semaphore indicating receipt of the control buffer pointer. The second event callback indicates receipt of the data buffer pointer.

EXECUTION

1. The client tries to get access to the shared control and data buffers by entering the MPCS used to provide mutually exclusive access to the buffers.
2. The contents of the control and data buffers are invalidated to synchronize their contents across processors.
3. If the control buffer contents indicate that the GPP had updated the control and data buffers, their contents are verified against the expected values. In this case, or if the control buffer indicates empty buffer, the contents of control and data buffers are modified to indicate that the DSP has updated them.
4. If the control buffer contents indicate that the DSP was the last to update them, the client sleeps for a few microseconds to simulate some other processing that can be done in this duration.
5. The contents of the control and data buffers are written back to synchronize their contents across processors.
6. Then the client releases control of the buffers by leaving the MPCS.
7. The client repeats the steps 1 through 6 for number of times specified by the user, or infinitely if so specified.

FINALIZATION

1. The client unregisters the notification for events from the GPP.
2. It then closes the handle to the MPCS object.

23.1.3 Invoking the application

The MPCSXFER sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./mpcsxfer.out`
Buffer Size	`128`
Number of iterations	`10000`
DSP processor Id	`0`

- The sample can be executed for infinite iterations by specifying the number of iterations as 0.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

24 MESSAGE_MULTI

24.1 Overview

This sample illustrates the following concepts in DSP/BIOS™ LINK:

1. Multi-application usage of DSPLink.

2. Dynamic configuration
3. Opening multiple pools dynamically

This sample supports a maximum of MAX_APPS (defined by default as 16) application instances. Each application instance on the GPP exchanges messages with a corresponding task on the DSP.

On the DSP side, this application illustrates use of multiple TSKs and POOLs with MSGQ.

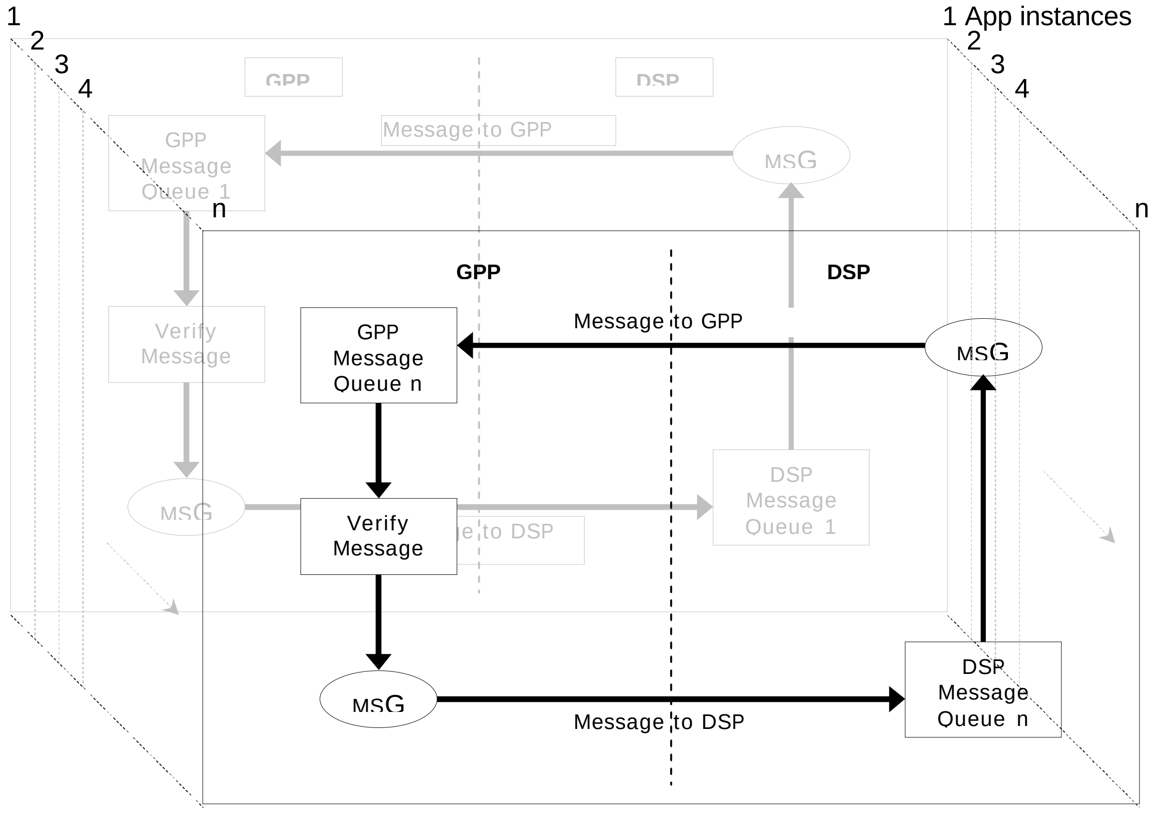


Figure 14. Message flow in the sample application – MESSAGE_MULTI

24.1.1 On the GPP side

The application code is almost the same as that of the message sample. However, it takes the application instance number as an additional parameter, and uses this value to decide the message queues that are used for message transfer on the GPP and DSP. The POOL ID used for the application is also the same as the application instance number.

INITIALIZATION

1. The application sets up DSPLink using PROC_setup. The dynamic configuration supports 17 pools (MAX_APPS + 1), and this configuration is passed to PROC_setup.
2. It then attaches to the DSP identified by DSP PROCESSOR IDENTIFIER.
3. It opens the common pool required for the Message Queue Transport (MQT). This pool is configured with the maximum number of control messages that may be required by the MQT.

4. It then opens a message queue identified by a specific name on the local processor. The name is generated using the application instance number as specified by the user as a parameter while executing the message_multi sample.
5. It then loads the DSP executable on the DSP
6. It starts the execution on the DSP.
7. It now also opens one pool required for allocating the messages that are transferred between the GPP and the DSP.
8. The client starts the execution on DSP.
9. It then opens the remote transport.
10. The application also creates a user-level semaphore, registers for notification for a specific event number and sends a notification to the DSP to inform the DSP task corresponding to the application number, to get prepared for message transfer.
11. Now it waits on the semaphore. When a notification is received from the DSP, it indicates that the DSP has completed setup and is ready for message transfer.
12. It then attempts to locate the queue opened on the DSP side. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not opened), it sleeps for some time and tries to locate the queue again.

EXECUTION

1. The client tries to get a message on the local queue. The get operation is specified to wait forever.
2. On receiving the message, it verifies the validity of contents of the received message.
3. It then sends the same message back to the DSP message queue.
4. Once the message is received, its contents are compared with the sequence number, which is incremented every time a get is successful.
5. The client repeats the steps 2 through 4 for number of times specified by the user, or infinitely if so specified.
6. For the case when finite number of iterations is specified, it frees the message that was received for the last get operation.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It closes the remote transport.
3. It un-registers the notification and deletes the user-level semaphore used for receiving notifications.
4. It stops the DSP execution.
5. It closes the local message queue.
6. It closes the pool specific to the application instance.
7. It also closes its handle to the common pool used for the Message Queue Transport.
8. It detaches itself from DSP.

9. It destroys the PROC component.

24.1.2 On the DSP side

INITIALIZATION

1. The common pool to be used for messaging is configured statically through the global `POOL_config` variable. The other MAX_APPS (16) pools that are opened dynamically as the GPP-side application instances are created, are also configured within the `POOL_config` variable with dummy values.
2. The MSGQ component is configured statically through the global `MSGQ_config` variable.
3. MAX_APPS tasks are created in the function `main ()`.
4. Each task registers for notification and waits for the notification to be received from the GPP. Notification is received by a specific task when its corresponding application instance is created on the GPP-side.
5. The task that becomes active now goes into its create phase.
6. It configures the application instance specific pool with actual POOL function table, parameters etc. and opens it.
7. It then sends a notification to the DSP that it has completed POOL setup.
8. It opens a message queue with a specific name on the local processor. The name is generated from the application instance number for this task.
9. It sets the above-opened queue as the error handler.
10. It then attempts to locate the queue opened on the GPP side. Locate is specified to wait forever. If the Locate call was unsuccessful (GPP queue still not opened), it sleeps for some time and tries to locate the queue again. The locate operation is synchronous.

EXECUTION

1. The task allocates a message from the pool.
2. It sends this message to the GPP message queue located earlier.
3. The task then tries to get a message on the local queue. The get operation is specified to wait forever.
4. These steps are repeated for number of iterations specified by the user, or infinitely, if so specified.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It resets the error handler which was set in the create phase.
3. It closes the local message queue.

24.1.3 Invoking the application

The message_multi sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable	`./messagemulti.out`

Number of iterations	`10000`
Application instance	`<1 -> 16>`
DSP processor Id	`0`

- For OS-specific instructions on execution of the message_multi sample, please refer to the install guide for the specific platform.
- The sample can be executed for infinite iterations by specifying the number of iterations as 0.
- DSP processor ID is optional argument for single DSP on Linux platforms. For multi dsp, this argument needs to be updated with DSP processor identifier. On PrOS, this argument needs to be passed with DSP processor identifier.

25 MESSAGE_MULTIDSP

25.1 Overview

This sample illustrates basic message transferring between GPP and Two DSPs. It transfers messages between a task running on GPP and task running on the DSP 0 and DSP 1.

This sample application is supported on the following configurations.

1. LINUXPC connected with two DM6437 over PCI.
2. J1 connected to DM6437 over VLYNQ interface.

On the DSP side, this application illustrates use of TSK with MSGQ.

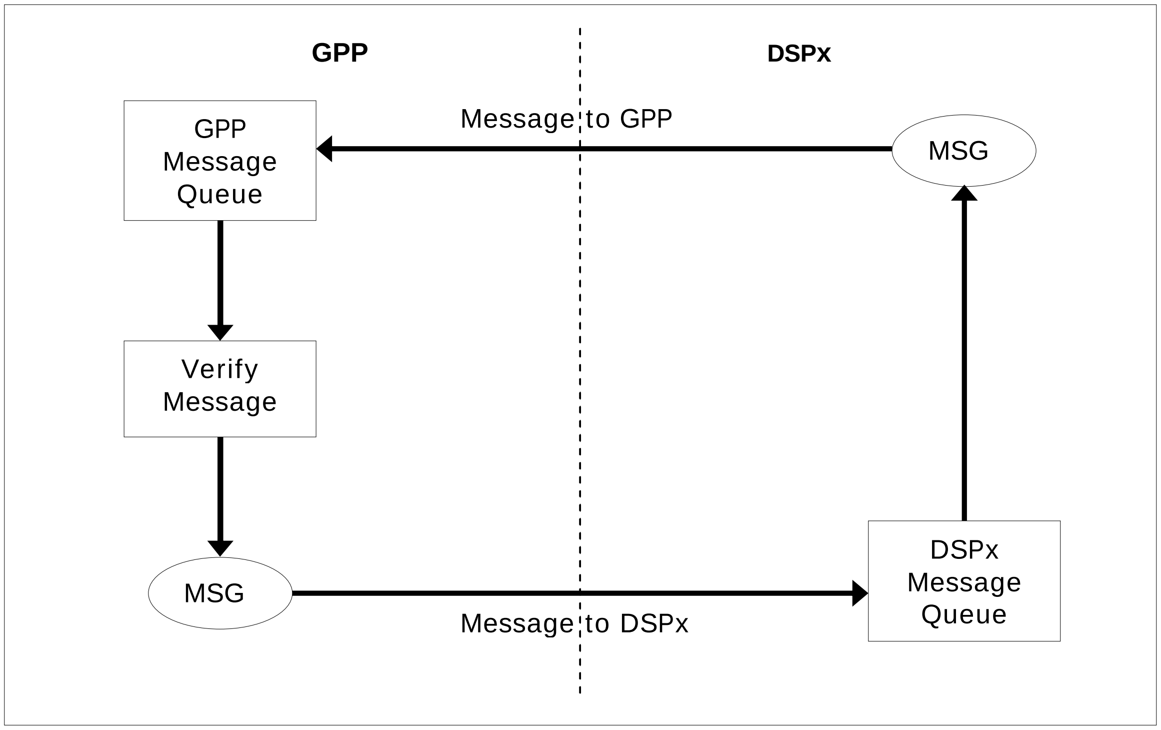


Figure 15. Message flow in the sample application – MESSAGE

In the above diagram DSPx indicates the DSP processors 0 and 1.

25.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSPs.
2. It then attaches to the DSPs identified by ID_DSP_PROCESSOR_0 and ID_DSP_PROCESSOR_1.
3. It opens the pools required for allocating the messages, depending on the DSP identifier and the physical link to be used for the data driver.
4. It then opens a message queue identified by a specific name on the local processor.
5. It sets the above-opened queue as the error handler.
6. It loads DSP executables (`message. out`) on to the DSPs.

   This application uses message.out as DSP executable. So rename message.out generated for DSP processor ID 0 to message_0.out and rename message.out generated for DSP processor ID 1 to message-1.out.
7. The client starts the execution on both DSPs.
8. It then opens the remote transports for both DSPs.
9. It then attempts to locate the queues opened on the both DSPs. Locate is specified to wait forever. If the Locate call was unsuccessful (DSP queue still not opened), it sleeps for some time and tries to locate the queue again.

EXECUTION

7. The client tries to get a message on the local queue. The get operation is specified to wait forever.
8. On receiving the message, it verifies the validity of contents of the received message.
9. It then sends the same message back to the DSP message queue from which DSP the message is received.
10. Once the message is received, its contents are compared with the sequence number corresponding to the DSP, which is incremented every time a get from the specific DSP is successful.
11. The client repeats the steps 2 through 4 for number of times specified by the user, or infinitely if so specified.
12. For the case when finite number of iterations is specified, it frees the message that was received for the last get operation.

FINALIZATION

9. The client releases the remote message queues created on both DSPs.
10. It closes the remote transports.
11. It stops execution on both the DSPs.
12. It resets the error handler which was set in the create phase.
13. It closes the local message queue.

14. It closes the pools.
15. It detaches itself from ID_DSP_PROCESSOR_0 and ID_DSP_PROCESSOR_1 .
16. It destroys the PROC component.

25.1.2 On the DSP side

25.1.2.1 Using TSK with MSGQ

INITIALIZATION

7. The pool to be used for messaging is configured statically through the global `POOL_config` variable.
8. The MSGQ component is configured statically through the global `MSGQ_config` variable.
9. The client task tskMessage is created in the function `main ()`.
10. It opens a message queue with a specific name on the local processor.
11. It sets the above-opened queue as the error handler.
12. It then attempts to locate the queue opened on the GPP side. Locate is specified to wait forever. If the Locate call was unsuccessful (GPP queue still not opened), it sleeps for some time and tries to locate the queue again. The locate operation is synchronous.

EXECUTION

7. The task allocates a message from the pool.
8. It sends this message to the GPP message queue located earlier.
9. The task then tries to get a message on the local queue. The get operation is specified to wait forever.
10. These steps are repeated for number of iterations specified by the user, or infinitely, if so specified.

FINALIZATION

4. The client releases the remote message queue on the DSP side.
5. It resets the error handler which was set in the create phase.
6. It closes the local message queue.

25.1.3 Invoking the application

The message sample takes the following parameters:

Parameter	Example Value
Absolute path of DSP executable to be run on DSP processor 0	`./message_0.out`
Absolute path of DSP executable to be run on DSP processor 1	`./message_1.out`
Number of iterations	`10000`

q The sample can be executed for infinite iterations by specifying the number of iterations as 0.

- In case of PROS "Due to Dynamic configuration change during execution of the MESSAGE_MULTI sample, MESSAGE_MULTIDSP sample should be run before MESSAGE_MULTI sample".

TESTSUITE

26 Overview

The information regarding the test suite is provided in the user guide of the DSP/BIOS LINK test suite product.

APPENDIX

27 Issue reclaim model

The issue reclaim model is graphically represented in the diagram below:

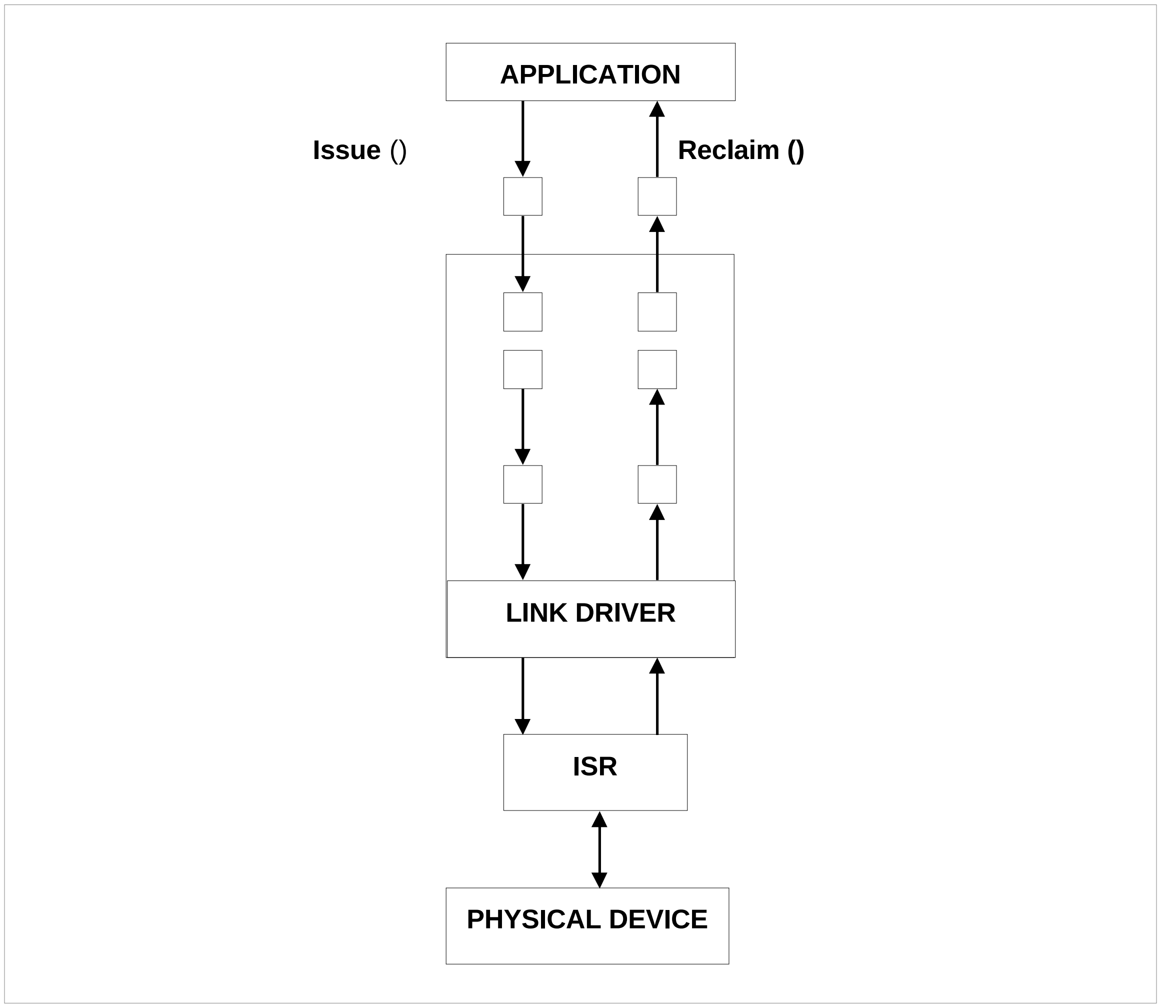


Figure 16. Issue Reclaim Model

The steps for data transfer with issue reclaim model may be summarized below:

1. Open a channel with defined buffer size & direction.
2. Issue a buffer for IO on the specified channel
    - Empty buffer for receiving data
    - Filled buffer for sending data
3. Attempt to reclaim the buffer. Reclaim will block until the IO operation completes or a timeout occurs.
    - This wait can be postponed to a later point in time for asynchronous IO.
4. A client must reclaim all the buffers issued to a channel.

28 Adding application or platform specific capabilities

As we have seen in earlier sections, DSP/BIOS™ LINK exports a basic API for processor control, data transfer and messaging.

However, depending upon the application and the target platform, it may be desired to extend the functionality of DSP/BIOS™ LINK. Some such capabilities are:

§ Leveraging power management features of DSP.

§ Initializing auxiliary hardware devices on the platform.

The APIs `PROC_control ()` and `CHNL_control ()` provide hooks to perform such control operations.

The execution flow for both these APIs is shown below:

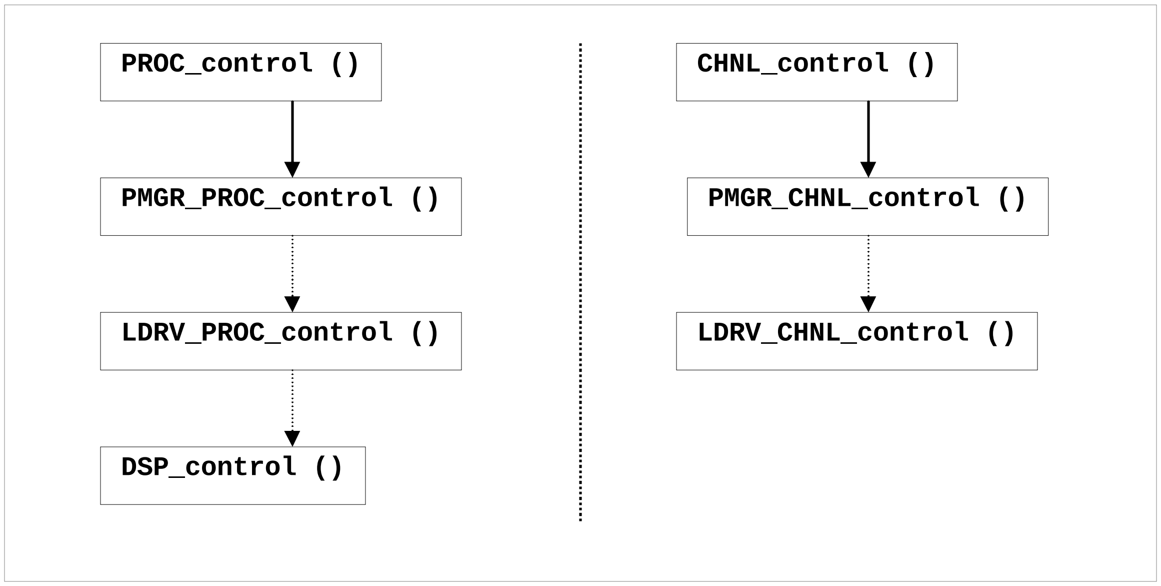


Figure 17. Execution flow: PROC_control () and CHNL_control ()

The arguments to both these APIs include a command and optional argument(s) for the specified command. For more details on syntax of these APIs refer to Source Reference Guide.

Depending upon the specified command, processing can be done at all (or any) of the stages shown in the diagram above.

In the default implementation, functions `PMGR_PROC_control ()` and `PMGR_CHNL_control ()` return status value `DSP_ENOTIMPL`.

These functions can, however, easily be modified to reach the functions `DSP_control ()` and `LDRV_CHNL_control ()` as shown by dotted arrows in the diagram above.

29 Passing arguments to DSP side application

Arguments to the DSP executable's main () can be passed through the API PROC_Load (). This API fills the ".args" buffer before writing it to DSP's memory spaces. This section is used by BIOS to pass arguments to main ().

The ".args" section is created during compilation of the DSP executable. To avoid overwriting areas outside this section the compiler needs to be instructed to create a large enough section based on the arguments that have to be passed to the DSP. The following sections describe the changes that are required to achieve this.

29.1 Passing arguments from the GPP side

The following code illustrates the method to pass arguments using the PROC_Load () API.

```
Uint32  argc = 0 ;
Char8 * argv [NUM_ARGS] ;

argc = NUM_ARGS ;

argv [0] = arg_string_1 ;
argv [1] = arg_string_2 ;
...
...
argv [NUM_ARGS - 1] = arg_string_end ;

status = PROC_Load (dspId, dspExecutableFileName, argc, argv) ;
```

29.2 Receiving arguments on the DSP side

The following line needs to be added to the tcf file to create the ".args" section of the specified size (in bytes).

```
prog.module("MEM").ARGSSIZE = <number of bytes> ;
```

30 Debugging Applications

30.1 On the GPP side

30.1.1 Trace statements

DSP/BIOS™ LINK displays the trace of its execution by conditionally printing the function entry and exit from all functions. This information can be used for debugging applications as well as for understanding DSP/BIOS™ LINK.

The following macros provided in the TRC subcomponent of OSAL allow selection of trace prints:

```
TRC_ENABLE
TRC_DISABLE
TRC_SETSEVERITY
```

The selection can be based on the severity of the message being displayed as well as the subcomponent origin of the message.

TRC_ENABLE and TRC_DISABLE macros allow selection of the trace prints based on component and subcomponent. These macros take identifiers for sub-components whose trace is required. See `signature.h` for the definition of these identifiers.

The TRC_SET_SEVERITY interface allows selection of the severity level of trace statements to print. The levels are defined from TRC_ENTER to TRC_LEVEL7, where TRC_LEVEL7 is the highest. The level TRC_ENTER and alternate level TRC_LEAVE is used to print function entry and exit from all the functions in DSP/BIOS™ LINK.

These macros need to be called upon module initialization to setup necessary data structures. (See the function `DRV_InitializeModule ()` in `drv_pmgr.c`).

Some examples below explain the usage:

```
TRC_ENABLE (ID_PMGR_PROC) ;
TRC_SET_SEVERITY (TRC_ENTER) ;
```

These statements enable prints from PMGR_PROC with the lowest severity allowing all trace prints to display.

```
TRC_ENABLE (ID_LDRV_ALL) ;
TRC_SET_SEVERITY (TRC_LEVEL2) ;
```

These statements enable prints from all files of LDRV with a severity allowing LEVEL2 and above trace prints to display.

```
TRC_ENABLE (ID_LDRV_POOL_ALL) ;
TRC_SET_SEVERITY (TRC_LEVEL2) ;
```

These statements enable prints from all files of the POOL sub-component of LDRV.

```
TRC_ENABLE (ID_OSAL_ALL) ;
TRC_DISABLE (ID_OSAL_MEM) ;
TRC_SET_SEVERITY (TRC_ENTER) ;
```

These statements enable from all subcomponents of OSAL except the MEM subcomponent with the lowest severity level allowing all trace statements to display.

30.1.2 Profiling

DSP/BIOS™ LINK contains code to keep track of various pieces of instrumentation information. This source code can be compiled out, and so does not interfere with the regular code path and does not impact the code execution negatively.

The GPP-side build configuration allows different levels of profiling to be set. Please refer to the section on build configuration for details.

The profiling levels are:

- No profiling: When profiling is not enabled, no instrumentation information is maintained by DSP/BIOS™ LINK.
- Basic profiling: When this profiling level is selected, standard instrumentation information maintained by DSP/BIOS™ LINK. This information includes:
  - Number of interrupts exchanged by GPP and DSP
  - Number of bytes read and written to DSP memory space by GPP
  - Amount of data exchanged
  - Channels that are currently open
  - Buffers that are currently queued.
  - Messages that have been transferred.
  - Messages that are currently queued, etc.
- Detailed profiling: When this profiling level is selected, detailed instrumentation is maintained. This includes storing the first few bytes of data exchanged on a channel. Enabling detailed profiling automatically enables standard profiling.

30.1.3 SET_FAILURE_REASON

DSP/BIOS™ LINK uses a mechanism to record an exception that may occur during execution. Such an error/exception is stored in a structure called ErrReason. This structure contains the fields:

Type	Name	Description
Bool	IsSet	Set to TRUE when a failure is recorded
Int32	FileId	Identifier for file in which the error occurred. File signature.h contains the list of file identifiers used in source code.
Int32	LineNum	Line number on which the error was recorded
DSP_STATUS	status	The error status.

The macro SET_FAILURE_REASON is used to record these failures wherever such failures are expected. However, only the first failure is recorded. This is especially helpful since the first failure can then trigger a chain of other errors making traceability difficult. A typical example of the usage of this macro is:

```
status = LDRV_DRV_Initialize (dspId) ;
if (DSP_FAILED (status)) {
    SET_FAILURE_REASON ;
}
```

30.2 On the DSP side

The DSP side can be debugged using CCS with the QuickTurn platform.

For debugging, the DSP must be halted in a known state. The following two methods can be used for achieving this:

30.3 Stopping execution in main

Execution of DSP can be suspended in 'main ()' by putting an infinite loop at the beginning of the function. However a simple 'while (1)' loop cannot be used as the compiler optimizes away the code after the while loop as that code becomes unreachable. To sneak through this optimization the following while loop can be used:

```
{
    volatile Int i = 1 ;
    while (i) ;
}
```

With this change in place on the DSP side application, the follow these steps to be to DEBUG the DSP application.

Follow these steps to break from the loop:

1. After PROC_Start () is successful on GPP, halt the DSP. The DSP will be executing in the while loop.
2. Load the symbols of the DSP executable using CCS.
3. Use 'Set PC to Cursor' to break from the loop.

The DSP application can now be debugged by placing breakpoints as required or single stepping through the code.

30.4 SET_FAILURE_REASON

The SET_FAILURE_REASON macro included with the DSP side sources can be used to log failure or optionally stop execution upon failure in DEBUG builds. If the macro DSPLINK_FAILURE_STOP is defined through compile flags, a failure in execution causing invocation of the SET_FAILURE_REASON macro causes the DSP execution to halt at the failure location.

The mechanism of halt at the failure location also depends on the USE_CCS_BREAKPOINT macro for the 55x-based platforms. If it is defined, this macro puts a software-breakpoint at the location where it is called from, allowing the debugging to continue from the location.

When USE_CCS_BREAKPOINT is not defined or for non-55x based platforms, this macro expands to the infinite loop mentioned in the previous section. The steps mentioned in the previous section can be used to proceed with debugging.

31 Configuring DSP/BIOS™ LINK

31.1 Dynamic configuration

The static configuration of DSPLink was earlier achieved through a textual configuration file (CFG_<PLATFORM>.TXT), which was processed during the build step of DSPLink, to generate configuration header and source files for both the GPP and DSP-sides of DSPLink. These generated files were compiled along-with the DSPLink GPP-side kernel module and DSP-side dsplink library.

From release 1.40.03, the textual configuration file has been replaced by a pre-defined "C" source file with pre-defined configuration values defined within a fixed structure format. This shall be compiled with the DSPLink user library by default.

The Dynamic Configuration of DSPLink is achieved through configuration items made available to DSPLink from the user-side on the GPP-side only. The GPP-side kernel module and DSP-side library do not need to be rebuilt.

31.1.1 Change in configuration

PROC_setup () api has been modified to optionally take a pointer to a configuration structure in the same format as the provided configuration source file. If a valid pointer is provided, the configuration values provided by the application are used. If none is provided, the default configuration is used. This ensures backward compatibility of existing applications.

No configuration source or header files shall be generated, resulting in the GPP-side kernel module and DSP-side DSPLink library not requiring to be rebuilt.

31.1.2 How to use dynamic configuration?

§ The file $(DSPLINK)\config\all\CFG_<Platform>.c provides the default configuration which dsplink uses. To change your configuration, you can make a copy of this file in your application specific code.

- Modify the <application_path>\CFG_<platform> file to adapt to your application specific needs.
- Rename LINKCFG_config to application specific name say LINKCFG_appConfig to avoid build conflict with the default configuration.
- Add <application_path>\CFG_<platform> file to the list of files to be compiled in your application.
- Modify your application code to have an extern declaration of LINKCFG_appConfig structure and pass it as a parameter to PROC_setup

This enables that any change in the configuration causes only rebuilding of application and not the dsplink code base.

31.2 GPP side

The need for configuration may arise due to any of the following considerations:

§ Porting to new platform/ physical link

§ Application specific requirements on the existing link driver

A pre-defined "C" source file for dsp configuration is provided with configuration values defined within a fixed structure format. This is compiled with the DSPLink user library by default.

The configuration file follows a specific naming convention:

CFG + <Name of platform> + <Name of variant, if any> + .c

e.g. CFG_Davinci.c, CFG_Davinci_DM6467.c, CFG_DM642_PCI.c, …

A predefined c source files are provide with configuration values defined in a fixed structure format for the GPP platform and for GPP OS.

The configuration file follows a specific naming convention:

CFG + <GPPARCH> + .c

CFG + <GPPOS>+.c

31.2.1 GPP

`NAME`	This field specifies the name of the GPP in the system. It is used for information purposes only.
`MAXMSGQS`	This field specifies the maximum number of message queues (MSGQs) that can be opened on the local processor.
`MAXCHNLQUEUE`	This field specifies the maximum queue length for all channels.
`POOLTABLEID`	This field specifies the ID of the POOL table (-1 if not needed).
`NUMPOOLS`	This field specifies the number of pools that are available for use by the driver.

31.2.2 DSP

`NAME`	This field specifies the name of the DSP being configured. It is used for information purposes only.
`ARCHITECTURE`	This field specifies the architecture of the DSP. This field takes enumerated values from the structure DspArch defined in the file `dsplink.h`.
`LOADERNAME`	This field specifies the name of the DSP executable loader.
`AUTOSTART`	This field specifies whether the DSP can be auto-started. This field is currently not used.
`EXECUTABLE`	This field specifies the default executable for the DSP to be loaded during autostart. This field is currently not used.
`DOPOWERCTRL`	This field indicates whether the power-control for the DSP is to be done by DSPLink.
`RESUMEADDR`	This field specifies the resume address for the DSP.
`RESETVECTOR`	This field specifies the address of the reset vector of the DSP.
`RESETCODESIZE`	This field specifies the size of code at the DSP reset vector.
`MADUSIZE`	This field specifies the size of the Minimum Addressable Data Unit (MADU) on the DSP in bytes.
`ENDIAN`	This field specifies the default endianism of the DSP.

WORDSWAP	This field specifies whether the words must be swapped when writing to memory.
MEMTABLEID	This field specifies the index of MEMTABLE to use for the DSP. If none are required, this field can take the value –1.
MEMENTRIES	This field specifies the number of memory information entries for the DSP. In case of a DSP having an MMU, this information may map its the MMU entries. In case of DSPs where this is not the case, this field may give information about any memory accessible to both the GPP and the DSP.
LINKDRVID	This field specifies the index of the LINKDRV section to use for accessing this DSP.
ARGUMENT1	This field platform specific argument1.
ARGUMENT2	This field platform specific argument2.
ARGUMENT3	This field platform specific argument3.
ARGUMENT4	This field platform specific argument4.

31.2.3 MEM Tables

This specifies the MEM entries for each DSP. There is one MEMTABLE section for each DSP. However, each MEMTABLE may contain more than one entry depending on the number of MEM entries desired for the DSP application.

ENTRY	ID of the entry in the MEMTABLE.
NAME	This field specifies a short abbreviation of the name of the entry in the MEMTABLE. The abbreviation is used for generating the constants related to this entry in the generated configuration file(s).
ADDRPHYS	Physical address of the memory indicated by this entry.
ADDRDSPVIRTUAL	Virtual address of the memory as seen by the DSP
ADDRGPPVIRTUAL	Virtual address of the memory as seen by the GPP
SIZE	Size of the memory indicated by this entry.
SHARED	Indicates whether the memory area is shared?
SYNCD	Indicates whether the memory area is synchronized?

31.2.4 Pools

This specifies the pools used in the system. A Pool is used for allocating the buffers and messages to be used for data transfer and messaging respectively. The POOL component provides a standard interface for configuration of the memory pools in the system, which may be differently implemented based on the requirements of the physical link and the system.

NAME	This field specifies the name of the Pool. It is used for information purposes only.
MEMENTRY	This field specifies the ID of the MEM entry used by the POOL. If the pool does not use any MEM entries, this field can take the value –1.

POOLSIZE	This field specifies the maximum size of the memory used by the pool. This field can take the value –1 if there is no configuration limit set on the maximum size supported for the pool.
IPSID	This field specifies the ID of the IPS used (if any)
IPSEVENTNO	This field specifies the IPS event number associated with this POOL (if any).
POOLMEMENTRY	Pool memory region section ID
ARGUMENT1	This field specifies argument 1 to the pool. The significance of this argument depends on the implementation of the pool.
ARGUMENT2	This field specifies argument 2 to the pool. The significance of this argument depends on the implementation of the pool.

31.2.5 Link Drivers

This section specifies the attributes of each of the physical link drivers to be used by DSP/BIOS™ LINK. There is one LINKDRV section for each DSP. The link driver may use one or more Inter-Processor-Signaling (IPS) component(s) based on the physical links supported between the GPP and the DSP.

NAME	This field specifies the name of the link driver. It is used for information purposes only.
HSHKPOLLCOUNT	This field specifies the poll value for which handshake waits (-1 if infinite).
MEMENTRY	This field specifies the ID of the MEM entry used by the link driver. If the link driver does not use any MEM entries, this field can take the value –1.
IPSTABLEID	This field specifies the ID of the IPS table used.
IPSENTRIES	This field specifies the number of IPS supported.
POOLTABLEID	This field specifies the pool id for allocating buffers.
NUMPOOLS	This field specifies the number of POOLs supported.
DATATABLEID	This field specifies the ID of the data driver table.
QUEUELENGTH	This field specifies the number of buffers that can be simultaneously queued for transfer by the DSP/BIOS™ LINK data transfer driver.
NUMDATADRV	This field specifies the number of data drivers supported.
MQTID	This field specifies the ID of the MQT.
RINGIOTABLEID	This field specifies the RingIO Table Id used for this DSP.
MPLISTTABLEID	This field specifies the MpList Table Id used for this DSP.
MPCSTABLEID	This field specifies the MPCS Table ID used for this DSP.

31.2.6 IPS Tables

This specifies the Inter-Processor-Signaling components used by each link driver. There is one `IPSTABLE` section for each link driver. However, each `IPSTABLE` may contain more than one entry depending on the number of different types of physical links supported for the DSP.

`NAME`	This field specifies the name of the IPS component. It is used for information purposes only.
`NUMIPSEVENTS`	This field specifies the number of IPS events to be supported.
`MEMENTRY`	This field specifies the ID of the MEM entry used by the IPS. If the IPS does not use any MEM entries, this field can take the value –1.
`GPPINTID`	This field specifies the interrupt no. to used by the IPS on GPP-side.
`DSPINTID`	This field specifies the interrupt no. to used by the IPS on DSP-side.
`DSPINTVECTORID`	This field specifies the interrupt vector no. to used by the IPS on DSP-side. (-1 if uni-directional to GPP)
`ARGUMENT1`	This field specifies argument 1 to the IPS. The significance of this argument depends on the implementation of the IPS.
`ARGUMENT2`	This field specifies argument 2 to the IPS. The significance of this argument depends on the implementation of the IPS.

31.2.7 MQTs

This specifies the Message Queue Transport (MQT) components used by each DSP. There can be only one `MQT` defined at a time for each DSP.

`NAME`	This field specifies the name of the MQT. It is used for information purposes only.
`MEMENTRY`	This field specifies the ID of the MEM entry used by the MQT. If the MQT does not use any MEM entries, this field can take the value –1.
`MAXMSGSIZE`	This field specifies the maximum size of messages supported by this MQT. If the MQT does not impose any limitation on the size of messages that can be transferred by it, this field can take the value –1.
`IPSID`	This field specifies the ID of the IPS used (if any)
`IPSEVENTNO`	This field specifies the event number associated with this MQT.
`ARGUMENT1`	This field specifies argument 1 to the MQT. The significance of this argument depends on the implementation of the MQT.
`ARGUMENT2`	This field specifies argument 2 to the MQT.

The significance of this argument depends on the implementation of the MQT.

31.2.8 DATA Tables

This specifies the Data Drivers used by each DSP. There is one `DATADRV` table section for each DSP. However, each `DATADRV` may contain more than one entry depending on the number of physical links supported for data transfer by the DSP.

`NAME`	This field specifies the name of the data driver. It is used for information purposes only.
`ABBR`	This field specifies a short abbreviation for the data driver name. This field is used for generating the constants related to this entry in the generated configuration file(s).
`BASECHANNELID`	This field specifies the base logical channel ID for this data driver. The channel IDs supported by this driver range between the `BASECHANNELID` and the (`BASECHANNELID + NUMCHANNELS – 1`) for this data driver.
`NUMCHANNELS`	This field specifies the number of logical channels supported by this data driver.
`MAXBUFSIZE`	This field specifies the maximum size of buffers supported by this data driver. If the data driver does not impose any limitation on the size of buffers that can be transferred by it, this field can take the value –1.
`INTERFACE`	This field specifies the address of the interface function pointer table for using this data driver.
`MEMENTRY`	This field specifies the ID of the MEM entry used by the data driver. If the data driver does not use any MEM entries, this field can take the value –1.
`POOLID`	This field specifies the ID of the pool used for allocating buffers that are transferred between the GPP and the DSP using this data driver.
`SIZE`	This field specifies the size of the control area required by the data driver component within the MEM entry to which it refers.
`QUEUEPERCHANNEL`	This field specifies the number of queued buffers per data channel.
`IRPSIZE`	This field specifies the size of each IO Request Packet for data transfer.
`IPSID`	This field specifies the ID of the IPS used (if any)
`IPSEVENTNO`	This field specifies the event number associated with this data streaming driver.
`ARGUMENT1`	This field specifies argument 1 to the data driver. The significance of this argument depends on the implementation of the data driver.

`ARGUMENT2`	This field specifies argument 2 to the data driver. The significance of this argument depends on the implementation of the data driver.

31.2.9 RINGIOTABLE

This specifies the configuration for RingIO component.

`NAME`	This field specifies the name of this RingIO table.
`MEMENTRY`	This field specifies the MEMENTRY to be used for placing this RingIO table.
`NUMENTRIES`	This field specifies the number of RingIO instances to be supported in the system.
`IPSID`	This field specifies the ID of the IPS used.
`IPSEVENTNO`	This field specifies the event number associated with this RingIO driver.

31.2.10 MPCSTABLE

This specifies the configuration for MPCS component.

`NAME`	This field specifies the name of this MPCS table.
`MEMENTRY`	This field specifies the MEMENTRY to be used for placing this MPCS table.
`NUMENTRIES`	This field specifies the number of MPCS instances to be supported in the system.
`IPSID`	This field specifies the ID of the IPS used (if any)
`IPSEVENTNO`	This field specifies the event number associated with this MPCS driver (if any).

31.2.11 MPLISTTABLE

This specifies the configuration for MPLIST component.

`NAME`	This field specifies the name of this MPLIST table.
`MEMENTRY`	This field specifies the MEMENTRY to be used for placing this MPLIST table.
`NUMENTRIES`	This field specifies the number of MPLIST instances to be supported in the system.
`IPSID`	This field specifies the ID of the IPS used (if any)
`IPSEVENTNO`	This field specifies the event number associated with this MPLIST driver (if any).

31.2.12 LOG

This specifies the configuration for logging instrumentation information.

`GDMSGQPUT`	This field indicates whether logging is to be enabled for: GPP->DSP MSG Transfer - MSGQ_Put call.
`GDMSGQSENDINT`	This field indicates whether logging is to be enabled

	for: GPP->DSP MSG Transfer - GPP sends interrupt.
GDMSGQISR	This field indicates whether logging is to be enabled for: GPP->DSP MSG Transfer - DSP receives interrupt.
GDMSGQQUE	This field indicates whether logging is to be enabled for: GPP->DSP MSG Transfer - Message queued at DSP.
DGMSGQPUT	This field indicates whether logging is to be enabled for: DSP->GPP MSG Transfer - MSGQ_Put call.
DGMSGQSENDINT	This field indicates whether logging is to be enabled for: DSP->GPP MSG Transfer - DSP sends interrupt.
DGMSGQISR	This field indicates whether logging is to be enabled for: DSP->GPP MSG Transfer - GPP receives interrupt.
DGMSGQQUE	This field indicates whether logging is to be enabled for: DSP->GPP MSG Transfer - Message queued at GPP.
GDCHNLISSUESTART	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Entering inside ISSUE call.
GDCHNLISSUEQUE	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Buffer is queued in internal structure on GPP.
GDCHNLISSUECOMPL	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - ISSUE call completed.
GDCHNLXFERSTART	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Initiating a buffer transfer by GPP.
GDCHNLXFERPROCESS	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Actual transfer of buffer is going to take place.
GDCHNLXFERCOMPL	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Buffer transfer is complete.
GDCHNLRECLSTART	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Entring RECLAIM call.
GDCHNLRECLPEND	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - Wait on a semaphore.
GDCHNLRECLPOST	This field indicates whether logging is to be enabled for: GPP->DSP CHNL Transfer - posting the Semaphore.
GDCHNLRECLCOMPL	This field indicates whether logging is to be enabled

	for: GPP->DSP CHNL Transfer - RECLAIM call completed.
`DGCHNLISSUEQUE`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - Buffer is queued in internal structure on DSP.
`DGCHNLXFERSTART`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - Initiating a buffer transfer by DSP.
`DGCHNLXFERPROCESSING`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - Actual transfer of buffer is going to take place.
`DGCHNLXFERCOMPLETE`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - Buffer transfer is complete.
`DGCHNLRECLPEND`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - Wait on a semaphore.
`DGCHNLRECLPOST`	This field indicates whether logging is to be enabled for: DSP->GPP CHNL Transfer - posting the Semaphore.
`MSGIDRANGESTART`	This field specifies the lower limit of the message ID range that must be ignored during instrumentation.
`MSGIDRANGEEND`	This field specifies the upper limit of the message ID range that must be ignored during instrumentation.

31.2.13 GPPOBJECT

This structure specifies the configuration for the gpp object.

`NAME`	Name of the GPP.
`MAXMSGQS`	Maximum MSGQs that can be opened
`MAXCHNLQUEUE`	Maximum Queue Length for all channels
`POOLTABLEID`	ID of the POOL table (-1 if not needed)
`NUMPOOLS`	Number of POOLs supported.
`GPPOSOBJECT`	Pointer to GPP OS object

31.2.14 GPPOSOBJECT

This structure specifies the GPP OS specific configuration.

`HANDLESIGNALS`	Should signals be handled for cleanup (Boolean flag)
`NUMSIGNALS`	Number of signals to be handled
`SIGNUMARRAY`	Pointer to the array of signals to be handled

31.3 DSP side

DSP side configuration is done through the TCF file. A base TCI include file is provided with DSP/BIOS™ LINK for each supported platform.

For example, for the DaVinci platform, the TCI file provided is `dsplink-dm6446gem-base.tci`. This file provides the basic definition of the memory regions and other configuration as required for using DSP/BIOS™ LINK. This file must be included for all scalability configurations of DSP/BIOS™ LINK.

The anatomy of a typical application TCF file is shown below:

`utils.importFile("dsplink-dm6446gem-base.tci");`	Line 1
`utils.importFile("dsplink-davinci-iom.tci");`	Line 2
`utils.importFile("dsplink-davinci-dio.tci");`	Line 3
`utils.importFile("<app>.tci");`	Line 4
`Platform specific application configuration goes here`	Line 5 - Line N
`prog.gen();`	Line N + 1

Here is the description of each statement listed below:

Line 1	...	Loads the base configuration file for DSP/BIOS™ LINK for DaVinci.
Line 2	...	Loads the configuration file for DSP/BIOS™ LINK for DaVinci containing the IOM driver used for data transfer.
Line 3	...	Loads the configuration file for DSP/BIOS™ LINK for DaVinci containing the DIO class driver used for data transfer. Here, the statement indicates need for the DIO class driver. The DIO class driver is required to use the SIO interface.
Line 4	...	This statement loads the platform independent application specific configuration.
Line 5 – Line N	...	Platform specific application configuration goes here
Line N + 1	...	This statement is an instruction to generate the CDB file.

q Do not change the DSP/BIOS™ LINK specific 'tci' files for any application specific configuration.

32 Understanding The MAKE System

32.1 Overview

This 'make' system is compatible with the GNU make utility. It also uses PERL for small tasks that cannot be accomplished with the GNU make.

This make system provides a single interface to build sources for all GPP side operating systems and platforms as well as DSP side applications developed for DSP/BIOS™. The make system can be used on Windows to build GPP side as well as DSP side libraries, applications, tests etc.

The make can be invoked from shell with following command:

```
gmake [TARGET] [VERBOSE=1]
```

The TARGET can be one of the following:

`all`	Make all build variants. [Default]
`debug`	Build DEBUG variant.
`release`	Build RELEASE variant.
`clean`	Delete all intermediate and output files.
`clobber`	Delete all directories created during build process.
`targets`	Build the target (.o/.ko) file from the intermediate object files.
`exports`	Export the specified file to a pre-defined location.

To build a component successfully the developer needs to be aware of the following four files:

§ MAKEFILE

§ COMPONENT

§ SOURCES

§ DIRS

32.1.1 MAKEFILE

Each component requires a make file. This MAKEFILE is standard for all the modules. User is not required to change this file. A warning to this effect is shown in these files.

A sample MAKEFILE file is shown below:

```
#   ============================================================================
#   @file   MAKEFILE
#
#   @path   $(DSPLINK)/dsp/src
#
#   @desc   This file is a standard interface to the make scripts.
#           Usually no change is required in this file.
#
#           To change the way a component is built, edit the file
#           COMPONENT situated under the directory $(DSPOS).
```

```
#
#   @ver    01.64
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ============================================================================


#   ============================================================================
#   Set the device type (GPP/DSP)
#   ============================================================================

export DEVICETYPE := DSP


#   ============================================================================
#   Get the directory separator used on the development host.
#   ============================================================================

ifneq ("$(ComSpec)", "")
    ifneq ("$(CYGWIN)", "")
        DIRSEP ?=/
    else
        DIRSEP ?=\\
    endif
else
    DIRSEP ?= /
endif

#   ============================================================================
#   Start the build process
#   ============================================================================

include $(DSPLINK)$(DIRSEP)make$(DIRSEP)start.mk
```

- The variable DEVICETYPE represents the processor for which the build is to be invoked. It can be set to either 'DSP' (for DSP side builds) or to 'GPP' (for GPP side builds).
- The variable DIRSEP is represents the directory separator on the development host. The variable is used in rest of the MAKE system, making it OS independent.
- You may use the directory separator specific to operating system on development host, if the build environment is specific to the operating system OR you will not be using another operating system on the development host.

32.1.2 COMPONENT

There is one COMPONENT file for every component in the OS specific folder for the component. This file affects the compilation and linking of the component by specifying OS specific attributes during the build process.

Following variables are defined in this file:

COMP_NAME	The name of the component.
COMP_PATH	Path of the component base directory.
COMP_TYPE	Type of the component. This can be either of LIB (library), DRV (driver) or EXE (executable) for the GPP-side. for the DSP-side, this can be of types ARC (archive) or EXE (executable).
COMP_TARGET	Name of the target file generated when the component is built.
COMP_MEMSPACE	Memory space in which the component is built, It could be USER or KRNL.
EXP_HEADERS	Headers files exported from the component.
USR_CC_FLAGS	Compiler flags specific to the component.
USR_CC_DEFNS	Compiler definitions specific to the component.
USR_LD_FLAGS	Additional linker options specific to the component.
STD_LIBS	Standard OS libraries to be linked into the component when it is built.
USR_LIBS	User specific libraries to be linked into the component when it is built.
EXP_TARGETS	Target file exported from the component.

For DSP side applications an additional parameter is defined:

COMP_MAP_FILE	MAP file for the component.

A sample COMPONENT file is listed below:

```
#   ============================================================================
#   @file   COMPONENT
#
#   @path   $(DSPLINK)/gpp/src/osal/PrOS
#
#   @desc   This file contains information to build a component.
#
#   @ver    01.64
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ============================================================================
#   ============================================================================
#   Generic information about the component
#   ============================================================================

COMP_NAME       := OSAL
COMP_PATH       := $(GPPROOT)$(DIRSEP)src$(DIRSEP)osal
COMP_TYPE       := LIB
COMP_TARGET     := OSAL.LIB


#   ============================================================================
#   Header file(s) exported from this component
#   ============================================================================

EXP_HEADERS     := \
    dpc.h       \
    isr.h       \
    kfile.h     \
    mem.h       \
    prcs.h      \
    sync.h      \
    trc.h       \
    cfg.h       \
    print.h     \
    osal.h      \
    $(GPPOS)$(DIRSEP)mem_os.h


#   ============================================================================
#   User specified additional command line options for the compiler
#   ============================================================================

USR_CC_FLAGS    :=

USR_CC_DEFNS    := -DTRACE_KERNEL


#   ============================================================================
#   User specified additional command line options for the linker
#   ============================================================================

USR_LD_FLAGS    :=


#   ============================================================================
#   Standard libraries of GPP OS required during linking
#   ============================================================================
STD_LIBS        :=


#   ============================================================================
#   User specified libraries required during linking
#   ============================================================================

USR_LIBS        :=


#   ============================================================================
#   Target file(s) exported from this module
#   ============================================================================

EXP_TARGETS     :=
```

32.1.3 SOURCES

This file provides a list of files that make up the component in a build configuration.

A sample SOURCES file is listed below:

```
#   ============================================================================
#   @file   SOURCES
#
#   @path   $(DSPLINK)/gpp/src/ldrv
#
#   @desc   This file contains list of source files to be compiled.
#
#   @ver    01.64
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ============================================================================

SOURCES :=

ifeq ($(USE_PROC), 1)
    SOURCES +=  ldrv.c                  \
                ldrv_proc.c             \
                DRV$(DIRSEP)ldrv_drv.c  \
                IPS$(DIRSEP)ldrv_ips.c  \
                SMM$(DIRSEP)ldrv_smm.c
endif

ifeq ($(USE_POOL), 1)
    SOURCES +=  POOLS$(DIRSEP)ldrv_pool.c

    ifeq ($(USE_PCPY_LINK), 1)
        SOURCES +=  POOLS$(DIRSEP)buf_pool.c
    endif
endif

ifeq ($(USE_CHNL), 1)
    SOURCES +=  ldrv_chnl.c    \
                ldrv_chirps.c  \
                DATA$(DIRSEP)ldrv_data.c
endif

ifeq ($(USE_MSGQ), 1)
    SOURCES +=  ldrv_msgq.c             \
                MQT$(DIRSEP)ldrv_mqt.c
endif

ifeq ($(USE_RINGIO), 1)
    SOURCES +=  RINGIO$(DIRSEP)ldrv_ringio.c
endif

ifeq ($(USE_MPCS), 1)
    SOURCES +=  MPCS$(DIRSEP)ldrv_mpcs.c
endif

ifeq ($(USE_MPLIST), 1)
    SOURCES +=  MPLIST$(DIRSEP)ldrv_mplist.c
endif
```

- For DSP side applications, the SOURCES file must also include the filenames that are 'generated' through processing of the tcf file by tconf.

Two additional parameters need to be defined for building a DSP side application executable.

TCF_FILE	Specifies the path of the application specific .tcf file relative to component base path.
CMD_FILE	Specifies the path of the application specific .cmd file relative to component base path.

32.1.4 DIRS

This file provides a list of sub-directories that make up the component in a build configuration.

A sample DIRS file is listed below:

```
#   ============================================================================
#   @file   DIRS
#
#   @path   $(DSPLINK)/gpp/src
#
#   @desc   This file defines the set of sub directories to be considered
#           by the MAKE system.
#
```

```
#   @ver    01.64
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ============================================================================


#   ============================================================================
#   Generic information about the component
#   ============================================================================

DIR_NAME    := SRC


#   ============================================================================
#   List of directories in the component
#   ============================================================================
ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAP3530")
DIRS  +=     \
    api
else # ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAP3530")
DIRS  +=     \
    arch     \
    gen      \
    osal     \
    ldrv     \
    pmgr     \
    api
endif  # ifeq ("$(TI_DSPLINK_PLATFORM)", "OMAP3530")
```

32.2 Common tasks

32.2.1 Adding a new source file

The source files can be of two types – header file (.h) and implementation file (.c).

HEADER FILE

If the new header file is local to a component and is not used by any other component, then it can reside in appropriate directory within the component.

If the header file is required by another component, then it should be exported during the build process. In this case, add the new header file to the list of similar header files defined by variable EXP_HEADERS in the COMPONENT file.

Since, there is one COMPONENT file per GPP OS, you will be required to make this change in the COMPONENT file for all GPP OSes.

IMPLEMENTATION FILE

The new implementation file simply needs to be compiled along with other such files.

Add the new implementation file to the list of other implementation files in the SOURCES file contained in the directory.

32.2.2 Changing the Compiler

The variable COMPILER represents the fully qualified path to the compiler used. It is defined in the file osdefs.mk for the default OS distribution, if no additional distributions are supported. If multiple distributions are supported for the OS, it is defined in the file <distribution>.mk for a specific distribution.

To change the compiler, simply change the value of this variable to the new compiler.

If the new compiler uses different switch settings, than the previous one, you may be required to update the variables based on the switches supported by the new compiler.

32.2.3 Changing the Archiver

The variable ARCHIVER represents the fully qualified path to the linker used. It is defined in the file osdefs.mk for the default OS distribution, if no additional distributions are supported. If multiple distributions are supported for the OS, it is defined in the file <distribution>.mk for a specific distribution.

To change the archiver, simply change the value of this variable to the new archiver.

If the new archiver uses different switch settings, than the previous one, you may be required to update the variables based on the switches supported by the new archiver.

32.2.4 Changing the Linker

The variable LINKER represents the fully qualified path to the linker used. It is defined in the file osdefs.mk for the default OS distribution, if no additional distributions are supported. If multiple distributions are supported for the OS, it is defined in the file <distribution>.mk for a specific distribution.

To change the linker, simply change the value of this variable to the new linker.

If the new linker uses different switch settings, than the previous one, you may be required to update the variable(s) based on the switches supported by the new linker.

You may also be required to update the commands using the variable LINKER for the targets - $(target_deb) and $(target_rel).

32.2.5 Supporting a new distribution

It is possible that you may be required to support more than one distribution of the operating system running on a processor. This possibility exists when more than one port of the OS is available e.g. in case of Linux, or when more than one target is supported by an operating system e.g. in case of DSP/BIOS™.

In such situations, it is inconvenient to continuously change the path to the code generation tools.

A distribution specific file (distribution.mk) can be created in such cases. In this file, all the definitions in the file osdefs.mk can be overridden. This file must set the value of variable USE_DISTRIBUTION as 1.

The listing below shows an example of a GPP side distribution file.

```
#   ============================================================================
#   @file   distribution.mk
#
#   @path   $(MAKEROOT)\gpp\make\GPPOSA\PLATFORMX
#
#   @desc   This makefile defines OS specific macros used by MAKE system.
#
#   @ver    01.64
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ============================================================================

ifndef DUMMY_MK

define DUMMY_MK
endef


#   ============================================================================
#   Let the make system know that a specific distribution for the GPP OS
#   is being used.
#   ============================================================================
USE_DISTRIBUTION    = 1


#   ============================================================================
#   Base directory for the GPP OS
#   ============================================================================
BASE_GPPOS      := GPPOSA


#   ============================================================================
#   Base for code generation tools - compiler, linker, archiver etc.
#   ============================================================================
BASE_CGTOOLS    := GPPOSA\bin


#   ============================================================================
#   Base directory for include files provided by GPP OS
#   ============================================================================
BASE_OSINC      := $(BASE_GPPOS)\inc

OSINC_GENERIC   := $(BASE_OSINC)\.
OSINC_PLATFORM  := $(BASE_OSINC)\PLATFORMX\.

ifneq ("$(VARIANT)", "")
```

```
OSINC_VARIANT   := $(BASE_OSINC)\PLATFORMX\.
endif


#   ============================================================================
#   Base directory for libraries provided by GPP OS
#   ============================================================================

BASE_OSLIB      := $(BASE_GPPOS)\Lib

OSLIB_GENERIC   := $(BASE_OSLIB)
OSLIB_PLATFORM  := $(BASE_OSLIB)

ifneq ("$(VARIANT)", "")
OSLIB_VARIANT   := $(BASE_OSLIB)
endif


#   ============================================================================
#   COMPILER
#   ============================================================================

#   ----------------------------------------------------------------------------
#   Name of the compiler
#   ----------------------------------------------------------------------------
COMPILER        := $(BASE_CGTOOLS)\compiler

CROSS_COMPILE   := crosscompile
export CROSS_COMPILE

#   ----------------------------------------------------------------------------
#   Command line switches used by the compiler
#
#   CC_SW_DEF       Command line defines
#   CC_SW_INC       Search path for header files
#   CC_SW_OBJ       Create object file
#   CC_SW_DEB       Include DEBUG information
#   ----------------------------------------------------------------------------
CC_SW_DEF       := -D
CC_SW_INC       := -I
CC_SW_OBJ       := -o
CC_SW_DEB       := -g

#   ----------------------------------------------------------------------------
#   Standard flags for the compiler
#   ----------------------------------------------------------------------------
STD_CC_FLAGS    := -Wall

#   ----------------------------------------------------------------------------
#   Flags for the compiler when building an executable
#   ----------------------------------------------------------------------------
EXE_CC_FLAGS    := -O

#   ----------------------------------------------------------------------------
#   Flags for the compiler when building a driver
#   ----------------------------------------------------------------------------
DRV_CC_FLAGS    := -O2

#   ----------------------------------------------------------------------------
#   Flags for the compiler when building a library
#   ----------------------------------------------------------------------------
LIB_CC_FLAGS    := -O2

#   ----------------------------------------------------------------------------
#   Standard definitions for the compiler
#   ----------------------------------------------------------------------------
STD_CC_DEFNS    :=


#   ============================================================================
#   ARCHIVER
#   ============================================================================
ARCHIVER        := $(BASE_CGTOOLS)\archiver

#   ----------------------------------------------------------------------------
#   Standard flags for the archiver
#   ----------------------------------------------------------------------------
STD_AR_FLAGS    := r


#   ============================================================================
#   LINKER
#   ============================================================================
LINKER      := $(BASE_CGTOOLS)\linker

#   ----------------------------------------------------------------------------
#   Command line switches used by the linker
#
#   LD_SW_LIB       Search path for libraries
#   LD_SW_OUT       Output filename
#   LD_SW_RELOC     Generate relocateable output
#   ----------------------------------------------------------------------------
LD_SW_LIB       := -L
LD_SW_OUT       := -o
LD_SW_RELOC     := -r

#   ----------------------------------------------------------------------------
#   Standard flags for the linker
#   ----------------------------------------------------------------------------
STD_LD_FLAGS    := -lc
```

```

#   ------------------------------------------------------------------------------
#   Flags for the linker when building an executable
#   ------------------------------------------------------------------------------
EXE_LD_FLAGS    :=

#   ------------------------------------------------------------------------------
#   Flags for the linker when building a driver
#   ------------------------------------------------------------------------------
DRV_LD_FLAGS    :=

endif   # ifndef DUMMY_MK
```

The listing below shows an example of a DSP side distribution file created for C64xx DSP, which is used to build DSP side applications on Windows workstation. This file depends on the paths where CGTOOLS and DSP/BIOS™ libraries are located on the build machine. You may need to change these if they are not already installed at the paths expected by the distribution file.

```
#   ==============================================================================
#   @file   c64xxp_5.xx_windows.mk
#
#   @path   $(DSPLINK)\make\DspBios
#
#   @desc   This makefile defines OS specific macros used by MAKE system for
#           the DSP/BIOS version 5.xx for C64XX PLUS on Windows.
#
#   @ver    01.64
#   ==============================================================================
#   Copyright (c) Texas Instruments Incorporated 2004
#
#   Use of this software is controlled by the terms and conditions found in the
#   license agreement under which this software has been supplied or provided.
#   ==============================================================================


ifndef C64XXP_5_XX_WINDOWS_MK

define C64XXP_5_XX_WINDOWS_MK
endef


#   ==============================================================================
#   Let the make system know that a specific distribution for the GPP OS
#   is being used.
#   ==============================================================================
USE_DISTRIBUTION := 1


#   ==============================================================================
#   Set the values of necessary variables to be used for the OS.
#   ==============================================================================
#   ----------------------------------------------------------------------------
#   Base directory for the DSP OS
#   ----------------------------------------------------------------------------
BASE_INSTALL    := C:\ti-tools
BASE_SABIOS     := $(BASE_INSTALL)\bios
BASE_BUILDOS    := $(BASE_SABIOS)\packages\ti\bios

#   ----------------------------------------------------------------------------
#   Base directory for the XDC tools
#   ----------------------------------------------------------------------------
XDCTOOLS_DIR    := $(BASE_SABIOS)\xdctools

#   ----------------------------------------------------------------------------
#   Base for code generation tools - compiler, linker, archiver etc.
#   ----------------------------------------------------------------------------
BASE_CGTOOLS    := $(BASE_INSTALL)\C6000\cgtools
BASE_CGTOOLSBIN := $(BASE_CGTOOLS)\bin

#   ----------------------------------------------------------------------------
#   Base for TCONF, platform files and dependent components
#   ----------------------------------------------------------------------------
BASE_TCONF      := $(XDCTOOLS_DIR)
BASE_PLATFORMS  := $(BASE_SABIOS)\packages
BASE_PSL        := $(BASE_SABIOS)\packages\ti\psl
BASE_CSL        :=
BASE_RTDX       := $(BASE_SABIOS)\packages\ti\rtdx

#   ----------------------------------------------------------------------------
#   Base directory for include files
#   ----------------------------------------------------------------------------
BASE_OSINC      := $(BASE_BUILDOS)\include
BASE_CGTOOLSINC := $(BASE_CGTOOLS)\include
BASE_RTDXINC    := $(BASE_RTDX)\include\c6000
BASE_PSLINC     := $(BASE_PSL)\include
BASE_CSLINC     :=

OSINC_GENERIC   := $(BASE_OSINC)
OSINC_PLATFORM  := $(BASE_CGTOOLSINC) $(BASE_RTDXINC) \
                   $(BASE_PSLINC) $(BASE_CSLINC)

ifeq ($(PLATFORM), DM642)
OSINC_PLATFORM  += $(BASE_INSTALL)\boards\evmdm642\include
endif # ifeq ($(PLATFORM), DM642)

ifneq ("$(VARIANT)", "")
OSINC_VARIANT   := $(BASE_OSINC)
endif
```

```
#   ----------------------------------------------------------------------------
#   Base directory for libraries
#   ----------------------------------------------------------------------------
BASE_OSLIB      := $(BASE_BUILDOS)\lib
BASE_CGTOOLSLIB := $(BASE_CGTOOLS)\lib
BASE_RTDXLIB    := $(BASE_RTDX)\lib\c6000
BASE_PSLLIB     := $(BASE_PSL)\lib
BASE_CSLLIB     :=

OSLIB_GENERIC   := $(BASE_OSLIB)
OSLIB_PLATFORM  := $(BASE_CGTOOLSLIB) $(BASE_RTDXLIB) \
                   $(BASE_PSLLIB) $(BASE_CSLLIB)

ifneq ("$(VARIANT)", "")
OSLIB_VARIANT   := $(BASE_OSLIB)
endif


#   ============================================================================
#   COMPILER
#   ============================================================================

#   ----------------------------------------------------------------------------
#   Name of the compiler
#   ----------------------------------------------------------------------------
COMPILER        := $(BASE_CGTOOLSBIN)\cl6x

#   ----------------------------------------------------------------------------
#   Command line switches used by the compiler
#
#   CC_SW_DEF       Command line defines
#   CC_SW_INC       Search path for header files
#   CC_SW_OBJ       Object file directory
#   CC_SW_DEB       Include debug information
#   CC_SW_REL       Release build
CC_SW_DEF       := -d
CC_SW_INC       := -I
CC_SW_OBJ       := -fr
CC_SW_DEB       := -g -d"_DEBUG" --no_compress
CC_SW_REL       := -o3

#   ----------------------------------------------------------------------------
#   Standard flags for the compiler
#   ----------------------------------------------------------------------------
STD_CC_FLAGS    := -q -pdr -pdv -pden -ml3 -mv6400+ --disable:sploop

#   ----------------------------------------------------------------------------
#   Standard flags for the compiler when building an executable
#   ----------------------------------------------------------------------------
EXE_CC_FLAGS    :=
#   ----------------------------------------------------------------------------
#   Flags for the compiler when building an archive
#   ----------------------------------------------------------------------------
ARC_CC_FLAGS    :=

#   ----------------------------------------------------------------------------
#   Standard definitions for the compiler
#   ----------------------------------------------------------------------------
STD_CC_DEFNS    :=


#   ============================================================================
#   ARCHIVER
#   ============================================================================
ARCHIVER        := $(BASE_CGTOOLSBIN)\ar6x

#   ----------------------------------------------------------------------------
#   Standard flags for the archiver
#   ----------------------------------------------------------------------------
STD_AR_FLAGS    := -r

#   ----------------------------------------------------------------------------
#   Archiver flags for extracting object files
#   ----------------------------------------------------------------------------
EXT_AR_FLAGS    := xq


#   ============================================================================
#   LINKER
#   ============================================================================
LINKER          := $(BASE_CGTOOLSBIN)\cl6x -z

#   ----------------------------------------------------------------------------
#   Command line switches used by the linker
#
#   LD_SW_INC       Search path for libraries
#   LD_SW_LIB       Include library name
#   LD_SW_OUT       Output file name
#   LD_SW_MAP       Map file name
#   LD_SW_RELOC     Generate relocateable output
#   ----------------------------------------------------------------------------
LD_SW_INC       := -i
LD_SW_LIB       := -l
LD_SW_OUT       := -o
LD_SW_MAP       := -m
LD_SW_RELOC     := -r

#   ----------------------------------------------------------------------------
#   Standard flags for the linker
#   ----------------------------------------------------------------------------
STD_LD_FLAGS    := -c -q -x

#   ----------------------------------------------------------------------------
#   Flags for the linker when building an executable
#   ----------------------------------------------------------------------------
EXE_LD_FLAGS    :=

#   ============================================================================
#   TCONF
#   ============================================================================
TCONF           := $(BASE_TCONF)\tconf


#   ----------------------------------------------------------------------------
#   Standard flags for TCONF
#   ----------------------------------------------------------------------------
STD_TCF_FLAGS   :=


endif   # ifndef C64XXP_5_XX_WINDOWS_MK
```

32.3 Text files generated during build process

Some text files are created as part of the build process in the $DSPLINK/gpp/BIN/Linux/Davinci/[RELEASE|DEBUG] directory.

These can be classified into the following types:

- COMPONENT_defines.txt states the –D defines needed during the build process.
- COMPONENT_includes.txt states the include directories needed during the build process.
- COMPONENT_flags.txt states the compiler flags needed during the build process.

Where COMPONENT is API, GEN, LDRV, PMGR, OSAL, SAMPLES (LOOP, MESSAGE, MPCSXFER, SCALE, RINGIO etc).

Out of these API is the component present in the user space whereas GEN, LDRV, PMGR, OSAL is the components present in kernel space. These files provide the include paths, definitions and compiler flags which are a guide for the build system in the porting process.

For deciding the include paths, definitions and compiler flags for the application, a good place to start is to look at the samples which are similar to the configuration that the application is using. For e.g. if the configuration includes MSGQ, refer to the include paths, definitions and compiler flags from MESSAGE_includes.txt, MESSAGE_defines.txt, MESSAGE_flags.txt respectively. If configuration includes RINGIO, refer to the include paths, definitions and compiler flags from RINGIO_includes.txt, RINGIO_defines.txt, RINGIO_flags.txt respectively.

33 Scripts to load and unload dsplinkk.ko module in Linux based targets

Scripts loaddsplink.sh and unloaddsplink.sh are provided under $dsplink/etc/host/scripts/Linux to load and unload dsplinkk.ko module.

1. To load the dsplinkk.ko module run the following command

   sh ./loaddsplink.sh

2. To unload the kernel module run the following command.

   sh ./unloaddsplink.sh.