DSP/BIOS™ LINK

OSK5910 Starter Kit (OSK)

Montavista Linux Professional Edition 3.1

LNK 058 USR

Version 1.10.01

APR 23, 2004

This page has been intentionally left blank.

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

TABLE OF FIGURES

A. 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 or more 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[2]. The release package contains full source code to enable the customers to port it to their specific platforms and/ or operating systems.

Currently DSP/BIOS™ LINK provides following services to its clients:

§ Basic processor control

§ Data transfer over logical channels

§ Messaging (based on MSGQ module of DSP/BIOS)[3]

A typical application may not require all these services. In such cases, DSP/BIOS™ LINK can be scaled at compile time to use 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.
q	This bullet indicates additional information.

3 Terms & Abbreviations

CCS	Code Composer Studio
IPC	Inter Processor Communication

[1] Applications differentiate the products. The application developers would prefer to focus on the application rather than the IPC mechanism.
[2] The current reference platform is the OMAP5912 Starter Kit (OSK).
[3] The MSGQ module is currently available as an add-on to DSP/BIOS. This component has a dependency on the data transfer services.

4 References

1.	SPRA987	MSGQ Module: DSP/BIOS Support for Variable Length Messaging
2.		MontaVista™ Linux® Professional Edition 3.1 Users Guide dated FEB 03, 2004

B. WHERE TO BEGIN?

5 Available Documents

5.1 Platform Specific

These documents are specific to the OMAP5910 OSK where Linux is running on the GPP OS.

1.	INSTALLATION GUIDE	InstallGuide.pdf
	This document provides information to install DSP/BIOS™LINK on the development host and setup the development platform.	
2.	USER GUIDE	UserGuide.pdf
	The current document. This document provides information to get started on DSP/BIOS™ LINK.	
3.	RELEASE NOTES	ReleaseNotes.pdf
	This document provides information on the current release [Version 1.10.01].	
4.	Shared Memory IOM Driver Design for OMAP	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.	
5.	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.	

5.2 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.	SOURCE REFERENCE GUIDE	LNK_018_REF.pdf
	This document provides the details of the objects and interfaces exported by various components of DSP/BIOS™ LINK.	
2.	PROCESSOR MANAGER	LNK_010_DES.pdf
	This document describes the detailed design of the Processor Manager component.	
3.	LINK DRIVER	LNK_012_DES.pdf
	This document describes the detailed design of the Link Driver component.	
4.	MESSAGING USING MSGQ	LNK_031_DES.pdf
	This document describes the detailed design of the messaging component utilizing MSGQ module of DSP/BIOS™.	
5.	TEST SUITE	LNK_015_DES.pdf

	This document describes the detailed design of the test suite included in DSP/BIOS™ LINK.	
6.	PORTING GUIDE	LNK_017_DES.pdf
	Provides recommendations and guidelines for the developers to port DSP/BIOS™ LINK to either a different GPP OS, a different platform or a different physical link.	
7.	DSP Executable Loader Design	LNK_040_DES.pdf
	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.	

C. SOFTWARE ARCHITECTURE

6 Overview

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

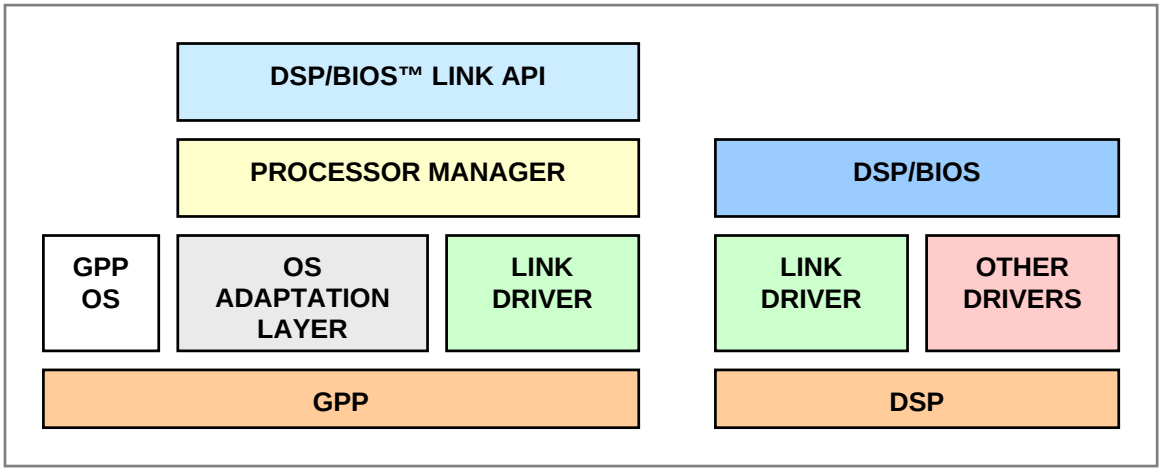

Figure 1. Software architecture of DSP/BIOS™ LINK

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 real processing happens in the PROCESSOR MANAGER. It builds upon the low-level control operations provided by the LINK DRIVER.

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.

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.

There is no specific DSP/BIOS™ LINK API on the DSP. The communication (data/ message transfer) is done using the DSP/BIOS™ modules- SIO/ GIO/ MSGQ.

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.

§ Stop execution.

In the current version, only one processor is supported. However, `processorId` is passed as an argument to the API for future compatibility.

7.2 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[4]. 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.

7.3 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[5]. It is based on the MSGQ module in DSP/BIOS™.

The messages are sent and received through message queues.

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

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.

On the GPP, the MSGQ leverages the data transfer infrastructure provided by the CHNL component.

8 Details

This section illustrates the interaction of the components described in the previous section. For more details on each component, refer to the corresponding design documents.

8.1 On the GPP side

The detailed architecture on the GPP side is shown below:

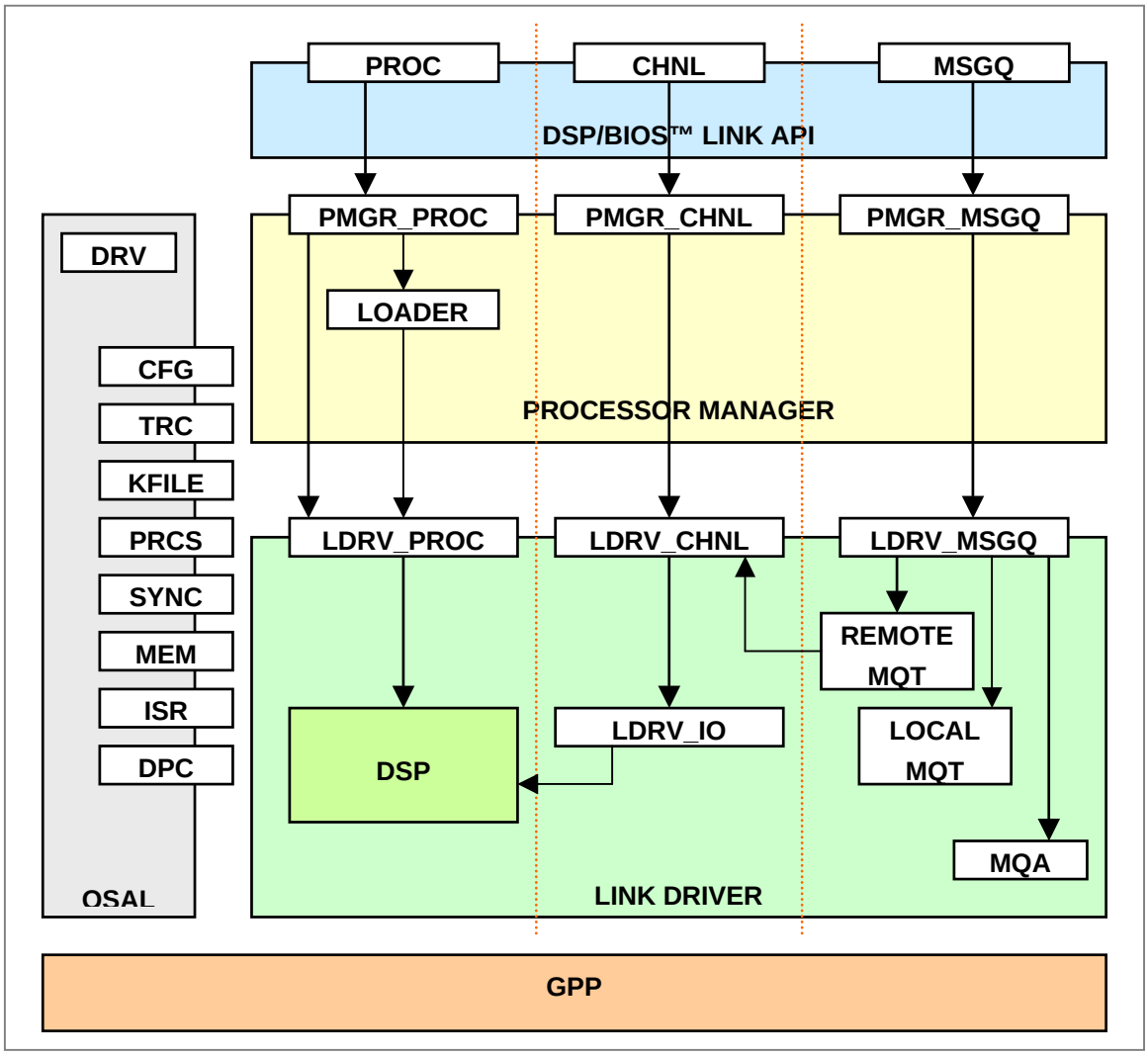

Figure 2. Architecture on GPP side

The vertical dotted lines indicate the division of functionality for processor control, data transfer and messaging.

For details on the loader component refer to DSP Executable Loader Design document.

8.2 On the DSP side

The detailed architecture on the DSP side is shown below:

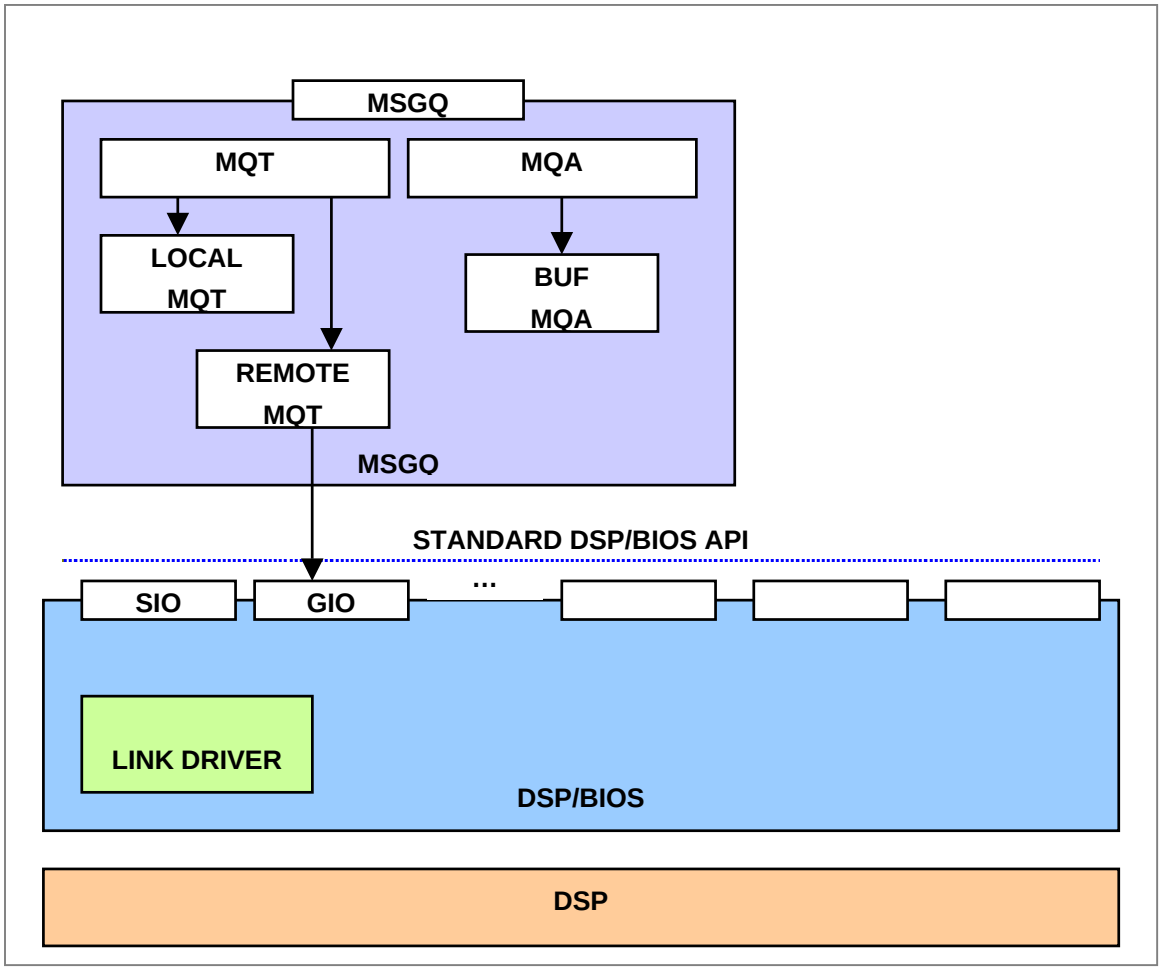

Figure 3. Architecture on DSP side

The LINK DRIVER is implemented as an IOM driver on the DSP side. Either of the class driver implementations – SIO and GIO can utilize the services of this IOM driver.

The applications can use the APIs exported by SIO and GIO modules to transfer data across the GPP and DSP on specified channels.

The MSGQ module uses the GIO interface to provide the messaging services.

The queues created for DSPLINK on the DSP side must have name as DSPLINK_DSP00MSGQXX where XX is the number used for identifying it on the GPP side.

On the GPP side, message queue is identified through a MSGQ ID (say YY) and it is recognized on the DSP side as DSPLINK_GPPMSGQYY.

9 Source Code Layout

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

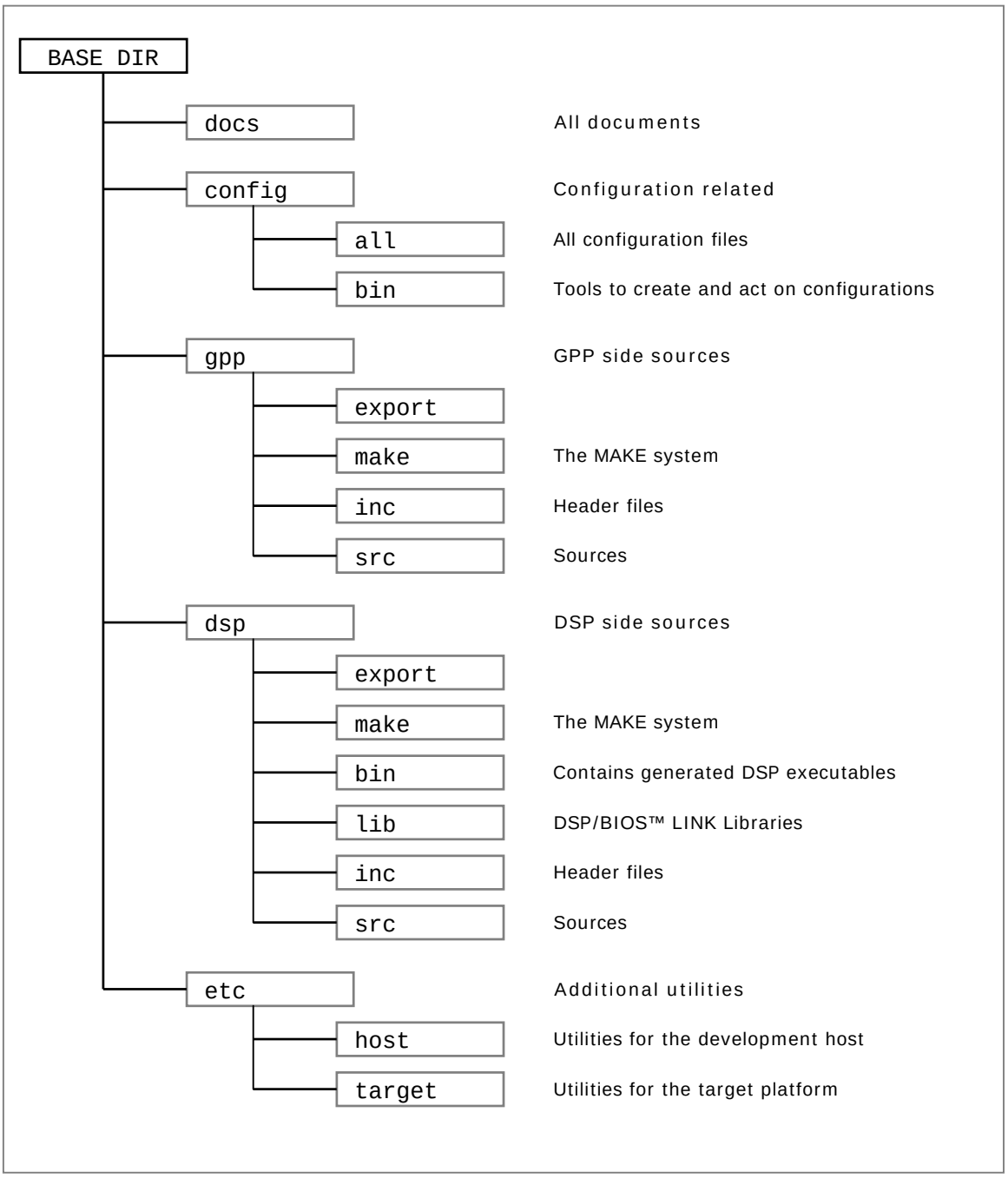

Figure 4. Top level view of directory structure

9.1 GPP side sources

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

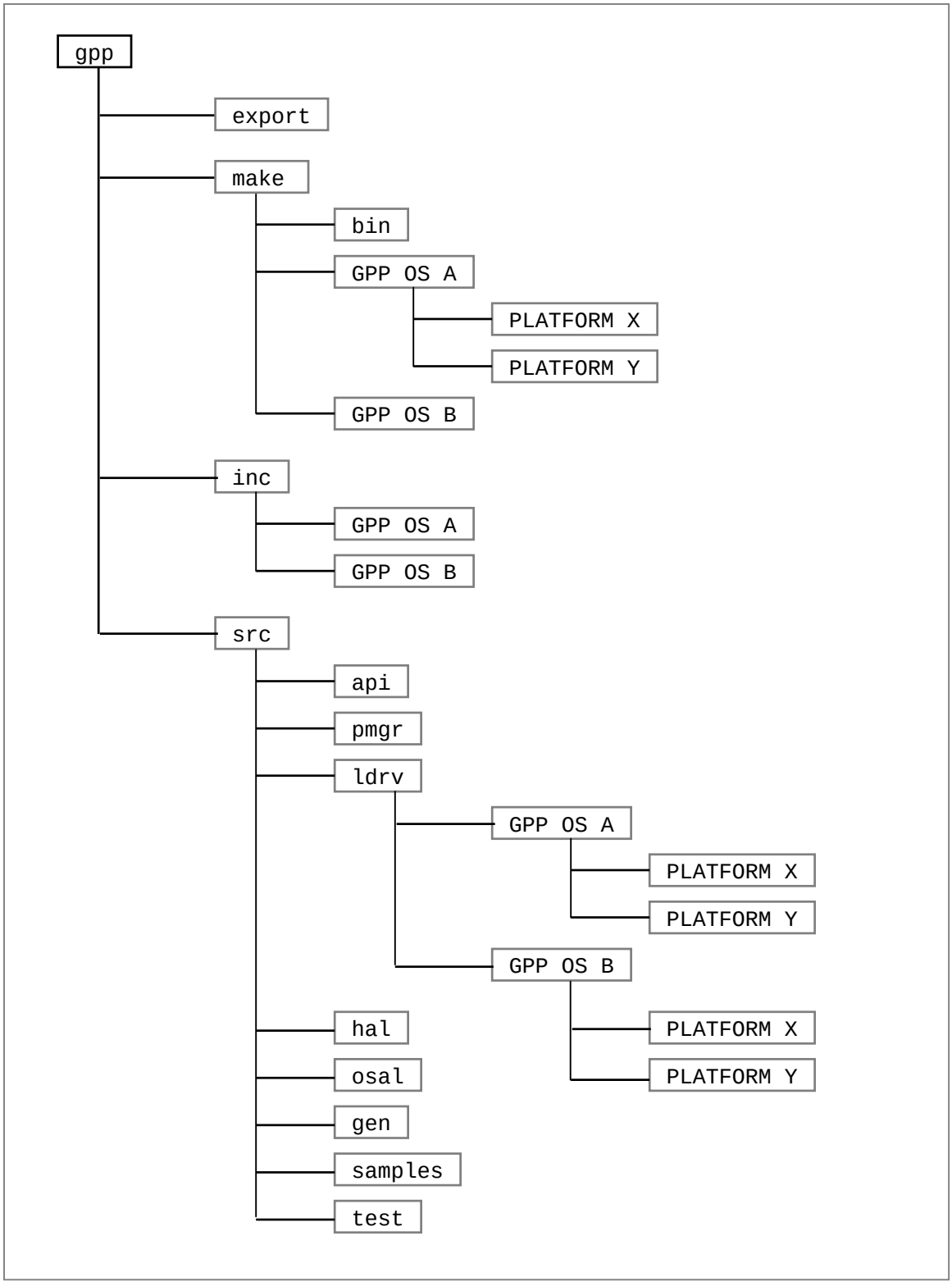

Figure 5. Directory structure for GPP side sources

9.2 DSP side sources

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

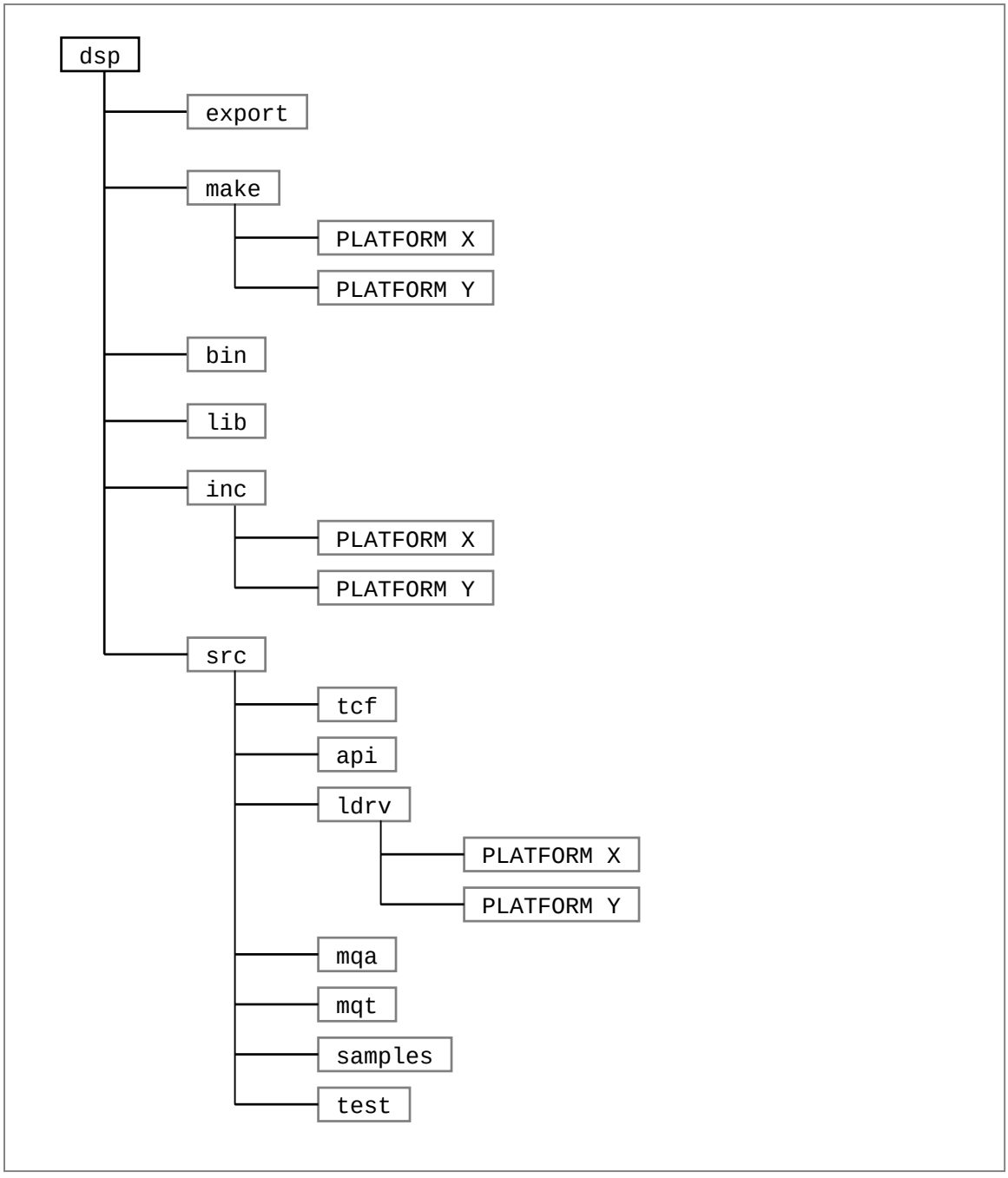

Figure 6. Directory structure for DSP side sources

D. BUILD PROCEDURE

10 Build Configuration

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. See section 11 for details on setting this environment variable.

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

The different menus presented during the build configuration are described below:

1. The first menu confirms if the environment variable DSPLINK is set correctly.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DSPLINK is currently defined as:

L:\dsplink



1.   Continue.

2.   Quit to change.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

2. If the environment variable DSPLINK points to a non-existent directory, the menu similar to one below is presented.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DSPLINK is currently defined as:

d:\dummy



!! ERROR !! Invalid path assigned to DSPLINK!
```

```
Could not find following directories:
d:\dummy


1.   Continue.

2.   Quit to change.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

3. Next menu allows user to choose the OS running on the GPP side.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Choose the OS running on GPP

1.   Montavista Linux Pro 3.1

2.   Montavista Linux Pro 3.0

3.   Monterey Linux

4.   Nucleus

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

4. Next menu allows user to choose the target platform.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Choose the target platform

1.   OMAP5912 OSK

2.   Innovator(TM) using OMAP5910

3.   DM310 with DM642 using HPI Interface
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

5. Next menu allows user to choose the variant of the target platform. If the variants are present, they are listed in the menu.

   If there is no variant (or not currently supported by DSP/BIOS™ LINK), following menu appears:

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

No variant is supported for this platform.

Press <ENTER> to continue...

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

6. Next menu allows user to choose the components to be included while building DSP/BIOS™ LINK.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Choose the target platform

1.   PROC + CHNL + MSGQ

2.   PROC + CHNL

3.   PROC

4.   DSP  (Minimal interface for DSP Control)

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

7. Next menu allows user to choose if the debug trace is enabled/ disabled.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
Enable debug trace?

0.   No

1.   Yes

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

8. Next menu allows user to choose the level of profiling information to be collected during execution.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
           DSP/BIOS(TM) LINK  Configuration Tool
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Enable profiling?

0.   No

1.   Yes. Basic only.

2.   Yes. Detailed.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

YOUR CHOICE :
```

Based on the input values, the configuration parameters are stored in the file CURRENTCFG.MK.

11 Build GPP side sources

To build the GPP side sources, follow the steps below:

1. Set up necessary environment variables.

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

- The above command assumes that you are using *tcsh* shell. If you are using *bash* shell, an equivalent script '*gppenv.bash*' is shipped with the release package that can be used.

  This script sets/ modifies following environment variables:

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

- This command can be included in the '.rc' file corresponding to your shell.

2. Change to the source directory:

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

3. Start the build process:

```
$ gmake –s [debug | release]
```

- The '-s' option causes gmake to build silently. Please refer to gmake documentation for other options.

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

   ~/dsplink/gpp/export/BIN/Linux/OMAP/DEBUG

   ~/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE

12 Build DSP side sources

To build the DSP side sources, follow the steps below:

1. Set up necessary environment variables.

```
L:\> dsplink\etc\host\scripts\msdos\dspenv
```

   This script sets/ modifies following environment variables:

PATH	Modifies the path to include scripts provided in the installation and executables from CCS installation.

- If CCS is installed at a location other than *C:\ti*, the file – *dspenv.bat* - must be modified appropriately.

2. Change to the make folder within your workspace:

```
L:\> cd dsplink\dsp\make\OMAP
```

3. Start the build process:

```
L:\dsplink\dsp\make\OMAP\> timake.exe dsplink.pjt [debug | release]
```

4. To build the DSP/BIOS LINK messaging library, use the following command:

```
L:\dsplink\dsp\make\OMAP\> timake.exe dsplinkmsg.pjt [debug | release]
```

- Following remarks are generated when building the DSP-side sources. These remarks can be safely ignored:

  dsplink.pjt:

  - File: "driver.c": remark #880-D: parameter "devid" was never referenced
  - File: "interrupt.c": remark #880-D: parameter "arg" was never referenced

  dsplinkmsg.pjt:

  - File: "mqtdsplink.c": remark #880-D: parameter "bufPtr" was never referenced
  - File: "mqtdsplink.c": remark #880-D: parameter "size" was never referenced

- The CCS project files shipped with DSP/BIOS™ LINK have messaging (using MSGQ) enabled by default.

- To exclude the messaging from DSP side builds, update the CCS project files to remove the definition of *_MSGQ_COMPONENT* from the compiler options.

E. 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/ unload the DSP and load another executable/ etc.

13.1.2 APIs used

1. `PROC_Setup ()`
2. `PROC_Attach ()`
3. `PROC_Load ()`

13.2 CHNL

13.2.1 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.

13.2.2 APIs used

1. CHNL_Create ()
2. CHNL_AllocateBuffer ()

13.3 MSGQ

13.3.1 Typical sequence

1. Initialize the allocator to be used for messaging.

   The buffers used for transferring messages are allocated from this allocator.

   The default allocator shipped with DSP/BIOS™ LINK is called MQABUF. It creates buffer pools of fixed size specified by the user. The memory for the buffer pool is allocated at the time of opening this allocator.

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

2. Open the local transport to be used for messaging.

   Local transport manages the queues created locally on the GPP.

3. Creates a message queue. All messages destined for the client will be added to this queue.
4. Creates a message queue where the asynchronous error messages will be queued.

   This queue can be same as the one created in the previous step.

5. Open the remote transport (on the DSP) to be used for messaging.
6. Locate the remote queue for sending messages.

- Each allocator must be initialized only once. However, multiple allocators (of differing IDs) using the same allocator function table can be configured using the static configuration tool. Each of these allocators must be initialized once.
- Steps 5 and 6 may require a response from the DSP. They should, therefore, be executed only after the DSP is running.

The queues created for DSPLINK on the DSP side must have name as DSPLINK_DSP00MSGQXX where XX is the number used for identifying it on the GPP side.

On the GPP side, message queue is identified through a MSGQ ID (say YY) and it is recognized on the DSP side as DSPLINK_GPPMSGQYY.

13.3.2 APIs used

1. MSGQ_AllocatorOpen ()
2. MSGQ_TransportOpen ()
3. MSGQ_Create ()
4. MSGQ_SetErrorHandler ()
5. MSGQ_Locate ()

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. Once the application completes, the execution is stopped.

14.1.2 Relevant APIs

1. `PROC_Start ()`
2. `PROC_Stop ()`

14.2 CHNL

14.2.1 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 the buffer that was issued on the channel 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).

14.2.2 Relevant APIs

1. `CHNL_Issue ()`
2. `CHNL_Reclaim ()`

14.3 MSGQ

14.3.1 Typical sequence

1. Allocate a message using the allocator.
2. Send the message to the message queue.
3. Receive the message from the message queue.
4. Free the message.

14.3.2 Relevant APIs

1. `MSGQ_Alloc ()`
2. `MSGQ_Put ()`
3. `MSGQ_Get ()`

4. MSGQ_Free ()

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 goes down to reset the hardware interfacing with 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 CHNL

15.2.1 Typical sequence

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

15.2.2 Relevant APIs

1. `CHNL_FreeBuffer ()`
2. `CHNL_Delete ()`

15.3 MSGQ

15.3.1 Typical sequence

1. Release the remote message queue.
2. Close the remote transport.
3. Delete the local message queue.
4. Close the local transport.
5. Close the allocator.

15.3.2 Relevant APIs

1. `MSGQ_Release ()`
2. `MSGQ_TransportClose ()`
3. `MSGQ_Delete ()`
4. `MSGQ_AllocatorClose ()`

F. 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.

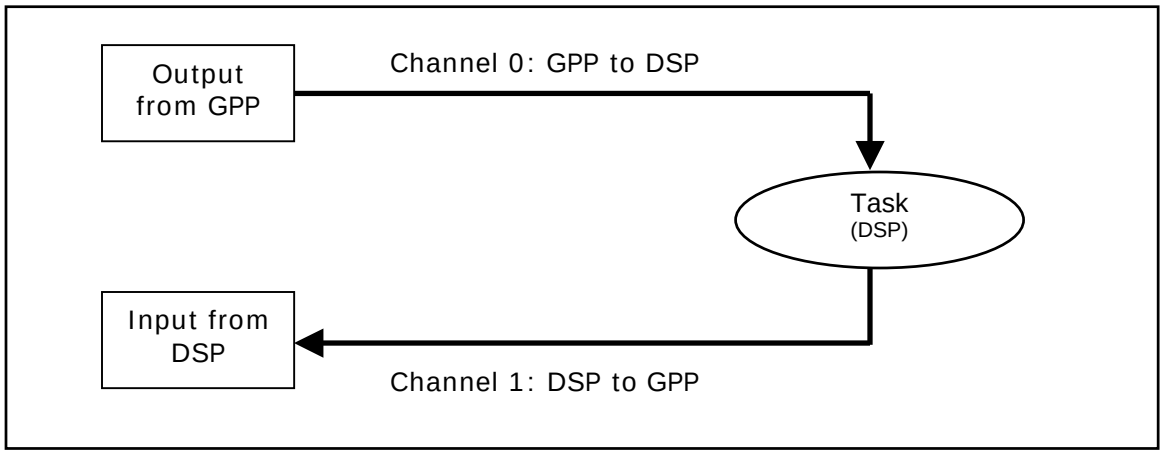

Figure 7. Data flow in the sample application – LOOP

This application illustrates a very simple data transfer scenario:

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 ID_PROCESSOR.
2. It loads DSP executable (`loop.out`) on the DSP.
3. It creates channels CHNL_ID_INPUT and CHNL_ID_OUTPUT for data transfer.
4. It allocates and primes one buffer each 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 detaches itself from ID_PROCESSOR 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. This task creates SIO channels for data transfer - TSK_INPUT_CHANNEL and TSK_OUTPUT_CHANNEL.
3. It allocates and primes the buffers for to be used for data transfer.

EXECUTION

1. The task issues an empty buffer on TSK_INPUT_CHANNEL and waits to reclaim it. The reclaim is specified to wait forever.
2. It then issues the same buffer on TSK_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 TSK_INPUT_CHANNEL and TSK_OUTPUT_CHANNEL.

16.1.2.2 Using SWI with GIO

INITIALIZATION

1. In the function `main ()`, GIO channels for data transfer - SWI_INPUT_CHANNEL and SWI_OUTPUT_CHANNEL are created.
2. 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.
3. 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 SWI_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 SWI_INPUT_CHANNEL and SWI_OUTPUT_CHANNEL are deleted.

16.2 Build

16.2.1 On the GPP side

The procedure to build loop application is same as that of dsplink.

1. Change to the directory containing loop sample.

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

2. To build the application:

```
$ gmake –s [debug | release]
```

3. Upon successful completion of build, loopgpp shall be created in the directories:

   ~/dsplink/gpp/export/BIN/Linux/OMAP/DEBUG

   ~/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE

16.2.2 On the DSP side

As described above, the DSP side application can either be executed with TSK or SWI. This configuration is controlled by the variable APPLICATION_MODE in file main.c.

Modify this value according to the configuration in which you want the DSP side to execute.

```
#define APPLICATION_MODE TSK_MODE
```

OR

```
#define APPLICATION_MODE SWI_MODE
```

As shipped in the release package the sample is configured to run in TSK_MODE.

For building the DSP side, change to the sample directory and build the project using timake.exe.

```
L:\> cd dsplink\dsp\src\samples\loop\OMAP
L:\dsplink\dsp\src\samples\loop\OMAP> timake loop.pjt [debug | release]
```

The DSP side binaries are created in the directories – loop\OMAP\Debug OR loop\OMAP\Release - based on type of build specified on the command line.

Loop sample built using the commands shown above contains both TEXT and BIOS sections in internal memory. To build the application with any of these sections in external memory, select another project file accordingly. Use

loop_bios.pjt	.bios section in external memory
loop_text.pjt	.text section in external memory
loop_bios_text.pjt	Both .bios and .text sections in external memory

16.3 Execute

The loop sample illustrates basic data streaming concepts in DSP/BIOS™ LINK.

16.3.1 Copying files to target file system

The generated binaries on the GPP side and DSP side and the data files must be copied to the target directory.

GPP Side

For executing the DEBUG build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/loopgpp
../montavista/target/opt/dsplink/samples/loop/DEBUG
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

For executing the RELEASE build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/loopgpp
../montavista/target/opt/dsplink/samples/loop/RELEASE
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

- Enter the commands shown above in single line.

DSP Side

The DSP binaries are built on the Windows host. These binaries must be copied into the target file system. Any FTP client can be used for this transfer.

For executing the DEBUG build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/loop/DEBUG:

   C:\dsplink\dsp\src\samples\loop\OMAP\Debug\loop.out

For executing the RELEASE build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/loop/RELEASE:

```
C:\dsplink\dsp\src\samples\loop\OMAP\Release\loop.out
```

16.3.2 Loading the kernel module: dsplinkk.o

To load the device driver, login as 'root' and enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink load [.]
```

- If the argument . (dot) in the command is omitted, the kernel module located at */opt/dsplink/* is loaded.

This action generates a warning indicating that the kernel module does not contain the GPL license. This warning can be safely ignored.

16.3.3 Invoking the application

To invoke the application enter the following commands:

```
$ cd /opt/dsplink/samples/loop
$ ./loopgpp loop.out <buffersize> <iterations>
```

e.g.

```
$ ./loopgpp loop.out 128 10000
```

16.3.4 Unloading the kernel module: dsplinkk.o

To unload the device driver, enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink unload
```

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.

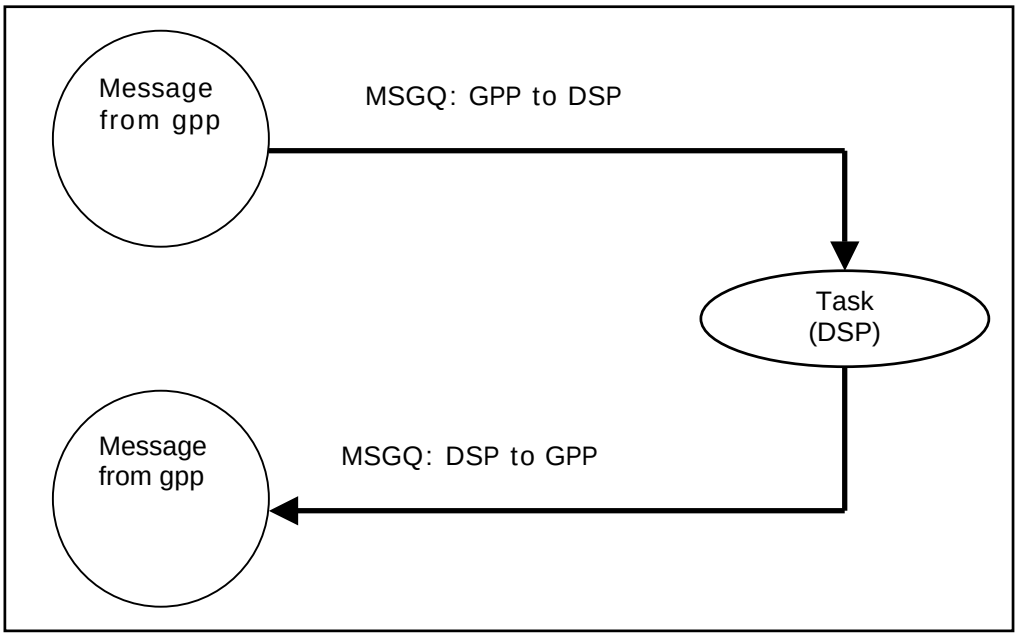

Figure 8. Message flow in the sample application – MESSAGE

This application illustrates a very simple message transfer scenario:

17.1.1 On the GPP side

INITIALIZATION

1. The client sets up the necessary data structures for accessing the DSP.
2. It then opens the default allocator to be used for messaging.
3. It then opens the local transport.
4. It then creates a message queue with GPPMSGQID identifier on the local processor.
5. It sets the above-created queue as the error handler.
6. It then attaches to the DSP identified by ID_PROCESSOR.
7. It loads DSP executable (`message.out`) on the DSP
8. The client starts the execution on DSP.
9. It then opens the remote transport.
10. It then attempts to locate the queue 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.

EXECUTION

1. The client allocates a message from the default allocator.
2. It puts the message the message on the DSP message queue.
3. It then tries to get a message on the local queue. The get operation is specified to wait forever.
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.
6. It issues an empty buffer on CHNL_ID_INPUT and waits to reclaim it. The reclaim is specified to wait forever.
7. It frees the message that was received on the local message queue on 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 detaches itself from ID_PROCESSOR.
5. It resets the error handler which was set in the create phase.
6. It deletes the local message queue.
7. It closes the locate transport.
8. It closes the default allocator.
9. It destroys the PROC component.

17.1.2 On the DSP side

17.1.2.1 Using TSK with MSGQ

INITIALIZATION

1. The client task tskMessage is created in the function `main ()`.
2. It initializes MSGQ module, default allocators and transports.
3. It then creates a message queue with DSPLINK_DSP00MSGQ00 name on the local processor (DSP).
4. It attempts to locate the queue created on the GPP side (DSPLINK_GPPMSGQ00). Locate is specified to wait forever. If the Locate call was unsuccessful (GPP queue still not created), it sleeps for some time and tries to locate the queue again.
5. It sets the above-created queue as the error handler.

EXECUTION

1. The task tries to get a message on the local queue. The get operation is specified to wait forever.
2. It then puts the same message on DSPLINK_GPPMSGQ00.

3. These steps are repeated until the MSGQ puts a message on the local message queue indicating that the GPP side transport has closed.

FINALIZATION

1. The client releases the remote message queue on the DSP side.
2. It unsets the error handler which was set in the create phase.
3. It deletes the local message queue.
4. It then finalizes MSGQ module, default allocators and transports.

17.1.2.2 Using SWI with MSGQ

INITIALIZATION

1. The SWIMESSAGE_create is called from the function `main ()`.
2. It initializes MSGQ module, default allocators and transports.
3. A SWI object is created for doing the message transfer. One of the attributes for the SWI object is the callback function `messageSWI`.
4. It then creates a message queue with DSPLINK_DSP00MSGQ00 name on the local processor (DSP). Here the callback function is specified as the SWI object created above.
5. It attempts to locate the queue created on the GPP side (DSPLINK_GPPMSGQ00). This Locate operation is asynchronous.

EXECUTION

1. Here the messageSWI SWI will be posted whenever a message is ready on the local message queue.
2. If the message ID is of type MSGQ_ASYNCLOCATEMSGID it populates its queue handle for the remote message queue (DSPLINK_GPPMSGQ00). This indicates the completion of the asynchronous locate which was started in initialization phase.
3. If the message is neither locate acknowledgement nor asynchronous error notification, it performs the get operation with no timeout.
4. It then puts the same message on DSPLINK_GPPMSGQ00.

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 unsets the error handler which was set in the create phase.
3. It deletes the local message queue.
4. It then finalizes MSGQ module, default allocator and transports.

17.2 Build

17.2.1 On the GPP side

The procedure to build sample application is same as that of dsplink.

1. Change to the directory containing message sample.

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

2. To build the application:

```
$ gmake –s [debug | release]
```

3. Upon successful completion of build, messagegpp shall be created in the directories:

   ~/dsplink/gpp/export/BIN/Linux/OMAP/DEBUG

   ~/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE

17.2.2 On the DSP side

As described above, the DSP side application can either be executed with TSK or SWI. This configuration is controlled by the variable APPLICATION_MODE in file main.c.

Modify this value according to the configuration in which you want the DSP side to execute.

```
#define APPLICATION_MODE TSK_MODE
```

OR

```
#define APPLICATION_MODE SWI_MODE
```

As shipped in the release package the sample is configured to run in TSK_MODE.

For building the DSP side, change to the sample directory and build the project using timake.exe.

```
L:\> cd dsplink\dsp\src\samples\message\OMAP
L:\dsplink\dsp\src\samples\message\OMAP> timake message.pjt [debug | release]
```

The DSP side binaries are created in the directories - message\OMAP\Debug OR message\OMAP\Release - based on type of build specified on the command line.

Message sample built using the commands shown above contains both TEXT and BIOS sections in internal memory. To build the application with any of these sections in external memory, select another project file accordingly.

message_bios.pjt	.bios section in external memory
message_text.pjt	.text section in external memory
message_bios_text.pjt	Both .bios and .text sections in external memory

17.3 Execute

The message sample illustrates basic message transferring concepts in DSP/BIOS™ LINK

17.3.1 Copying files to target file system

The generated binaries on the GPP side and DSP side and the data files must be copied to the target directory.

GPP Side

For executing the DEBUG build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/messagegpp
../montavista/target/opt/dsplink/samples/message/DEBUG
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

For executing the RELEASE build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/messagegpp
../montavista/target/opt/dsplink/samples/message/RELEASE
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

- Enter the commands shown above in single line.

DSP Side

The DSP binaries are built on the Windows host. These binaries must be copied into the target file system. Any FTP client can be used for this transfer.

For executing the DEBUG build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/message/DEBUG

   C:\dsplink\dsp\src\samples\message\OMAP\Debug\message.out

For executing the RELEASE build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/message/RELEASE

   C:\dsplink\dsp\src\samples\message\OMAP\Release\message.out

17.3.2 Loading the kernel module: dsplinkk.o

To load the device driver, login as 'root' and enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink load [.]
```

- If the argument . (dot) in the command is omitted, the kernel module located at */opt/dsplink/* is loaded.

This action generates a warning indicating that the kernel module does not contain the GPL license. This warning can be safely ignored.

17.3.3 Invoking the application

To invoke the application enter the following commands:

```
$ cd /opt/dsplink/samples/message
$ ./messagegpp message.out <number of transfers>
```

e.g.

```
$ ./messagegpp message.out  10000
```

17.3.4 Unloading the kernel module: dsplinkk.o

To unload the device driver, enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink unload
```

18 SCALE

18.1 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 & MSGQ, and SWI with GIO & MSGQ.

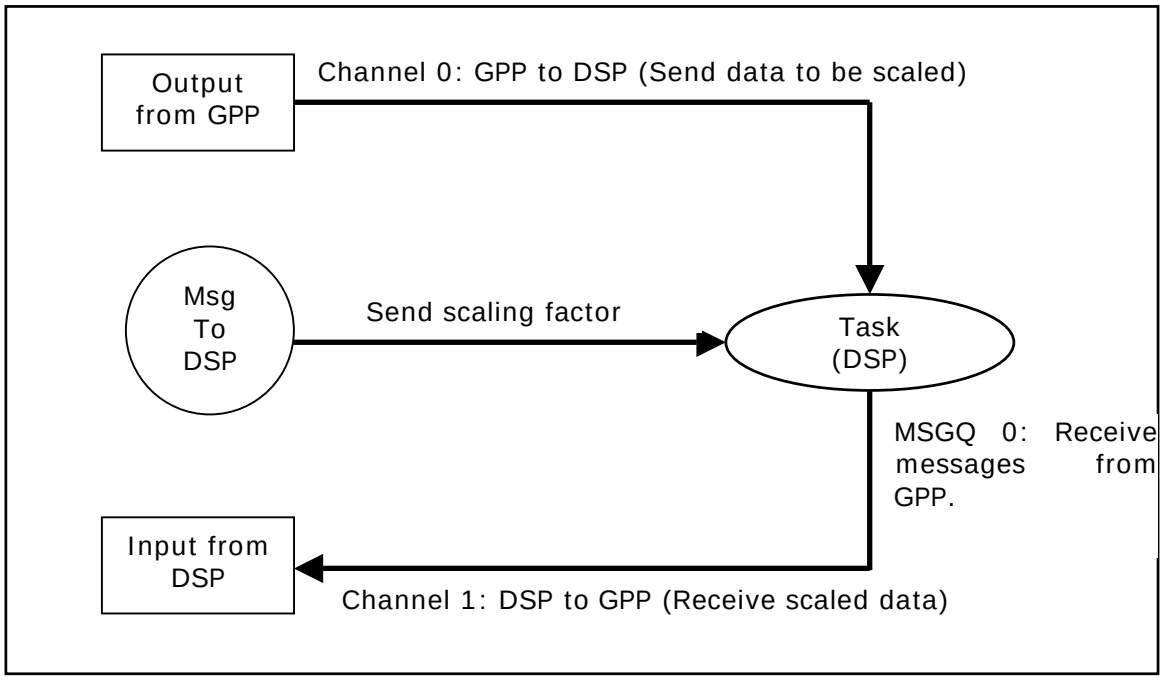

Figure 9. Data and message flow in the sample application – SCALE

This application illustrates a very simple message + data transfer scenario:

18.1.1 On the GPP side

INITIALIZATION

1. The client calls APIs required for making the DSP accessible.
2. It opens the default allocator to be used for messaging.
3. It then opens the local transport.
4. It then attaches to the DSP identified by ID_PROCESSOR.
5. It loads DSP executable (`scale.out`) on the DSP
6. It creates channels CHNL_ID_INPUT and CHNL_ID_OUTPUT for data transfer.
7. It allocates and initializes one buffer each of specified size for data transfer on these channels.
8. The client starts the execution on DSP.
9. 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 buffers allocated for data transfer.
4. It deletes the channels CHNL_ID_INPUT and CHNL_ID_OUTPUT.
5. It detaches itself from ID_PROCESSOR.
6. It closes the local transport.
7. It then closes the default allocator.
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 task initializes MSGQ module, the default allocator and transports.
3. This task creates SIO channels for data transfer - TSK_INPUT_CHANNEL and TSK_OUTPUT_CHANNEL.
4. It allocates and initializes the buffer for to be used for data transfer.
5. It then creates a message queue with DSPLINK_DSP00MSGQ00 name on the local processor (DSP).

EXECUTION

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

4. 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.
5. 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 TSK_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 TSK_INPUT_CHANNEL and TSK_OUTPUT_CHANNEL.
3. The task frees the buffer allocated for data transfer.
4. Then it finalizes MSGQ module, default allocator and transports.

18.1.2.2 Using SWI with GIO and MSGQ

INITIALIZATION

6. SWISCALE_create is called from the function `main ()`.
7. It initializes MSGQ module, default allocator and transports.
8. It then creates two GIO channels for data transfer - SWI_INPUT_CHANNEL and SWI_OUTPUT_CHANNEL.
9. 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.
10. The buffers to be used for data transfer are then allocated and initialized.
11. It then creates a message queue with DSPLINK_DSP00MSGQ00 name on the local processor (DSP). Here the callback function is specified as the message SWI object created above.

EXECUTION

1. To initiate the data transfer a READ request on the input buffer is submitted on the SWI_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:

5. The data and message SWIs are deleted.
6. The local message queue is deleted.
7. The GIO channels SWI_INPUT_CHANNEL and SWI_OUTPUT_CHANNEL are deleted.
8. The buffers allocated for data transfer are freed.
5. Then the MSGQ module, default allocator and transports are finalized.

18.2 Build

18.2.1 On the GPP side

The procedure to build scale application is same as that of dsplink.

1. Change to the directory containing scale sample.

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

2. To build the application:

```
$ gmake –s [debug | release]
```

3. Upon successful completion of build, scalegpp shall be created in the directories:

   ~/dsplink/gpp/export/BIN/Linux/OMAP/DEBUG

   ~/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE

18.2.2 On the DSP side

As described above, the DSP side application can either be executed with TSK or SWI. This configuration is controlled by the variable APPLICATION_MODE in file main.c.

Modify this value according to the configuration in which you want the DSP side to execute.

```
#define APPLICATION_MODE TSK_MODE
```

OR

```
#define APPLICATION_MODE SWI_MODE
```

As shipped in the release package the sample is configured to run in TSK_MODE.

For building the DSP side, change to the sample directory and build the project using timake.exe.

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

```
L:\dsplink\dsp\src\samples\scale\OMAP> timake scale.pjt [debug |
release]
```

The DSP side binaries are created in the directories – scale\OMAP\Debug OR scale\OMAP\Release - based on type of build specified on the command line.

Scale sample built using the commands shown above contains both TEXT and BIOS sections in internal memory. To build the application with any of these sections in external memory, select another project file accordingly.

scale_bios.pjt	.bios section in external memory
scale_text.pjt	.text section in external memory
scale_bios_text.pjt	Both .bios and .text sections in external memory

18.3 Execute

The scale application illustrates a very simple message + data transfer scenario:

18.3.1 Copying files to target file system

The generated binaries on the GPP side and DSP side and the data files must be copied to the target directory.

GPP Side

For executing the DEBUG build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/scalegpp
../montavista/target/opt/dsplink/samples/scale/DEBUG
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

For executing the RELEASE build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/scalegpp
../montavista/target/opt/dsplink/samples/scale/RELEASE
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

- Enter the commands shown above in single line.

DSP Side

The DSP binaries are built on the Windows host. These binaries must be copied into the target file system. Any FTP client can be used for this transfer.

For executing the DEBUG build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/scale/DEBUG

   C:\dsplink\dsp\src\samples\scale\OMAP\Debug\scale.out

For executing the RELEASE build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/samples/scale/RELEASE

   C:\dsplink\dsp\src\samples\scale\OMAP\Release\scale.out

18.3.2 Loading the kernel module: dsplinkk.o

To load the device driver, login as 'root' and enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink load [.]
```

- If the argument . (dot) in the command is omitted, the kernel module located at */opt/dsplink/* is loaded.

This action generates a warning indicating that the kernel module does not contain the GPL license. This warning can be safely ignored.

18.3.3 Invoking the application

To invoke the application enter the following commands:

```
$ cd /opt/dsplink/samples/scale
$ ./scalegpp scale.out <buffersize> <iterations>
```

e.g.

```
$ ./scalegpp scale.out 128 10000
```

18.3.4 Unloading the kernel module: dsplinkk.o

To unload the device driver, enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink unload
```

G. TESTSUITE

19 Overview

The test suite provides a common shell to execute individual test cases. It is tolerant to different faults that occur during the execution of the test cases. It also ensures that testing can continue normally even after a test fails due to a major defect. Unrecoverable system failures are exceptions to this. It also provides the generic functions required for implementing various test suites.

19.1 API Tests

This test suite verifies the compliance to the documented API. It tests the API against valid and invalid arguments, basic data transfer (with data integrity), and operations that cause state changes.

19.2 Behavioral Tests

This test suite verifies the behavior through a series of typical usage scenarios. It exercises these scenarios in both single and multi-threaded environments.

19.3 Analysis Tests

This test suite measures the system performance through a raw single channel data transfer. The time for checking data integrity is excluded from the calculations.

The test is done for data transfer in synchronous and asynchronous modes with varying size of data buffers.

19.4 Stress Tests

This test suite measures the limits of the system under stress conditions. These conditions can be:

- Low resource availability
- Clients in multiple processes
- Data transfer from multiple concurrent threads
- Various timing delays to simulate application processing

20 Building the test suite

20.1 GPP Side

The procedure to build test suite is same as that of dsplink.

1. Build the sources using the steps described in section 11.
2. To build the test suite:

```
$ cd dsplink/gpp/src/test
$ gmake –s [debug | release]
```

3. Upon successful completion of build, `testsuite.out` shall be created in the directories:

   `~/dsplink/gpp/export/BIN/Linux/OMAP/DEBUG`

   `~/dsplink/gpp/export/BIN/Linux/OMAP/RELEASE`

20.2 DSP Side

A batch file is included in the release to build the DSP side testsuite. Follow the instructions below to build the DSP side testsuite:

1. Build the sources using the steps described in section 11.
2. To build the test suite:

```
L:\> cd dsplink\etc\host\scripts\msdos
L:\dsplink\etc\host\scripts\msdos> dspmake.bat OMAP [debug | release]
```

   Upon successful completion of build, binaries shall be created in the directories:

   `L:\dsplink\dsp\bin\test\OMAP\Debug`

   `L:\dsplink\dsp\bin\test\OMAP\Release`

- Please refer to the batch file for the command used to build an individual DSP side test suite executable for a specific build configuration.

21 Executing the test suite

The test suite expects the dsp binaries and data files used for driving the tests in following directory structure on the target:

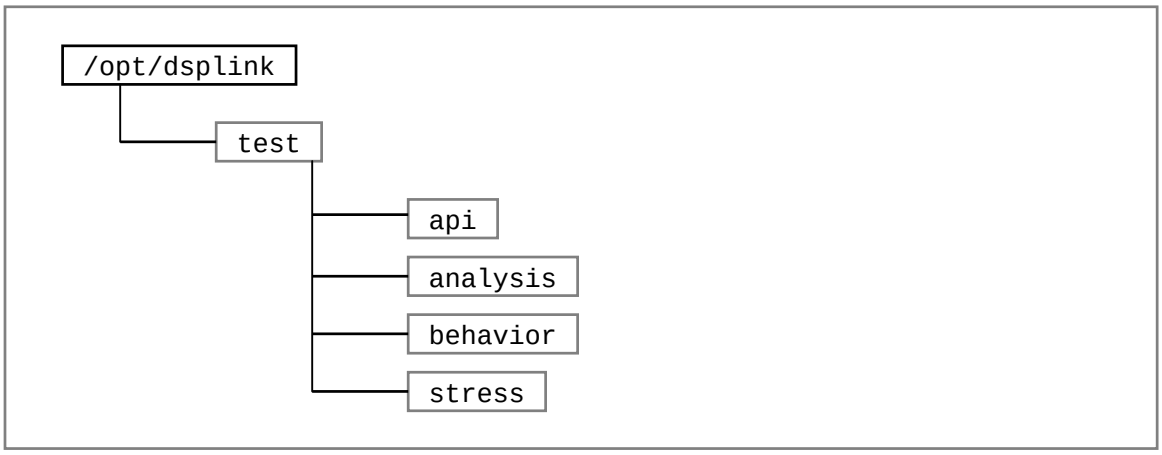

Figure 10. Directory structure for DSP binaries & data files expected by the test suite

21.1 Copying files to target file system

The generated binaries on the GPP side and DSP side and the data files must be copied to the target directory.

21.1.1 GPP Side

For testing the DEBUG build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/testsuite.out
../montavista/target/opt/dsplink/test/
$ cp gpp/export/BIN/Linux/OMAP/DEBUG/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

For testing the RELEASE build, follow the steps below to copy the relevant binaries:

```
$ cd ~/dsplink
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/testsuite.out
../montavista/target/opt/dsplink/test/
$ cp gpp/export/BIN/Linux/OMAP/RELEASE/dsplinkk.o
../montavista/target/opt/dsplink/
$ cp etc/target/scripts/Linux/dsplink ../montavista/target/opt/dsplink/
```

- Enter the commands shown above in single line.

21.1.2 DSP Side

The DSP binaries are built on the Windows host. These binaries must be copied into the target file system. Any FTP client can be used for this transfer.

For testing the DEBUG build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/test/api:

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivebuf.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\sendbuf.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivemsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\sendmsg*.out

2. FTP following files into the directory ~/montavista/target/opt/dsplink/test/analysis:

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivesendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\sendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\getputmsg*.out

3. FTP following files into the directory ~/montavista/target/opt/dsplink/test/behavior:

   C:\dsplink\dsp\bin\test\OMAP\Debug\sendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivebuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivesendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\endofstream*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\multichnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\reclaimtimeout*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\flushtest*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\idletest*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\siodeletetest*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\echomsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\locatetimeoutmsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\sendmsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivemsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\suppliermsg*.out

4. FTP following files into the directory ~/montavista/target/opt/dsplink/test/stress:

   C:\dsplink\dsp\bin\test\OMAP\Debug\multichnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\multitaskchnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Debug\receivemsg*.out

For testing the RELEASE build:

1. FTP following files into the directory ~/montavista/target/opt/dsplink/test/api:

   C:\dsplink\dsp\bin\test\OMAP\Release\receivebuf.out

   C:\dsplink\dsp\bin\test\OMAP\Release\sendbuf.out

   C:\dsplink\dsp\bin\test\OMAP\Release\receivemsg*.out

C:\dsplink\dsp\bin\test\OMAP\Release\sendmsg*.out

2. FTP following files into the directory ~/montavista/target/opt/dsplink/test/analysis:

   C:\dsplink\dsp\bin\test\OMAP\Release\receivesendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\sendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\getputmsg*.out

3. FTP following files into the directory ~/montavista/target/opt/dsplink/test/behavior:

   C:\dsplink\dsp\bin\test\OMAP\Release\sendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\receivebuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\receivesendbuf*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\endofstream*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\multichnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\reclaimtimeout*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\flushtest*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\idletest*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\siodeletetest*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\echomsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\locatetimeoutmsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\sendmsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\receivemsg*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\suppliermsg*.out

4. FTP following files into the directory ~/montavista/target/opt/dsplink/test/stress:

   C:\dsplink\dsp\bin\test\OMAP\Release\multichnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\multitaskchnlstress*.out

   C:\dsplink\dsp\bin\test\OMAP\Release\receivemsg*.out

21.2 Loading the kernel module: dsplinkk.o

To load the device driver, login as 'root' and enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink load [.]
```

- If the argument . (dot) in the command is omitted, the kernel module located at */opt/dsplink/* is loaded.

This action generates a warning indicating that the kernel module does not contain the GPL license. This warning can be safely ignored.

21.3 Invoking the test suite

There are two modes in which the test suite works.

SINGLE

To invoke a single test only

```
$ cd /opt/dsplink/test
$ ./testsuite.out single <testsuite_name> <testcase> <absolute path of datafile>
```

The mapping between the test suite to be executed and the corresponding command line argument is shown below:

Test suite to be executed	Argument on command line
API	LINKAPITEST
BEHAVIOR	LINKBEHAVIORTEST
ANALYSIS	LINKANALYSISTEST
STRESS	LINKSTRESSTEST

- The testsuite_name can be specified in any case. Internally the names are converted to consistent casing before checking.

For example to execute test case - API_ProcLoad - from the API test suite execute the following commands:

```
$ cd /opt/dsplink/test
$ ./testsuite.out single LINKAPITEST API_ProcLoad /opt/dsplink/test/api/api_procload.dat
```

- Enter the commands shown above in a single line.

SCRIPT

In contrast to single mode, the script mode invokes all the tests specified in a test suite.

To invoke multiple tests through a script:

```
$ cd /opt/dsplink/test
$ ./testsuite.out script <absolute path of script file>
```

To execute api testsuite run the following command:

```
$ cd /opt/dsplink/test
$ ./testsuite.out script /opt/dsplink/test/api/all_api.tsc
```

To execute behavior testsuite run the following command:

```
$ cd /opt/dsplink/test
$ ./testsuite.out script /opt/dsplink/test/behavior/all_bvr.tsc
```

To execute analysis testsuite run the following command:

```
$ cd /opt/dsplink/test
$ ./testsuite.out script /opt/dsplink/test/analysis/all_ana.tsc
```

To execute stress testsuite run the following command:

```
$ cd /opt/dsplink/test
$ ./testsuite.out script /opt/dsplink/test/stress/all_stress.tsc
```

- Enter the commands shown above in a single line.

21.4 Unloading the kernel module: dsplinkk.o

To unload the device driver, enter following commands on the command prompt.

```
$ cd /opt/dsplink
$ ./dsplink unload
```

H. APPENDIX

22 Issue reclaim model

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

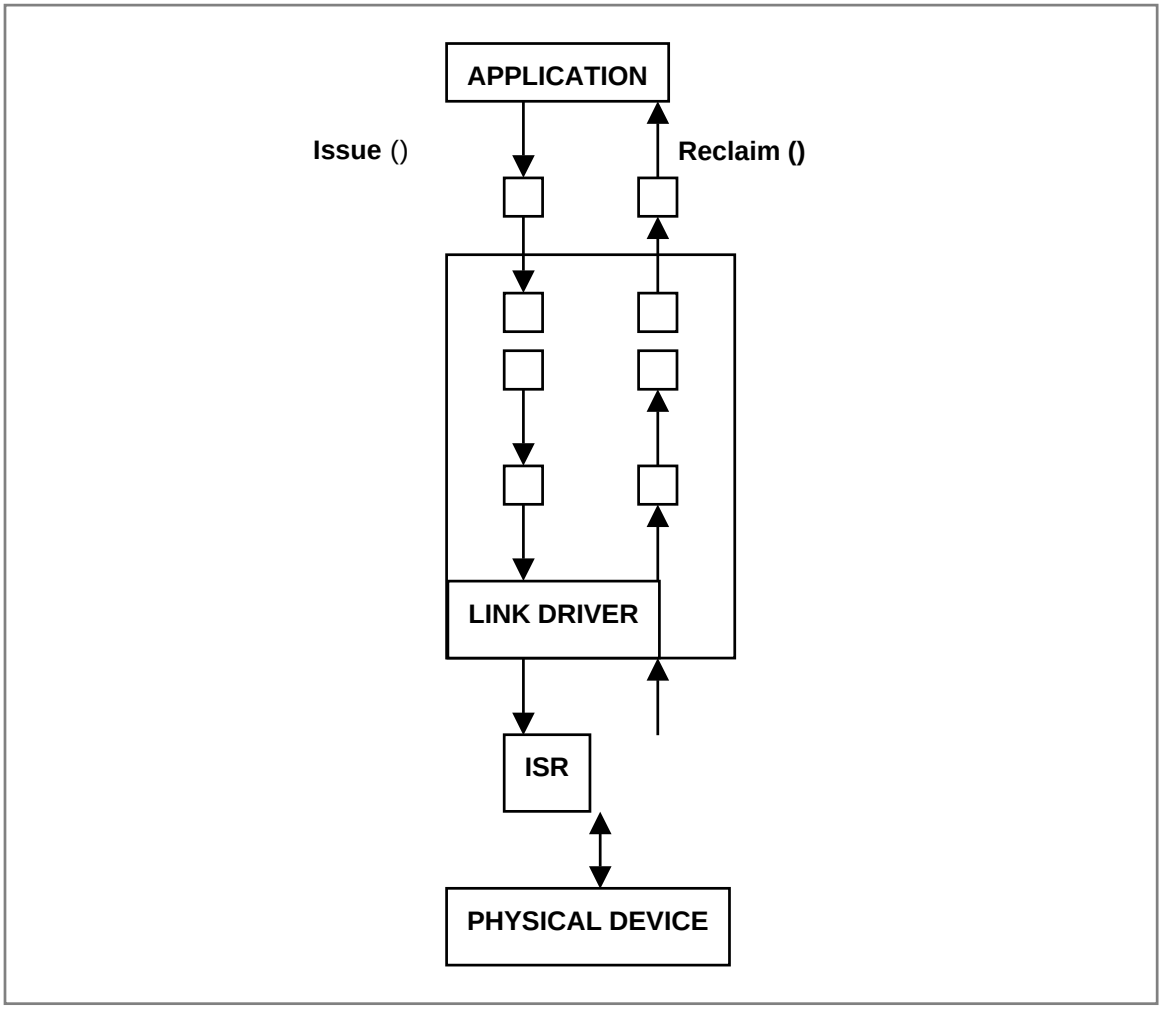

Figure 11. Issue Reclaim Model

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.

23 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:

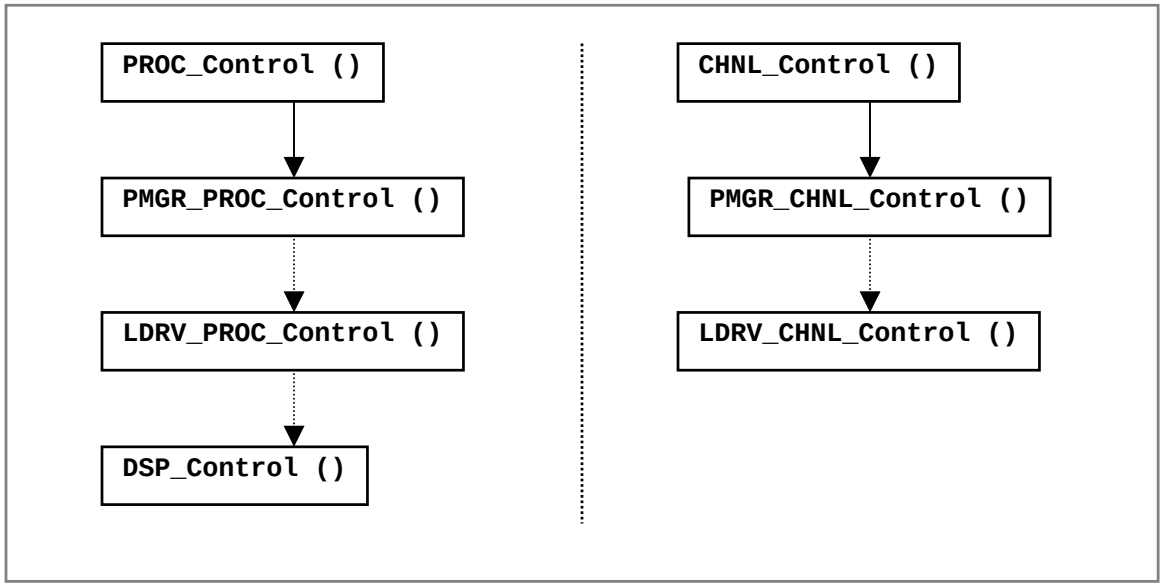

Figure 12. Execution flow: PROC_Control () and CHNL_Control ()

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.

24 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.

24.1 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) ;
```

24.2 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> ;
```

25 Debugging Applications

25.1 On the GPP side

25.1.1 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 subcomponents of LDRV with a severity allowing LEVEL2 and above trace prints to display.

```
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.

25.1.2 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, 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.

25.1.3 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_IO_Initialize () ;
if (DSP_FAILED (status)) {
    SET_FAILURE_REASON ;
}
```

25.2 On the DSP side

The DSP side can be debugged using CCS. However, both ARM and DSP must be 'running free' before ARM is booted up with Linux:

1. Start CCS immediately after switching on the board.
2. Put the GPP and DSP both in run free mode using the GEL files supplied with the release.
   - Use GEL function `Startup_RunFree ()` on DSP, and
   - Use GEL function `Reset_RunFree ()` on ARM.
3. Boot up ARM with Linux.

Now the user application can be executed from the Linux command prompt. When the application calls `PROC_Start ()`, DSP starts execution from the entry point of the executable. For debugging, the DSP must be halted in a known state. The following two methods can be used for achieving this:

25.3 Stopping execution in main

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

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

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

25.4 SET_FAILURE_REASON

The SET_FAILURE_REASON macro included with the DSP side sources can be used to stop execution upon failure in debug builds.

When USE_CCS_BREAKPOINT 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, this macro expands to the infinite message mentioned in the previous section. The steps mentioned in the previous section can be used to proceed with debugging.

26 Configuring DSP/BIOS™LINK

26.1 GPP side

The behavior of DSP/BIOS™ LINK can be configured through various hardware and software parameters defined in a textual configuration file. 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

The configuration file follows a specific naming convention:

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

e.g. CFG_OMAP.TXT, CFG_DM310_DM642.TXT, …

26.1.1 DRIVER

This section contains fields that affect the overall configuration of the driver:

NAME	This field is used for information purposes only.
COMPONENTS	This field specifies the configuration under which DSP/BIOS™ LINK is built. However, this field is used for information purposes only.
QUEUE	This field specifies the number of buffers that can be simultaneously queued for transfer by the DSP/BIOS™ LINK driver.
LINKTABLES	This field specifies the number of tables used by DSP/BIOS™ LINK driver. The tables contain the information for the physical links between GPP and DSP.
MMUTABLES	This field specifies the number of MMU tables for the DSP(s).
NUMMQAS	This field specifies the number of allocators that are available for use by the MSGQ component.
NUMMQTS	This field specifies the number of transports that are available for use by the MSGQ component.
LOCALMQT	This field specifies the index of the transport used as the local MQT.

26.1.2 GPP

NAME	This field is used for information purposes only.
NUMDSPS	This field specifies number of DSPs available on the hardware platform and used by DSP/BIOS™ LINK.

26.1.3 DSP

NAME	This field 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.
EXECUTABLE	This field specifies the default executable for the DSP. This

	field is currently not used.
LOADER	This field specifies the function pointer interface to parse and load the DSP executable on the DSP.
LINKTABLES	This field specifies the index of the LINKTABLE section to use for accessing this DSP.
NUMLINKS	This field specifies the number of physical links between GPP and DSP. It also represents the number of entries in the LINKTABLE.
AUTOSTART	This field specifies whether the DSP can be auto-started. This field is currently not used.
RESETVECTOR	This field specifies the address of the reset vector of the DSP.
WORDSIZE	This field specifies the size of the minimum addressable unit on the DSP in bytes.
ENDIAN	This field specifies the default endianism of the DSP.
MMUFLAG	This field specifies whether DSP/BIOS™ LINK should enable the MMU on the DSP.
MMUTABLE	This field specifies the index of MMUTABLE to use for the DSP. This value is significant if MMUFLAG is set to TRUE.
MMUENTRIES	This field specifies the number of MMU entries to make for the DSP. It also represents the size of MMU table for the DSP.
INTERFACE	This field specifies the function pointer interface to access the services of the DSP subcomponent for this DSP.
MQTID	This field specifies the index of the transport used as the remote MQT for this DSP.

26.1.4 Link Tables

This section specifies the attributes of each of the physical link drivers to be used by DSP/BIOS™ LINK. There is one LINKTABLE section for each DSP. However, each LINKTABLE may contain more than one entry depending on the number of physical links between GPP and DSP.

NAME	This field is used for information purposes only.
ABBR	This field specifies short abbreviation for the link driver. Though not currently used, it will be used to pass configuration information to the DSP side components.
NUMCHANNELS	This field specifies the number of channels supported on the physical link.
BASECHANNELID	This field specifies the base value taken for the ID of channels available on the link driver.
MAXBUFSIZE	This field specifies the maximum size of buffer to allocate and use on the channels on the link.

INTERFACE	This field specifies the address of the interface function pointer table to use for communicating with the link driver.
ARGUMENT1	This field specifies argument 1 to the physical link driver. The significance of this argument depends on the implementation of the link driver.
ARGUMENT2	This field specifies argument 2 to the physical link driver. The significance of this argument depends on the implementation of the link driver.

26.1.5 MMU Tables

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

ADDRVIRTUAL	Virtual address of the memory as seen by the DSP
ADDRPHYSICAL	Physical address of the memory mapped
SIZE	Size of the memory mapped
ACCESS	Access rights to the memory mapped (read, write, all)
PRESERVE	Whether the MMU entry needs to be preserved
MAPINGPP	Indicates whether the memory area needs to be mapped to GPP address space.

26.1.6 MQA

NAME	This field is used for debugging purposes only.
INTERFACE	This field specifies the address of the interface function pointer table for using this MQA.

26.1.7 MQT

NAME	This field is used for debugging purposes only.
INTERFACE	This field specifies the address of the interface function pointer table for using this MQT.
LINKID	This field specifies the ID of the physical link used by this MQT.

26.2 DSP side

DSP side configuration is done through the TCF file. The anatomy of a typical TCF file is shown below:

```
load(utils.findFile('assert.tci'));                          Line 1
load(utils.findFile('dsplink-omap-base.tci'));               Line 2
load(utils.findFile('dsplink-omap-dio.tci'));                Line 3
// Application Specific Configuration goes here              Line 4
prog.gen("app.cdb");                                         Line 5
```

```
assert.check();                                                    Line 6
```

Here is the description of each statement listed below:

Line 1	…	Loads the support for assertions in the TCF file.
Line 2	…	Loads the base configuration for DSP/BIOS™ LINK for OMAP.
Line 3	…	Loads the configuration specific to the class driver for DSP/BIOS™ LINK. 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 is a comment. It is placeholder for any application specific configuration.
Line 5	…	This statement is an instruction to generate a CDB file – `app.cdb`. The generated CDB file is built along with the application sources.
Line 6	…	This statement is an instruction to check for assertions while the TCF file is being generated.

- The files – dsplink-omap-base.tci and dsplink-omap-dio.tci – included in the illustration above are representative of real files.
- Do not change the 'tci' files for any application specific configuration.

27 Understanding MAKE system on GPP

27.1 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 compile sources for all GPP side operating systems and platforms.

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) 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

27.1.1 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)/gpp/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 corresponding
#           COMPONENT file.
#
#   @ver    01.00
```

```
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2002
#
#   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.
#   ============================================================================



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

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



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

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

- 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.

27.1.2 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, DRV or EXE.
COMP_TARGET	Name of the target file generated when the component is built.
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.

A sample COMPONENT file is listed below:

```
#   ============================================================================
#   @file   COMPONENT
#
#   @path   $(DSPLINK)/gpp/src/osal/Linux
#
#   @desc   This file contains information to build a component.
#
#   @ver    01.00
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2002
#
#   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       := $(PROJROOT)$(DIRSEP)src$(DIRSEP)osal
COMP_TYPE       := DRV
COMP_TARGET     := osal.o


#   ============================================================================
#   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)drv_os.h


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

USR_CC_FLAGS    :=

USR_CC_DEFNS    := -D__KERNEL__ -DMODULE -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     :=
```

27.1.3 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    01.00
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2002
#
#   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 :=  ldrv.c

ifeq ($(USE_PROC), 1)
SOURCES +=  ldrv_proc.c
endif

ifeq ($(USE_CHNL), 1)
SOURCES +=  ldrv_chnl.c    \
            ldrv_io.c

ifeq ($(PLATFORM), OMAP)
SOURCES += shm.c
endif

ifeq ($(PLATFORM), DM310_DM642)
SOURCES += hpi_driver.c
endif
endif

ifeq ($(USE_MSGQ), 1)
SOURCES +=  ldrv_msgq.c  \
            mqabuf.c     \
            lmqt.c       \
            rmqt.c
endif
```

27.1.4 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    01.00
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2002
#
#   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
#   ============================================================================

DIRS  =     \
    gen     \
    osal    \
    ldrv    \
    pmgr    \
    api
```

27.2 Common tasks

27.2.1 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.

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 file.

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

27.2.2 Changing the Compiler

The variable COMPILER (defined in the file osdefs.mk for the default OS distribution, and the file <distribution>.mk for a specific distribution) represents the fully qualified path to the compiler used.

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 following variables based on the switches supported by the new compiler:

CC_SW_DEF

CC_SW_INC

CC_SW_OBJ

CC_SW_DEB

STD_INC_PATH

STD_CC_FLAGS

EXE_CC_FLAGS

DRV_CC_FLAGS

LIB_CC_FLAGS

STD_CC_DEFNS

COMPILER_DEB

COMPILER_REL

If specific compiler flags were added in any of the COMPONENT file(s) then following variables should also be updated accordingly.

USR_CC_FLAGS

27.2.3 Changing the Archiver

The variable ARCHIVER (defined in the file osdefs.mk for the default OS distribution, and the file <distribution>.mk for a specific distribution) represents the fully qualified path to the linker used.

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 following variables based on the switches supported by the new archiver:

STD_AR_FLAGS

ARCHIVE_DEB

ARCHIVE_REL

27.2.4 Changing the Linker

The variable LINKER (defined in the file osdefs.mk for the default OS distribution, and the file <distribution>.mk for a specific distribution) represents the fully qualified path to the linker used.

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 following variable(s) based on the switches supported by the new linker:

LD_SW_LIB

LD_SW_OUT

LD_SW_RELOC

STD_LIB_PATH

STD_LD_FLAGS

EXE_LD_FLAGS

DRV_LD_FLAGS

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

If specific linker flags were added in any of the COMPONENT file(s) then following variables should also be updated accordingly.

USR_LD_FLAGS

27.2.5 Supporting a new distribution

It is possible that you may be required to support more than one distributions of a GPP OS. This possibility exists when more than one port of the OS is available e.g. in case of Linux.

In such situations, it is difficult 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.

```
#   ============================================================================
#   @file   distribution.mk
#
#   @path   $(MAKEROOT)\gpp\make\GPPOSA\PLATFORMX
#
#   @desc   This makefile defines OS specific macros used by MAKE system.
#
#   @rev    00.05
#   ============================================================================
#   Copyright (c) Texas Instruments Incorporated 2002
#
#   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      := C:\GPPOSA


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

You will also be required to update the build configuration file – `dsplinkcfg.opt`.

The addition of GPL distribution of Linux is illustrated below:

```
#   ============================================================================
#   OPT_GPPOS
#
#   List operating systems running on GPP OS.
#
#   Each element in this list is an array of length 3.
```

```
#       - 1st element is the value assigned to the build variable: GPPOS.
#       - 2nd element is the value assigned to the build variable: DISTRIBUTION.
#       - 3rd element is the name of the OS for display on the menu for
#         selecting the GPP OS.
#   ============================================================================

@OPT_GPPOS      = (
    [
        "",
        "",
        ""
    ],
    [
        "Nucleus",
        "",
        "Nucleus",
    ],
    [
        "Linux",
        "montavista",
        "Montavista Linux",
    ],
    [
        "Linux",
        "gpl",
        "GPL Linux",
    ]
) ;
```

I. FREQUENTLY ASKED QUESTIONS

28 Generic

Q: Why does the code below result in incorrect behavior?

```
if (DSP_SUCCEEDED (MSGQ_Create (GPPMSGQID, NULL))) {
    ...
}
```

A: DSP_SUCCEEDED(x) and DSP_FAILED(x) are macros.

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

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

```
status = MSGQ_Create (GPPMSGQID, NULL) ;
if (DSP_SUCCEEDED (status)) {
    ...
}
```

Q: Why does the DSP component have two APIs for setting up the DSP (DSP_Setup () & DSP_Initialize ()), but only a single API for finalization (DSP_Finalize ())?

A: DSP_Setup () makes the DSP accessible to the ARM, following which DSP_Initialize () actually configures the DSP. DSP_Finalize () frees up any resources acquired by DSP_Initialize () and places the DSP in a known state.

However, it is not required to make the DSP "not accessible" to the ARM in the end, hence there is no DSP_Destroy () API to correspond with the DSP_Setup () API.

29 Build Issues

Q: When I try to open the DSP side loop application project from CCS, it asks me for the loop.cdb and .cmd files. I can't seem to find these anywhere.

Am I doing something wrong?

A: These errors can be ignored. The CDB and CMD files are generated from the TCF file as part of the pre-build step.

Also, note that building the project file (.pjt) through timake.exe the recommended method for building DSP side.

30 Shared Memory Driver

Q: What is the priority of the data transfer channels?

A: The data transfer channels do not have any priority. Transfers are scheduled in a round-robin manner.

Q: Is it possible to change the priority of channels?

A: No. Not without changing the implementation.

Q: Is it true that only one buffer can reside in the shared memory output data region at a time (and similarly for the input region)?

A: Yes.

Q: How does LINK handle endianism of code/data on OMAP?

A: LINK currently supports only the NO_SWAP mode on OMAP. The endianism block on OMAP in the Traffic Controller is not modified by LINK and is left in its default state (no-swap). Please refer to the OMAP TRM for details on this block. Modifications of the TC block may have impact on code-loading and integrity of data buffers in the external memory. The placeholders for endianism conversion are maintained for future enhancements.

Q: Why is the maximum buffer size to be transferred restricted to 16K bytes in the link configuration, when the input and output region in shared memory is so large?

A: 16K is a sample configuration for the shared memory driver. It can be modified in the driver configuration as per need. It is only limited by the size of shared memory that has been reserved.

Q: The shared memory driver copies data from the user buffer to the shared memory area. How much data is copied from the buffer if the actual data is smaller than the size of the buffer?

A: The caller is expected to specify the actual size of data in the buffer as an argument to the `CHNL_Issue ()` API. When transferring this data buffer, this size is used during copies from/ to the shared memory area.

Q: What happens if I issue a buffer of size different than the one expected on the other processor?

A: The link driver takes the minimum of the two buffer sizes, and transfers the buffer of this size. The buffer contents thus get truncated to the size of the smaller buffer. In debug build, an assertion fails in such a scenario, indicating this.

Q: Can the size of shared memory area reserved for communication between GPP and DSP be reduced?

A: In the default configuration, during the OS boot up 1MB memory is reserved for the shared memory link driver for DSP/BIOS™ LINK.

The minimum size required for this shared memory area is dependent on the maximum size of data buffers that need to be supported on a channel.

e.g. the minimum size required for supporting 16KB data buffers is: 24 bytes + 2 * 16KB. Here 24 bytes are reserved for control information and 16KB areas are required for the input and output buffers.

The current implementation of the shared memory link driver implementation leaves a hole of approx .5MB between the input buffer area and the output buffer area.

If exact application requirements are known, this hole can be removed and in such a case the reserved memory area size will match the actual memory area used. Refer to the document "Shared Memory IOM Driver For OMAP 5910/5912" (LNK_019_DES) for details on this computation.

The shared memory driver files on GPP and DSP, the LINK configuration (linkcfg.txt) and the base `.tci` files need to be modified according to your configuration. The specific modifications required are as follows:

GPP Side sources

1. linkcfg.txt:

The section [LINKTABLE0] needs to be modified as follows:

§ ARGUMENT1 should be modified to have the starting address of the shared memory area (as seen from GPP).

§ MAXBUFSIZE should be modified according to the configuration you need.

The section [MMUTABLE0] entry [0]:

§ ADDRPHYSICAL should be modified to have the physical address of the shared memory area. This is mapped to virtual address 0x200000 for the DSP using DSP MMU.

2. shm.c:

§ Modify the pointer computations for output buffer such that it is adjacent to the input buffer. The maximum buffer size specified in linkcfg.txt should be used. `linkAttr->maxBufSize` gives this default value.

DSP Side sources:

1. dsplink-omap-base.tci:

§ The segment SHMMEM is used to reserve the shared memory area. Modify its definition according to the configuration specified in linkcfg.txt.

2. shm.c:

§ Modify definition of `SHM_BASE`, `SHM_LEN` and `SHM_BUFFER_LEN` according to the values specified on the GPP side.

31 COFF Loader

Q: Can the COFF loader be replaced with another loader?

A: Yes. DSP/BIOS™ LINK allows a new loader to be plugged into the system. The loader needs to specify its interface through a '`LoaderInterface`' data structure. Refer to "DSP Executable Loader" (`LNK_040_DES`) design document for more details.

32 Messaging using MSGQ

Q: What is the maximum size of messages that can be exchanged using

	DSPLINK messaging?
A:	The user configures the maximum message size that the remote MQT can transfer. The user must ensure that the size of messages transferred between the GPP and DSP is less than or equal to the MQT maximum message size. The maximum message size is limited by what is supported by the physical link between the two processors.
Q:	Are the message queues bidirectional?
A:	The message queues are unidirectional. They are created on the receiving side. Senders locate the queue to which they wish to send messages. The queues may be distributed across several processors.
Q:	Can multiple threads receive messages on the same message queue?
A:	Multiple threads/processes must not receive messages on the same MSGQ. Only a single thread/process owns the local MSGQ for receiving messages. However, multiple threads/processes may send messages to the same message queue.
Q:	Can the message queue be used to send variable sized messages?
A:	Variable sized messages can be sent using the DSPLINK messaging component. The message must contain the fixed message header as the first element. However, the rest of the message may be variable-sized.
Q:	Does the messaging and data transfer have same priority?
A:	Messages are transferred by DSPLINK across the physical link at a higher priority than data buffers.
Q:	Is it a requirement for a client to create a MSGQ when sending messages?
A:	No, only a client that receives messages must create a MSGQ.
Q:	Is the message transfer via MSGQ synchronous or asynchronous?
A:	Unlike the data transfer channels where the client is waiting for data to arrive on a designated channel, the message transfer is asynchronous. The messages may be used to intimate occurrence of an error, change in state of the system, a request based on user input, etc.
Q:	Can the same message queue names be used across processors?
A:	The message queue names must be unique over the complete system. This includes message queues created across all processors in the system.
Q:	Is there a fixed format for the messages in the MSGQ?
A:	The messages must have a fixed header as their first field. This header is used by the messaging component for including information required for transferring the message. The contents of the message header are reserved for use internally within DSPLINK and should not directly be modified by the user.
Q:	How is the physical data link between GPP and DSP selected for messaging?
A:	The messaging component can utilize any physical data links between GPP and DSP. This can be configured through the static configuration system.

	Messaging can be configured either to use the same physical link as used by the channels, or a different physical link for each processor.
Q:	Does the messaging component provide the feature to send acknowledgement on reception of the message?
A:	While sending a message, the user can choose to specify a reply MSGQ. On receiving the message, the receiver may extract information about the reply MSGQ from the message, and use it for replying to the received message.
Q:	Can messaging be scaled out of DSPLINK if only data transfers are required?
A:	DSPLINK is scalable to allow the users to scale out only the messaging component, or both the messaging and channel components.
Q:	What are control messages?
A:	Control messages are used internally by the remote transport for exchanging control information between the two processors. For example, control messages are used by the transport during MSGQ_Locate (). The user must also configure the allocator with buffers of the control message size expected by the transport.
Q:	Why does the local transport need to be opened in addition to the remote transport when messages are exchanged only with remote processor?
A:	For exchanging messages with the remote processor, the remote transport must always be opened. The local transport manages local queues, and hence needs to be opened whenever a queue is created for receiving messages. If the application only needs to send messages to the remote processor, it does not need to open the local transport.
Q:	Can a client on GPP use DSPLINK messaging for sending messages to a MSGQ residing on the same processor?
A:	No, local messaging on GPP is currently not supported.
Q:	Can a client allocate messages of any size less than maximum configured for the allocator?
A:	The allocator provided within DSPLINK is a fixed size allocator. The client will not be able to allocate messages of sizes that are not configured within the allocator. Also the minimum size of the message to be allocated must be greater than or equal to the size of the fixed message header.