Programming Guide for DSP/BIOS™ Bridge

Making**Wireless**

Table of Contents

List of Figures

List of Tables

Please read the “Important Notice” on the next page.

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.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

1 Products		2 Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

1 Introduction

DSP/BIOS Bridge enables asymmetric multiprocessing on target platforms which contain a general-purpose processor (GPP) and one or more attached DSPs. DSP/BIOS Bridge is a combination of software for both the GPP Operating System (OS) and DSP OS that links the two operating systems together. This linkage enables applications on the GPP and the DSP to easily communicate messages and data in a device-independent, efficient fashion.

For the GPP OS, DSP/BIOS Bridge adds an Application Programming Interface (API), which enables multiple clients, (i.e., users of the DSP), to simultaneously make use of the resources on the DSP.

GPP OS applications or device drivers can use the API to:

- initiate signal processing tasks on the DSP,
- exchange messages with DSP tasks,
- stream data buffers to and from DSP tasks,
- pause, resume, and delete DSP tasks, and
- perform resource status queries.

For the DSP side, DSP/BIOS Bridge adds a DSP-side API, which enables message exchange between the DSP and GPP as well as streaming of large blocks of data. DSP/BIOS Bridge also defines a DSP *node* abstraction that groups blocks of related code and data together into functional blocks called nodes. As GPP clients make GPP-side API calls, nodes will be appropriately created and started on the DSP, and then deleted when no longer needed. Nodes can be used for signal processing purposes as well as general control processing.

With the DSP node abstraction, the GPP and DSP-side APIs, and its underlying components, DSP/BIOS Bridge enables GPP application developers to utilize the DSP as a processing resource

1.1 Purpose

The purpose of this manual is to introduce DSP/BIOS Bridge programming concepts and procedures. The manual is organized in seven chapters:

Introduction includes a brief introduction to DSP/BIOS Bridge.

DSP/BIOS Bridge Architecture describes the components and Application Programming Interfaces provided by DSP/BIOS Bridge. It describes the partitioning of general-purpose processor and DSP software and provides a high-level summary of operation for a typical application.

Developing DSP/BIOS Bridge Applications summarizes the mechanics of building and running DSP/BIOS Bridge applications.

DSP/BIOS Bridge Nodes details the DSP node abstraction provided by DSP/BIOS Bridge. Included for each node type are descriptions of communication paths, execution timeline, function signatures, and execution context for each node function.

DSP/BIOS Bridge Configuration Database (DCD) describes procedures for configuring node objects in the DSP/BIOS Bridge Configuration Database and summarizes node deployment and node integration.

DSP/BIOS Bridge Examples describes how to build and run the example applications included in the Linux DSP/BIOS Bridge. The description for each example details application design concepts and implementation details.

Overlaying DSP/BIOS Bridge Nodes describes how developers can use DSP/BIOS Bridge to overlay the code and data for their DSP nodes at runtime. Included are details on configuring overlay sections, and three examples to illustrate different overlay scenarios.

Dynamically Loading DSP/BIOS Bridge Nodes describes how developers can use DSP/BIOS Bridge to load DSP nodes dynamically. Included are details on dynamic node creation, dynamic node debugging, and different examples to illustrate the use of dynamic nodes.

1.2 File Name

The file name of this document is ‘db_linux_pguide.doc’.

1.3 Naming Convention

Throughout the document some file names related to OMAP2430/3430 end with .*64P and OMAP2420 related files end with .*55L. The naming conventions are derived from the DSP processor family used in the platform. For example OMAP2420 has C55 based DSP processor, which is a 16-bit processor and OMAP2430/3430 has C64 based DSP processor, which is a 32 bit processor.

2 DSP/BIOS Bridge Architecture

2.1 Overview

DSP/BIOS Bridge is designed for platforms that contain a GPP and one or more attached DSPs. The GPP is considered the master or "host" processor, and the attached DSPs are processing resources that can be utilized by applications and drivers running on the GPP.

The abstraction that DSP/BIOS Bridge supplies is a direct link between a GPP program and a DSP task. This communication link is partitioned into two types of sub-links: *messaging* (short, fixed-length packets) and data *streaming* (multiple, large buffers). Each sub-link operates independently, and features in-order delivery of data, meaning that messages are delivered in the order they were submitted to the message link, and stream buffers are delivered in the order they were submitted to the stream link.

In addition, a GPP client can specify what inputs and outputs a DSP task uses. DSP tasks typically use message objects for passing control and status information and stream objects for efficient streaming of real-time data. A conceptual picture of the relationships between a GPP client program and a DSP task is shown in the following figure:

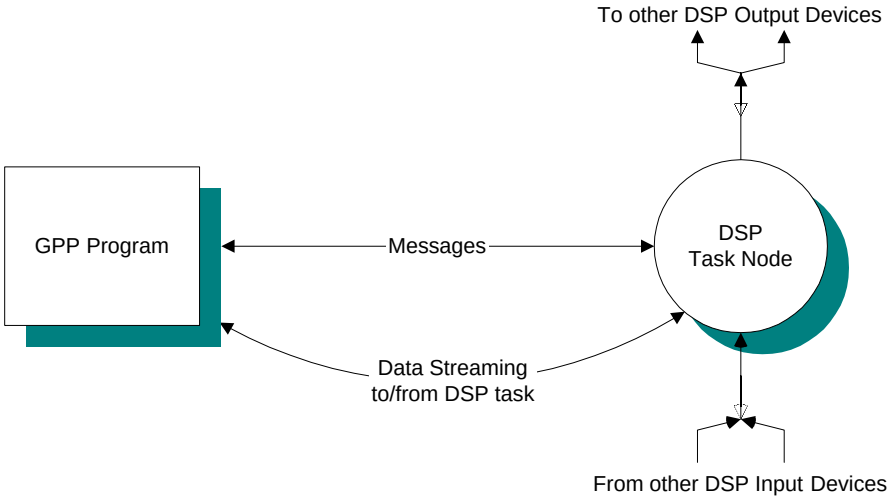

Figure 1 Relationship Between GPP Client and DSP Task

2.1.1 GPP Software Architecture

A GPP application communicates with its associated DSP task running on the DSP subsystem using the DSP/BIOS Bridge API. For example, a GPP audio application can use the API to pass messages to a DSP task that is managing data flowing from analog-to-digital converters (ADCs) to digital-to-analog converters (DACs).

From the perspective of the GPP OS, the DSP is treated as just another peripheral device. Most high level GPP OS typically support a device driver model, whereby applications can safely access and share a hardware peripheral through standard driver interfaces. Therefore, to allow multiple GPP applications to share access to the DSP, the GPP side of DSP/BIOS Bridge implements a device driver for the DSP.

Since driver interfaces are not always standard across GPP OS, and to provide some level of interoperability of application code using DSP/BIOS Bridge between GPP OS, DSP/BIOS Bridge provides a standard library of APIs which wrap calls into the device driver. So, rather than calling GPP OS specific driver interfaces, applications (and even other device drivers) can use the standard API library directly.

2.1.2 DSP Software Architecture

For DSP/BIOS, DSP/BIOS Bridge adds a devie-independent streaming I/O (STRM) interface, a messaging interface (NODE), and a Resource Manager (RM) Server. The RM Server runs as a task of DSP/BIOS and is subservient to commands and queries from the GPP. It executes commands to start and stop DSP signal processing nodes in response to GPP programs making requests through the (GPP-side) API.

DSP tasks started by the RM Server are similar to any other DSP task with two important differences: they must follow a specific task model consisting of three C-callable functions (node create, execute, and delete), with specific sets of arguments, and they have a pre-defined task environment established by the RM Server.

Tasks started by the RM Server communicate using the STRM and NODE interfaces and act as servers for their corresponding GPP clients, performing signal processing functions as requested by messages sent by their GPP client. Typically, a DSP task moves data from source devices to sink devices using device independent I/O streams, performing application-specific processing and transformations on the data while it is moved. For example, an audio task might perform audio decompression (ADPCM, MPEG, CELP) on data received from a GPP audio driver and then send the decompressed linear samples to a digital-to-analog converter.

2.2 Components

The functional components of DSP/BIOS Bridge are shown in the following figure:

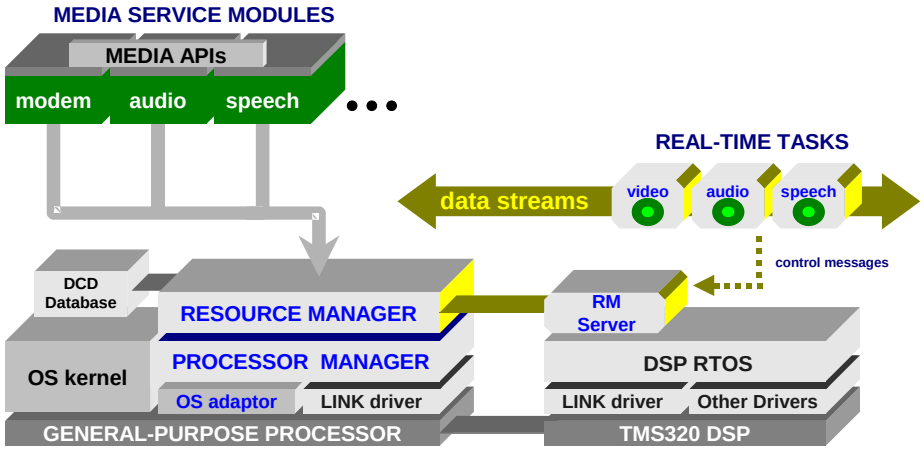

Figure 2 DSP/BIOS Bridge Components

In this typical configuration, a GPP is connected to one or more DSPs. End-user applications on the GPP make calls into media service modules or drivers, which are written to use the DSP/BIOS Bridge API to manage DSP resources.

The Resource Manager is responsible for dynamically instantiating DSP resources to meet allocation requests, monitoring DSP resources, dynamically loading DSP code as needed, and implementing policies for managing DSP resources when there are conflicting requests. The Resource Manager sits on top of the Platform Manager, which is responsible for statically loading a base code image to the DSP, starting and stopping the DSP, and implementing data streaming. The Platform Manager sits on top of a GPP OS adaptation layer and a DSP “link” Driver for low-level interprocessor communication with the DSP. A dynamic configuration database (DCD) stores configuration information as well as packaged DSP content for the platform.

On the DSP side, DSP/BIOS communicates with the GPP via a Host "link" Driver. On top of DSP/BIOS sits the Resource Manager (RM) Server. The primary duty of the RM Server is to dynamically create, execute, and destroy DSP processing nodes under Resource Manager control. Other duties include routing messages between the GPP and individual nodes, altering task priorities, and responding to Resource Manager configuration commands and status queries. A dedicated data stream is used to send commands from the Resource Manager to the RM Server, and another dedicated stream is used to send responses back to the Resource Manager.

DSP task nodes are separate execution threads running on the DSP that implement control or signal processing algorithms. Task nodes communicate with one another, and with the GPP, via short fixed-length messages and/or device-independent stream I/O.

2.3 Host Software

2.3.1 Overview

This section summarizes the GPP-side interfaces provided by DSP/BIOS Bridge. It describes the partitioning of functionality into separate modules (**DSPManager**, **DSPProcessor**, **DSPNode**, and **DSPStream**), includes processor, node, and stream state diagrams, and concludes with an operational flow that illustrates how a typical DSP/BIOS Bridge application will control processing nodes on a DSP.

2.3.2 DSPManager Module

The **`DSPManager`** Module is the highest-level module of the DSP/BIOS Bridge API and is primarily used to obtain DSP processor and manipulate node configuration information. The DSPManager APIs are summarized in the following table, and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 1 DSP Manager APIs

Function	Purpose
`DSPManager_Open`	Open handle to DSP/BIOS Bridge driver.
`DSPManager_Close`	Close handle to DSP/BIOS Bridge driver.
`DSPManager_EnumNodeInfo`	Enumerate and get information about nodes configured in the DSP/BIOS Bridge Configuration Database (DCD).
`DSPManager_EnumProcessorInfo`	Enumerate and get information about available DSP processors.
`DSPManager_RegisterObject`	Register object with the DCD.
`DSPManager_UnregisterObject`	Unregister object from the DCD.
`DSPManager_WaitForEvents`	Wait on one or more DSP/BIOS Bridge events.

2.3.3 DSPProcessor Module

The **DSPProcessor** Module is used to manipulate DSP processor objects, which represent particular DSP subsystems linked to the GPP. Processor objects are used to create, execute, and delete signal-processing nodes on a particular DSP subsystem.

The **DSPProcessor** APIs are summarized in the following table and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 2 DSP Processor APIs

Function	Purpose
DSPProcessor_Attach	Reserve (GPP-side) resources for a processor; get a handle for the processor
DSPProcessor_EnumNodes	Enumerate the processing nodes currently allocated for a DSP
DSPProcessor_GetResourceInfo	Get DSP resource information
DSPProcessor_GetState	Report the (execution) state of a DSP processor
DSPProcessor_RegisterNotify	Register API client to be notified on specific events
DSPProcessor_Detach	Free (GPP-side) resources dedicated to a processor
DSPProcessor_Load	Reset a processor and load a base program image
DSPProcessor_Start	Start a processor, running the code loaded by **DSPProcessor_Load**
DSPProcessor_Ctrl	Pass control information to the processor's GPP device driver
DSPProcessor_Stop	Stops a running Processor

As DSP/BIOS Bridge clients make **DSPProcessor** API calls, the corresponding DSP processors will transition between a set of pre-defined states. These states are shown in the following state diagram. Note that the API functions **DSPProcessor_Load** , **DSPProcessor_Start** and **DSPProcessor_Stop** will cause transitions to the **Loaded** , **Running** and **Stopped** states, respectively.

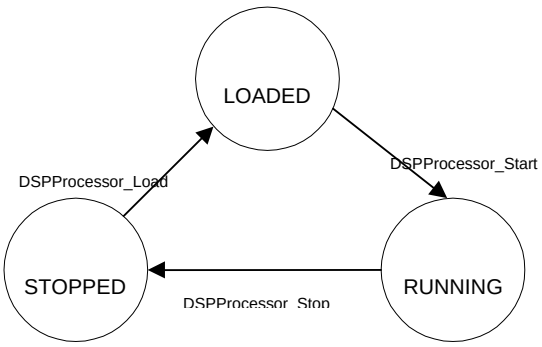

Figure 3 Processor State Diagram

2.3.4 DSPNode Module

The **DSPNode** module is used to manipulate node objects, which represent control and signal processing elements running on a particular DSP.

The **DSPNode** APIs are summarized in the following table, and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 3 DSP Node APIs

Function	Purpose
`DSPNode_Allocate`	Reserve (GPP-side) resources for a node; return a handle for the node
`DSPNode_Connect`	Connect two streams of two allocated (but not yet created) nodes
`DSPNode_ConnectEx`	Connect two streams of two allocated (but not yet created) nodes and additionally passes the connection parameter to DSP side
`DSPNode_Create`	Create a node on a DSP processor, and execute the node's create-phase function
`DSPNode_Run`	Start a node's execute phase; or resume execution of a paused node
`DSPNode_GetMessage`	Retrieve a message from a node (if available)
`DSPNode_PutMessage`	Send a message to a node
`DSPNode_AllocMsgBuf`	Allocate a buffer whose descriptor will be passed to a DSP node within a message
`DSPNode_FreeMsgBuf`	Free a buffer allocated with **`DSPNode_AllocMsgBuf`**
`DSPNode_RegisterNotify`	Register with API to be notified of specific events for this node
`DSPNode_Pause`	Temporarily suspend execution of a node
`DSPNode_Terminate`	Signal a node to tell it to exit its execute phase
`DSPNode_Delete`	Run the node's delete-phase function, and release all DSP-side and GPP-side resources for the node
`DSPNode_GetAttr`	Report the attributes of a node, including its execution state
`DSPNode_ChangePriority`	Change a node's runtime priority

As DSP/BIOS Bridge clients make **DSPNode** API calls, the corresponding nodes will transition between a set of pre-defined states, as shown in the following state diagram. Note that this description applies to task and XDAIS socket nodes (described later), but does not apply to device nodes (also described later).

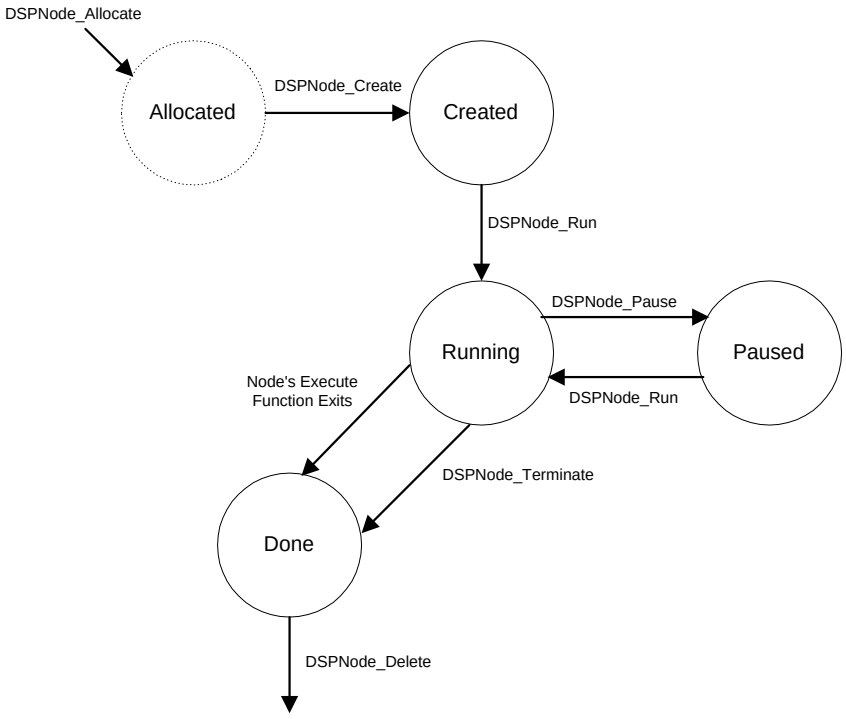

Figure 4 Node State Diagram

A node begins its life when a GPP client calls the API function **DSPNode_Allocate**. **DSPNode_Allocate** will allocate data structures on the GPP to allow control and communication with the node once it gets created on the DSP. In the **Allocated** state the node exists only on the GPP (denoted by a dashed circle

in the diagram) – it does not exist on the DSP yet. Once a node has been allocated on the GPP, its stream connections to other nodes can be setup via calls to **DSPNode_Connect**.

Once all stream connections for the node have been defined, the GPP client calls **DSPNode_Create** to create the node in a pre-run state on the DSP. Once the DSP task has been created, its create-phase function will be executed on the DSP. It is in the create-phase function that the DSP task can allocate any additional resources it needs for its execute phase.

When the GPP client calls **DSPNode_Run** the node will be launched into its execute phase by the RM Server on the DSP. In the **Running** state the node performs its runtime signal processing functions. The GPP client can temporarily suspend execution of a task or XDAIS socket node by calling **DSPNode_Pause**, (causing a transition to the **Paused** state), and then resume execution by calling **DSPNode_Run** again.

The node will transition to the **Done** state when it exits its execute-phase function. The exit can occur either because the node completes its processing, or because the GPP client calls **DSPNode_Terminate** to signal the node to exit.

When the GPP client calls **DSPNode_Delete** a command will be sent to the RM Server on the DSP to run the node's delete-phase function. Once the delete-phase function has run, all DSP-side and GPP-side resources for the node will be freed, causing the node to be destroyed.

2.3.5 DSPStream Module

The **DSPStream** module is used to manipulate stream objects, which represent logical channels for streaming data between the GPP and nodes on a particular DSP.

The **DSPStream** APIs are summarized in the following table and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 4 Stream APIs

Function	Purpose
`DSPStream_Open`	Open a stream, get a stream object handle
`DSPStream_GetInfo`	Get information about a stream
`DSPStream_AllocateBuffers`	Allocate data buffers for use with a stream
`DSPStream_FreeBuffers`	Release previously allocated data buffers
`DSPStream_PrepareBuffer`	Prepare a user-provided, pre-allocated buffer for use with a stream
`DSPStream_UnprepareBuffer`	Unprepare a user-provided buffer following use with a stream
`DSPStream_Issue`	Send a buffer to a stream
`DSPStream_Reclaim`	Request a buffer back from a stream
`DSPStream_Select`	Select a ready stream for I/O
`DSPStream_RegisterNotify`	Register with API to be notified of specific events on this stream
`DSPStream_Idle`	Idle a stream
`DSPStream_Close`	Close a stream, free the stream object

As DSP/BIOS Bridge clients make **DSPStream** API calls, the corresponding streams will transition between a set of pre-defined states, as shown in the following state diagrams.

The state diagram for a DSP→GPP input stream, (from an API-client perspective), is shown below. Stream state transitions occur based upon either a client making I/O requests via the API, or by low level I/O completion events.

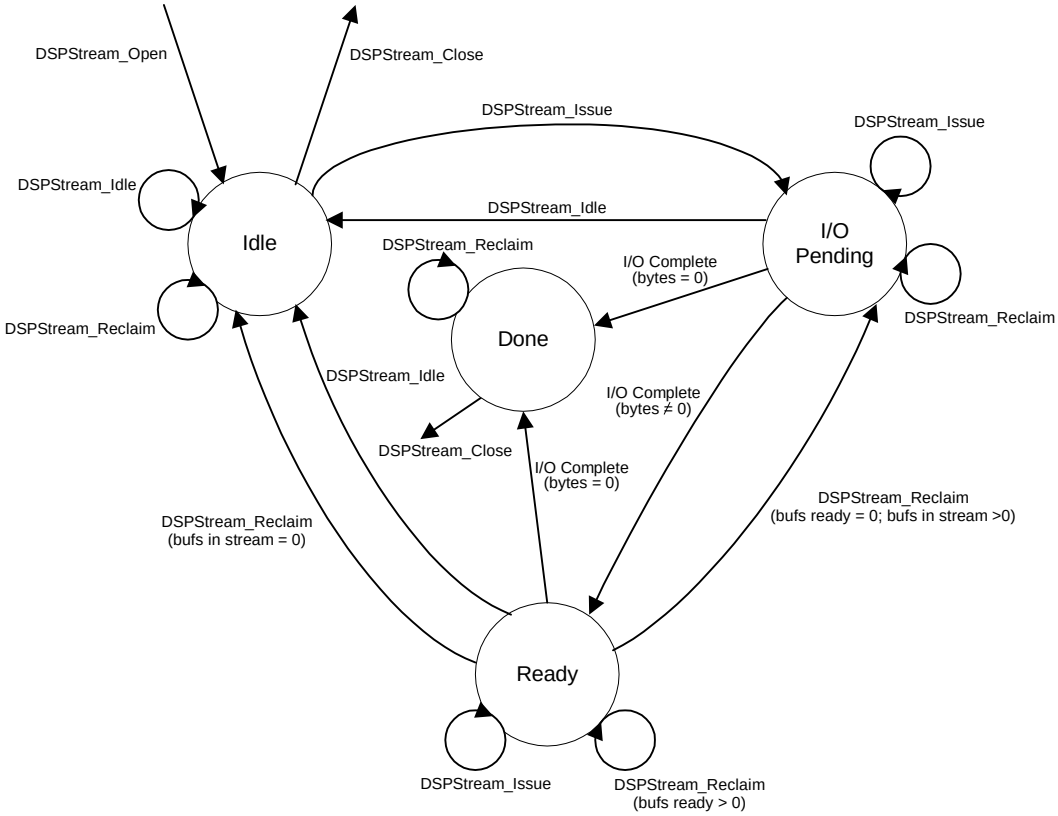

Figure 5 Input Stream State Diagram

A call to **DSPStream_Open** will create a stream object and return a handle to the object to the client. The stream is initially in the **Idle** state, with no pending I/O. The GPP client can then call **DSPStream_AllocateBuffers** to allocate data buffers for use with the stream. In some situations (e.g., another application passes buffers to the DSP/BIOS Bridge API client), the client will not allocate the buffers, but will prepare the pre-allocated buffers for the stream, using **DSPStream_PrepareBuffer**.

DSPStream_Issue is called to submit (i.e., enqueue), data buffers to the stream. Calling **DSPStream_Issue** causes a transition to the **I/O Pending** state.

When the underlying I/O operation completes the stream transitions from **I/O Pending** to the **Ready** state, signifying that the stream is ready to return at least one buffer to the client. For example, on an input stream the client submits an empty buffer to the stream via **DSPStream_Issue**; on I/O completion a buffer of data is ready, and the client can call **DSPStream_Reclaim** (without blocking) to retrieve the buffer. A call to **DSPStream_Reclaim** while the stream is in the **I/O Pending** state will cause the GPP client thread to block until I/O completion, or a timeout. A call to **DSPStream_Reclaim** while the stream is in the **Ready** state will cause a transition back to the **I/O Pending** state if, after retrieving a buffer, no more buffers are currently available to be reclaimed, (i.e., bufs ready = 0). If buffers are available, (bufs ready > 0), then the stream will remain in the **Ready** state. If **DSPStream_Reclaim** is called in the **Ready** state, and only one buffer is in the stream, then the reclaim will cause the stream to transition back to the **Idle** state, as there are no more pending I/O operations.

An “end of stream” event occurs when the input stream receives a zero-length data buffer from the DSP. When the low-level I/O operation completes, and the number of bytes in the transferred buffer is zero, the stream will transition to the **Done** state. **Done** is a terminal state; the only way to exit this state is via **DSPStream_Close**.

Calling **DSPStream_Idle** while in the **I/O Pending** or **Ready** states causes a transition back to the **Idle** state. If **DSPStream_Idle** is called on an input stream, any currently buffered input data will be discarded as part of the state transition, (i.e., all data buffers will be made ready to be reclaimed).

After **DSPStream_Idle** is called, data buffers that were enqueued to the stream via **DSPStream_Issue** can be reclaimed using **DSPStream_Reclaim**. **DSPStream_Reclaim** is allowed in the **Idle** state as long as there are buffers ready to be reclaimed. (**DSPStream_GetInfo** can be called to determine the number of buffers currently issued to the stream.)

Calling **DSPStream_Close** in the **Idle** state causes the stream to be closed and the stream object to be deleted.

The state diagram for a GPP→DSP output stream is shown in the following figure.

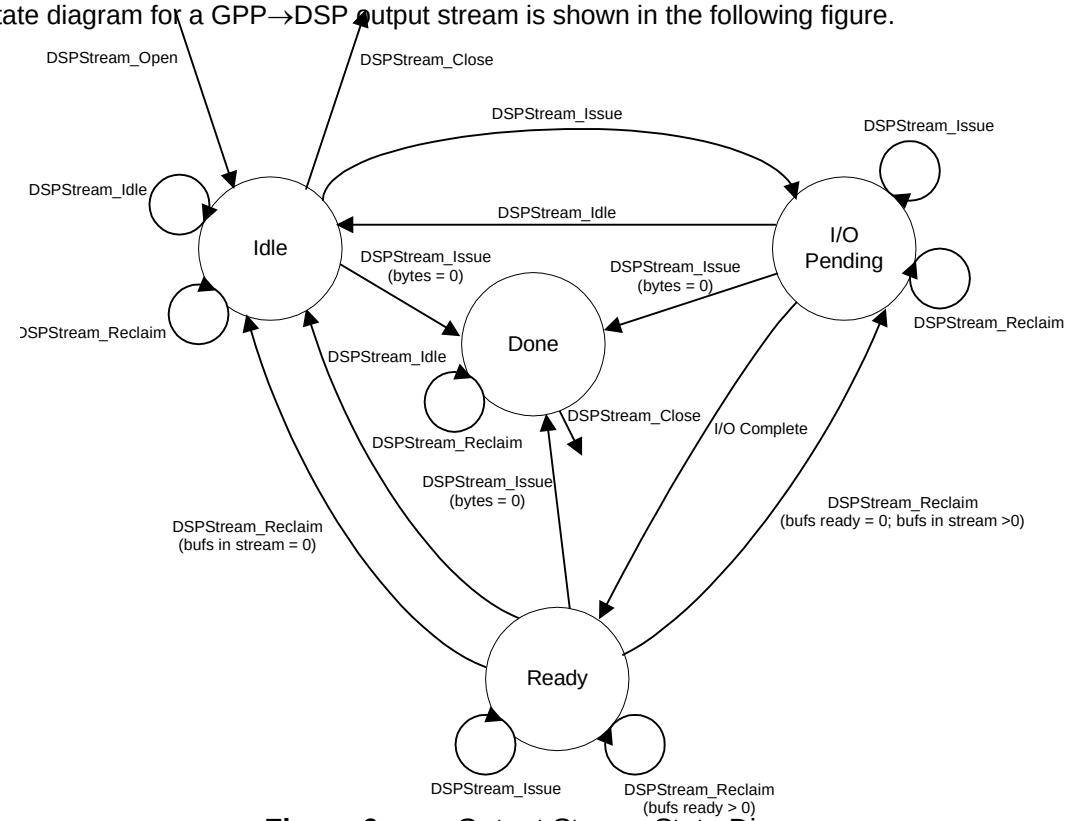

Figure 6 Output Stream State Diagram

For output streams the **Done** state is reached when a zero-length buffer is issued to the stream by the API client. Also, if **DSPStream_Idle** is called on an output stream the operation depends upon the **bFlush** argument for the function call. If **bFlush** is FALSE, any currently buffered data will be transferred through the stream, and GPP thread execution will be suspended for as long as required for the data to be consumed by the stream. If **bFlush** is TRUE, then any currently buffered data will be discarded, and no blocking will occur.

2.3.5.1 Stream Buffer Sizes

The two processors (GPP and DSP) that are linked together by DSP/BIOS Bridge may have different elementary data sizes. For example, on the OMAP platform the ARM has 8-bit bytes, and the DSP has a minimal addressable unit (MAU) data size of 16-bits. The DSP/BIOS Bridge APIs allow GPP-side code to specify buffer sizes in bytes, while the DSP-side of the APIs recognizes sizes in DSP MAUs. Because of this 2-to-1 bytes-to-MAUs ratio, GPP-side buffer allocations must be made in sizes that are an increment of DSP MAUs (i.e., an even number of bytes), otherwise the DSP-side may not be able to properly interpret the data sent from the GPP.

GPP-side buffer allocations must be in increments of DSP MAUs, otherwise results will be undefined.

2.3.6 Controlling Nodes

A GPP application can use DSP/BIOS Bridge's GPP-side API calls to create, launch, and shutdown processing nodes on the DSP. This section provides a high-level summary of the typical operational flow for an application. It shows how nodes are controlled from the GPP-side APIs and summarizes the corresponding actions that occur on both the GPP and DSP.

2.3.6.1 Boot Up

At system boot time the GPP OS will load a device driver for each DSP subsystem that is configured. As part of initialization, each device driver will be responsible for powering up the DSP, resetting it, loading a base code image to the DSP, and starting the DSP running. The device driver will also put the DSP in a low power state following initialization.

2.3.6.2 Select and Attach to a DSP

A GPP application or driver that wants to use a DSP first opens a handle to the DSP/BIOS Bridge driver using **DSPManager_Open**. Then it will call the API function **DSPManager_EnumProcessorInfo** to get information about available DSPs. The GPP client will then call **DSPProcessor_Attach** to reserve GPP-side resources for controlling and communicating with a particular DSP processor. **DSPProcessor_Attach** will return a handle to a DSP processor object for subsequent use in API function calls. Calling **DSPProcessor_Attach** does not affect the execution state of the DSP processor; it simply reserves GPP side resources for subsequent calls.

2.3.6.3 Allocate and Connect DSP Nodes

Once a DSP has been selected and attached to, the GPP client calls **DSPNode_Allocate** to allocate DSP nodes for the selected processor. **DSPNode_Allocate** reserves GPP-side resources for a node and returns a handle to the node data structure. Once a node is allocated on the GPP, its data streams can be connected to other streams of allocated nodes via **DSPNode_Connect**. **DSPNode_Allocate** and **DSPNode_Connect** do not affect the execution on the target DSP; node allocation and stream connection is performed on the GPP before any DSP interaction is performed.

2.3.6.4 Create Nodes on the DSP

Once a node has been allocated on the GPP, and all of its streams have been connected, **DSPNode_Create** can be called to create the node on the DSP processor. When **DSPNode_Create** is called for a task node the Resource Manager on the GPP will send commands to the RM server running on the DSP, which will: 1) allocate the node's default resources and create a task in DSP/BIOS, and 2)

execute the node's create-phase function to allocate any node-specific resources that are needed during the node's real-time execution phase. After task node creation all of the node's resources are allocated, and the node exists in DSP/BIOS, but the node is suspended in a (pre-run) inactive state.

2.3.6.5 Launch DSP Tasks

DSP task nodes are launched into their real-time execute phase when the GPP-side API client calls **DSPNode_Run** for the node. Internally, **DSPNode_Run** sends a command to the RM Server running on the DSP to change the priority of the task from (pre-run) inactive to the configured runtime priority of the task. Once the task node is running, the GPP client can stream data buffers to/from the task as well as exchange short messages with the task.

2.3.6.6 Stream Data to/from DSP Tasks

To stream data buffers to/from a DSP task, a GPP client must first call the API function **DSPStream_Open** to get a stream object handle. The GPP client then allocates data buffers for the stream by calling **DSPStream_AllocateBuffers**; alternatively, if the buffers are already pre-allocated (e.g., they were passed to the client by another GPP application), the GPP client prepares the buffers for the stream using **DSPStream_PrepareBuffer**. Once stream buffers are allocated and prepared, **DSPStream_Issue** can be used to submit filled data buffers to an output stream, or empty buffers to an input stream. The GPP client calls **DSPStream_Reclaim** to request an empty buffer back from an output stream or filled buffers back from an input stream. Submitting a data buffer to a stream (via **DSPStream_Issue**) will not block GPP thread execution, but requesting a buffer back from the stream (via **DSPStream_Reclaim**) may cause the GPP thread to block, pending I/O completion. **DSPStream_Select** can be used to block execution of a GPP thread until at least one of a specified set of streams has buffers ready for reclamation. **DSPStream_Idle** can be called to reset a stream, and **DSPStream_FreeBuffers** is used to release the stream buffers that were allocated via **DSPStream_AllocateBuffers**.

2.3.6.7 Exchange Messages with DSP Nodes

A GPP client can retrieve messages from a specific DSP node by calling **DSPNode_GetMessage** and can send messages to a specific node using **DSPNode_PutMessage**.

2.3.6.8 Terminate DSP Nodes

A GPP client can terminate execution of a DSP node by calling the API function **DSPNode_Terminate**. **DSPNode_Terminate** will cause the Resource Manager to send a shutdown message to the specific node. **DSPNode_Terminate** will block GPP thread execution until the node responds by exiting its execute-phase function.

2.3.6.9 Delete DSP Nodes

A GPP client calls **DSPNode_Delete** to delete a task node from a DSP. Calling **DSPNode_Delete** will cause the Resource Manager to send a command to the RM Server running on the specific DSP, which will: 1) run the node's delete-phase function (which should free all resources allocated by the node's create-phase function), and then 2) cleanup all (RM Server-created) DSP-side resources for the node. When the RM Server responds back to the Resource Manager, the Resource Manager will then delete all GPP-side resources for the node.

2.3.6.10 Detach from DSP

A GPP client calls **DSPProcessor_Detach** to release a DSP processor that it no longer needs. DSP processor objects are reference counted; when **DSPProcessor_Detach** is called by the last client using the processor, the processor object and all other GPP-side resources for the processor will be freed. **DSPProcessor_Detach** does not affect the execution state of the DSP processor; it allows a GPP client to release a processor and allows the Resource Manager to release all resources for the processor if no other GPP clients are attached to the processor. Finally, close handle to DSP/BIOS Bridge driver using DSPManager_Close.

2.4 DSP Software

2.4.1 Overview

This section summarizes the DSP-side application interfaces provided by DSP/BIOS Bridge. It describes: the partitioning of functionality into separate interface modules (**STRM**, and **NODE**), the Resource Manager Server, the DSP task model, and concludes with a DSP task node example.

2.4.2 STRM Module

The STRM module is used to manipulate stream objects, which represent logical channels for streaming data between nodes on the DSP and other nodes the same DSP, other nodes on different DSPs, or applications running on the GPP.

The STRM APIs are summarized in the following table and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 5 Stream APIs (DSP)

Function	Purpose
`STRM_create`	Open a stream
`STRM_allocateBuffer`	Allocate a data buffer for use with a stream
`STRM_control`	Perform a device-dependent control operation
`STRM_delete`	Close a stream
`STRM_freeBuffer`	Release a previously allocated data buffer
`STRM_idle`	Idle a stream
`STRM_issue`	Send a buffer to a stream
`STRM_reclaim`	Request a buffer back from a stream

2.4.3 NODE Module

The NODE module is used by a node running on the DSP to exchange messages with another node or with applications running on the GPP. It also provides functions that allow management of message buffers, and blocking of execution until a certain message or stream I/O event has occurred.

The NODE APIs are summarized in the following table and described in detail in the *DSP/BIOS Bridge Reference Guide*.

Table 6 Node APIs (DSP)

Function	Purpose
`NODE_getMsg`	Retrieve a message for this node, either from the GPP, or from another node
`NODE_putMsg`	Send a message from this node to another node, or to the GPP
`NODE_wait`	Wait until a message has arrived for this node, or a stream is ready
`NODE_allocMsgBuf`	Allocate buffer whose descriptor will be sent to the GPP as a message
`NODE_freeMsgBuf`	Free a buffer allocated with NODE_allocMsgBuf

2.4.4 Resource Manager Server

The Resource Manager (RM) Server is an internal component of DSP/BIOS Bridge. It runs as a task of DSP/BIOS and is subservient to commands and queries from Resource Manager running on the GPP. As GPP programs make requests through the (GPP-side) DSP/BIOS Bridge APIs, the Resource Manager will send commands to the RM Server on the DSP to implement the corresponding action.

The primary role of the RM Server is to dynamically create, execute, and destroy signal-processing nodes on the DSP at runtime. When it creates a node it also creates an "environment" for the node, which includes default resources for the node, such as an RTOS task for the node, a message queue and notification semaphore, etc., which the node will use during its execute phase. When the RM Server deletes a node it will free up the resources allocated during the node's create phase.

Additional RM Server duties include altering DSP node execution priorities, responding to Resource Manager configuration commands and status queries, and routing messages between the GPP and individual nodes. The RM Server uses a dedicated data stream to receive commands from the Resource Manager and a dedicated stream to send responses back to the Resource Manager.

2.4.5 DSP Task Model

The task model used in DSP/BIOS Bridge follows that provided by DSP/BIOS's TSK module (see *TMS320 DSP/BIOS User's Guide*). Tasks are independent execution threads. Each task has its own run-time stack, its own copy of processor registers, and an execution priority. Tasks can be pre-empted by higher priority tasks (or software or hardware interrupts), and can block their execution pending some I/O or synchronization event. The TSK scheduler manages the set of task objects, initiating and preempting tasks as appropriate.

DSP task nodes created by the RM Server are similar to "normal" DSP/BIOS tasks with a few important restrictions:

- They must follow a three-phase model (see below), providing three C-callable functions with specific arguments. To avoid name space collisions with other statically configured nodes, each node function should be prefixed by **`<NODE_NAME>_<VENDOR>_`**.
- All task node creation and deletion operations run in the context of another task (the RM server); only the node's real-time execute-phase is run as an independent thread (see the *DSP/BIOS Bridge Nodes* chapter below).
- Task nodes have a pre-defined task environment established by the RM Server.

The three-phase task model partitions node functionality in three blocks of code (three separate functions), corresponding to the major periods of the node's life cycle:

Create – all DSP resources needed by the node are allocated,

Execute – node's real-time processing is performed,

Delete – all resources allocated for the node are freed

Task state is maintained in a context structure that is passed from one task phase to another. For example, in its create phase a task will likely allocate memory buffers, create streams, etc. It will place these resource handles and other task-specific state information into a task-specific context data structure. This context structure will then be passed to the task's execute-phase, so that the allocated memory, streams, etc., can be used as the task does its meaningful work.

As the brunt of a task's code footprint is often due to setup, breaking the task's code into the separate phases, and coupling this with (future) dynamic loading/overlaying, enables the Resource Manager on the GPP to efficiently utilize DSP memory. Requiring all resource allocation to be done in the create-phase also enables the Resource Manager to ensure that all resources needed for a task are available before actually initiating the task's execution.

2.4.5.1 Example

The following example illustrates the three-phase task model and the passing of task state between the three task functions. Some pseudo code is used, as the goal here is to highlight the basic concepts. Complete examples are described later in *DSP/BIOS Bridge Examples*.

In this example a GPP application makes calls into the API to control an audio filter task running on an attached DSP. The GPP-side API is used to control the DSP, but no data is streamed between the GPP and the DSP. A block diagram for the example is shown below.

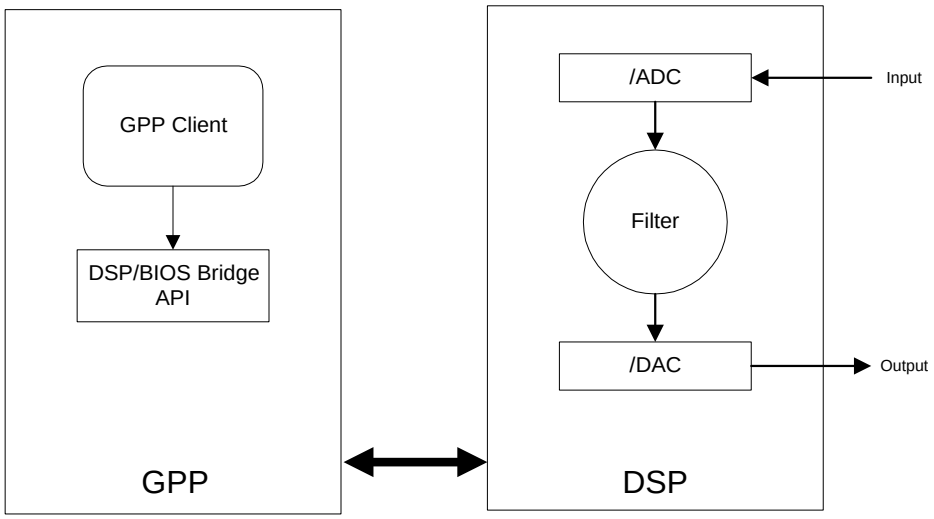

Figure 7 Using a DSP as a Filter

To initiate the Filter task on the DSP, the GPP application will:

- Attach to the DSP
- Allocate the Filter task node and the /ADC and /DAC device nodes
- Connect the nodes

- Create the nodes on the DSP
- Start the Filter node running

The DSP-side code for the Filter task is broken into three phases:

2.4.5.2 Create Phase

The Filter task inputs data from the /ADC device and outputs processed data to the /DAC device. In Filter's create-phase function (shown below), it must create the streams to the two devices and allocate buffers for the streams. The node's create-phase function will open the streams and allocate stream buffers based upon stream definition information passed from the GPP (via RMS_StrmDef structures). Once the streams are successfully opened, and the stream buffers successfully allocated, the create function will place the corresponding handles into a **`filterData`** context object. This context object will be attached to the node's environment (by the statement **`env->moreEnv = (Ptr) filterPtr`**), so that it can be accessed during the node's execute-phase function, where the actual filtering operation is performed.

```
Int filterCreate(Int argLength, Char * argData, Int numInStreams,
    RMS_StrmDef * inDef[], Int numOutStreams, RMS_StrmDef * outDef[],
    NODE_EnvPtr env)
{
    struct filterData * filterPtr;
    STRM_Attrs attrs = STRM_ATTRS;

    /* allocate and zero-fill a context structure for this instance of
Filter */
    if((filterPtr = MEM_calloc(0, sizeof(filterData), 0)) == NULL) {
        return(RMS_EOUTOFMEMORY);
    }

    /* create input and output streams, save handles in context structure
*/
    attrs.nbufs = inDef[0]->nbufs;   /* define input stream attributes
*/
    attrs.segid = inDef[0]->segid;
    attrs.align = inDef[0]->align;
    attrs.timeout = inDef[0]->timeout;
    filterPtr->inStream = STRM_create(inDef[0]->name, STRM_INPUT,
        inDef[0]->bufsize, attrs);
    attrs.nbufs = outDef[0]->nbufs;  /* define output stream attributes
*/
    attrs.segid = outDef[0]->segid;
    attrs.align = outDef[0]->align;
    attrs.timeout = outDef[0]->timeout;
    filterPtr->outStream = STRM_create(outDef[0]->name, STRM_OUTPUT,
        outDef[0]->bufsize, attrs);

    /* check for stream creation failure */
    if((filterPtr->inStream == NULL) || (filterPtr->outStream == NULL)) {
        return(RMS_ESTREAM);
    }

    /* allocate stream buffers, saving buffer addresses in context
structure */
    filterPtr->buf1 = STRM_allocateBuffer(filterPtr->inStream,
        inDef[0]->bufsize);
    filterPtr->buf1size = inDef[0]->bufsize;
```

```
        filterPtr->buf2 = STRM_allocateBuffer(filterPtr->outStream,
            outDef[0]->bufsize);
        filterPtr->buf2size = outDef[0]->bufsize;

        /* attach context structure to node's environment */
        env->moreEnv = (Ptr) filterPtr;

        /* return OK if all allocations successful */
        if((filterPtr->buf1 != NULL) && (filterPtr->buf2 != NULL)) {
          return(RMS_EOK);
        }
        else {
            return(RMS_EOUTOFMEMORY);
        }
    }
```

Execute Phase

Filter's execute-phase function (shown below) is a simple loop that submits an empty buffer to its input stream, reclaims a filled buffer, filters the data in the input buffer, submits the filtered data to its output stream, and then reclaims an empty buffer from the output stream. Note that the first thing the function does is to de-reference the **`env->moreEnv`** pointer to access the context structure attached to the node's environment in the create phase.

The execute function will watch for a shutdown message (`RMS_EXIT`) from the GPP; when `RMS_EXIT` arrives, it will break out of the processing loop, idle streams, reclaim outstanding buffers, and return.

```
    Int filterExecute(NODE_EnvPtr env)
    {
        RMS_DSPMSG msg;
        Ptr buf;
        Arg bufArg;

        /* de-reference environment to access context structure for this
    instance */
        struct filterData * filterPtr = (struct filterData *)env->moreEnv;

        /* initialize message from GPP to something other than RMS_EXIT */
        msg.dwCmd = ~RMS_EXIT;

        /* issue an empty buffer to prime the input stream */
        STRM_issue(filterPtr->inStream, filterPtr->buf1, BUFSIZE, BUFSIZE,
    0);

        /* initialize buffer pointer to buf2 */
        buf = filterPtr->buf2;

        /* initiate signal processing loop */
        while(msg.dwCmd != RMS_EXIT) {

            /* issue an empty buffer to the input stream */
            STRM_issue(filterPtr->inStream, buf, BUFSIZE, BUFSIZE, 0);

            /* reclaim a full buffer from the input stream */
            STRM_reclaim(filterPtr->inStream, &buf, NULL, &bufArg);

            ' do filtering '

            /* issue a full buffer to the output stream */
            STRM_issue(filterPtr->outStream, buf, BUFSIZE, BUFSIZE, 0);
```

```
            /* reclaim an empty buffer from the output stream */
            STRM_reclaim(filterPtr->outStream, &buf, NULL, &bufArg);

            /* check for shutdown message from the GPP (but don't block) */
            NODE_getMsg(env, &msg, 0);
        }

        /* idle the two streams */
        STRM_idle(filterPtr->inStream, TRUE);
        STRM_idle(filterPtr->outStream, TRUE);

        /* reclaim the buffer remaining in the input stream */
        STRM_reclaim(filterPtr->inStream, &buf, NULL, &bufArg);

        return(RMS_EOK);
    }
```

2.4.5.3 Delete Phase

Filter's delete-phase function (shown below) will delete any resources it allocated in the create phase. Note again that the first thing the function does is to de-reference the **`env->moreEnv`** pointer to access the context structure attached to its environment.

```
    Int filterDelete(NODE_EnvPtr env)
    {
        /* de-reference environment to get context structure for this
    instance */
        struct filterData * filterPtr = (struct filterData *)env->moreEnv;

        /* free input stream buffer and delete the input stream */
        if (filterPtr->inStream != NULL) {
            if (filterPtr->buf1 != NULL) {
                STRM_freeBuffer(filterPtr->inStream, filterPtr->buf1,
                    filterPtr->buf1size);
            }
            STRM_delete(filterPtr->inStream);
        }

        /* free output stream buffer and delete the output stream */
        if (filterPtr->outStream != NULL) {
            if (filterPtr->buf2 != NULL) {
                STRM_freeBuffer(filterPtr->outStream, filterPtr->buf2,
                    filterPtr->buf2size);
            }
            STRM_delete(filterPtr->outStream);
        }

        /* free the context structure for this instance */
        MEM_free(0, filterPtr, sizeof(filterData));

        return(RMS_EOK);
    }
```

3 Developing DSP/BIOS Bridge Applications

3.1 The Mechanics of Building DSP/BIOS Bridge Applications

DSP/BIOS Bridge provides both GPP and DSP-side application programming interfaces (APIs), allowing GPP and DSP programs to communicate and exchange data with each other. The GPP and DSP-side programming environments are very different and separate development tools and procedures are used to build each side. These procedures are summarized below.

3.1.1 The GPP-Side of the Application

GPP applications that use DSP/BIOS Bridge are built in the same environment GPP developers are familiar with. Applications are built using the standard procedures and facilities provided by the GPP OS Vendor. To use the DSP/BIOS Bridge APIs, you should follow standard GPP C/C++ programming practices for the selected OS, and 1) include the DSP/BIOS Bridge API header file, and 2) link to the appropriate DSP/BIOS Bridge libraries.

3.1.1.1 DSP/BIOS Bridge Headers for GPP Code

As with any API, you must include an API header file in your C/C++ source code. For DSP/BIOS Bridge, you must include the header file **dbapi.h** in any source code that will be calling DSP/BIOS Bridge functions. The DSP/BIOS Bridge header should be included after OS-specific headers.

```
/* after #including GPP OS headers... */
#include <dbapi.h>    /* required for DSP/BIOS Bridge apps */
```

Once this file is included, your C source code can access any of the DSP/BIOS Bridge GPP API functions and structures. The directory containing DSP/BIOS Bridge headers must be specified to the compiler using –I switch.

3.1.1.2 Linking with the DSP/BIOS Bridge Libraries

To build your GPP application you must link to the library named libbridge.so, which is a (manually created) soft link to the latest version of DSP/BIOS Bridge API import library. These files must be placed in the default directory for host and target libraries so that the linker can locate them. Usually this directory is /lib or /usr/lib. See ld and ldconfig for more information. Figure 8 summarizes the build process.

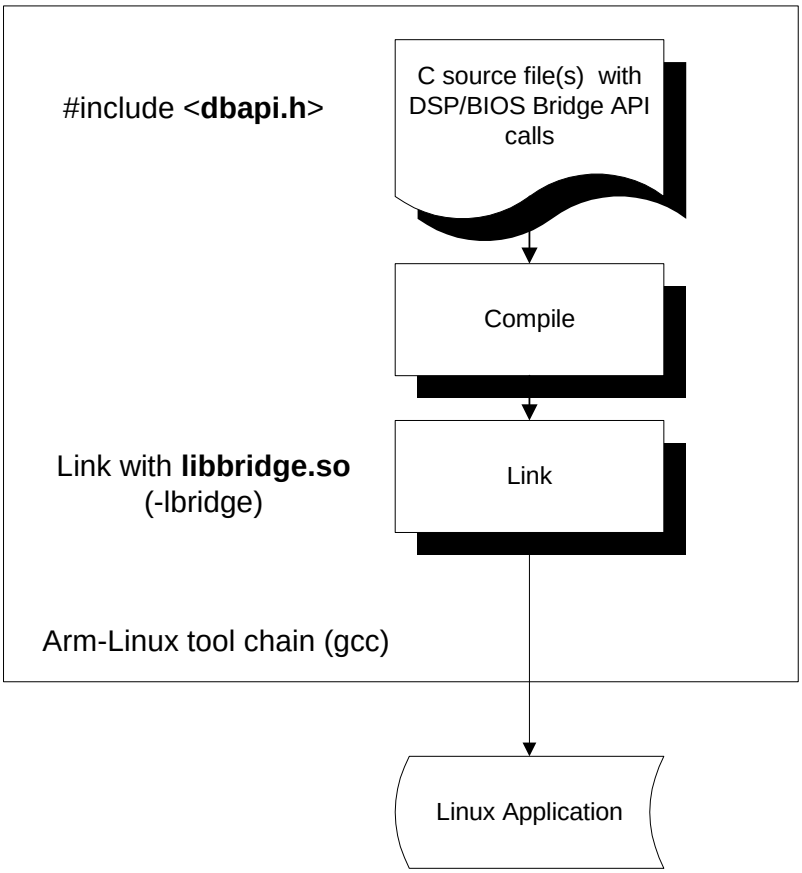

Figure 8 Linux-Side Build Process

At runtime, when your application makes DSP/BIOS Bridge API calls, these calls will execute code residing in the shared library provided with the Linux bridge driver.

3.1.2 The DSP-Side of the Application

For DSP side build procedures please refer to DSP-side Bridge documentation (to be written in the future).

3.1.3 Configuring Your Application in the DSP/BIOS Bridge Configuration Database

In DSP/BIOS Bridge applications blocks of related DSP code and data are abstracted as signal processing “nodes”. With this DSP-side node abstraction, GPP-side DSP/BIOS Bridge APIs allow a GPP application to create, execute, and delete nodes on the DSP at runtime. DSP/BIOS Bridge maintains a Configuration Database that holds configuration information about DSP nodes.

So, in addition to the normal compile/link/debug cycle for GPP and DSP applications, you will also need to configure DCD information for your application’s DSP nodes. For example, each DSP node has a unique identifier and other properties such as node type, runtime priority, stack size, function names, etc. Before the node can be used by your GPP application, its configuration information needs to be registered in the DCD. For further information about Configuring DSP Side node please refer to DSP-side Bridge documentation (to be written in the future).

3.1.4 Building the DSP Base Image

All DSP-side code are built and statically linked into a “DSP base image,” which is in Common Object File Format (COFF). This all encompassing image will contain the DSP/BIOS kernel, DSP/BIOS Bridge binaries, drivers, application startup code, as well as code and data for nodes that will later be instantiated by DSP/BIOS Bridge in response to GPP application requests. For further information about building the DSP Base Image please refer to DSP-side Bridge documentation (to be written in the future).

3.2 Loading the DSP/BIOS Bridge Driver and Running the DSP Base Image

There are two methods for loading and running the DSP base image: 1) Using the DSP/BIOS Bridge DSP AutoStart feature; or 2) Using a DSP/BIOS Bridge development utility **cexec**.

3.2.1 Loading DSP/BIOS Bridge driver

The DSP/BIOS Bridge driver is loaded using **insmod**. A device node is then created in the file system using **mknod**. Please refer Linux man pages for details about **insmod** and **mknod**. The DSP/BIOS Bridge API library opens this device node to request DSP/BIOS Bridge driver services. The following is a sample script to load the DSP/BIOS Bridge driver; this script can be called during system initialization to load the DSP/BIOS Bridge driver whenever Linux boots.

```
# invoke insmod with all arguments we were passed
# and use a pathname, as newer modutils don’t look in . by default

insmod ./bridgedriver.o $* || exit 1

# remove stale nodes
rm -f /dev/dspbridge

# Get the major device number
MAJ=`awk "\\$2==\"DspBridge\" {print \\$1}" /proc/devices`

if [ "$MAJ" == "" ]
then
    echo "Can't find DspBridge driver, can't continue"
    exit 1
fi
CNT=`echo "$MAJ" | wc -l`
if [ $CNT != 1 ]
then
    echo "Found multiple instances of DspBridge driver, can't continue"
    exit 1
fi

# Create device node with minor number 0
mknod /dev/dspbridge c $MAJ 0
```

3.2.2 Unloading DSP/BIOS Bridge driver

The DSP/BIOS Bridge driver is unloaded using **rmmod**. The device node file is deleted using **rm**. Please refer Linux man pages for details about **rmmod** and **rm**. The following is a sample script to unload the DSP/BIOS Bridge driver:

```
# Unload DSP/BIOS Bridge driver
rmmod bridgedriver
```

```
# Remove device node file
rm -f /dev/dspbridge
```

When loading the DSP/BIOS Bridge driver, users can specify the major device number. The default is zero, which indicates automatic assignment. The minor device number is hardcoded to zero. The syntax is:

```
insmod ./bridgedriver.o driver_major=<major number>
```

3.2.3 AutoStart Feature

When loading the DSP/BIOS Bridge driver, users can specify an optional argument to load and start the DSP base image. The syntax is:

```
insmod ./bridgedriver.o base_img=<DSP image file>
```

3.2.4 `cexec` utility

The **`cexec`** utility is an application that can load and start a new DSP image from the GPP. See the *DSP/BIOS Bridge Reference Guide* for details on **`cexec`** usage.

3.3 DSP/BIOS Bridge Exception Handling

In the event of a DSP exception, developers have the ability to signal this event to any GPP application through DSP/BIOS Bridge's DSP exception handling mechanism. The mechanism works as follows:

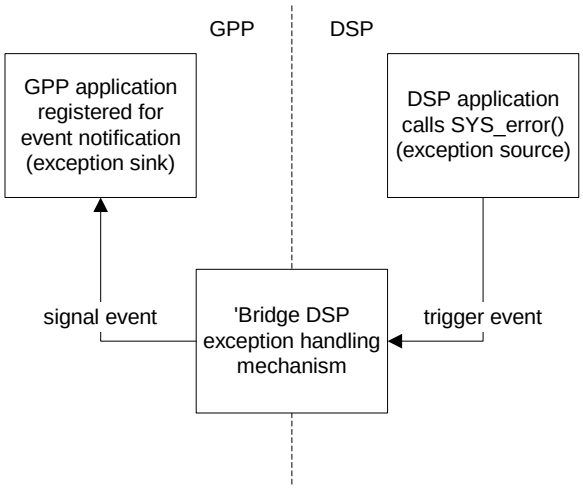

Figure 9 DSP Exception Handling Mechanism

1) A GPP application registers to receive notification for the class of DSP/BIOS exceptions. This requires the user to call **`DSPProcessor_RegisterNotify`** with `DSP_SYSERROR` as the event mask. To register for other classes of events, call **`DSPProcessor_RegisterNotify`** with the appropriate event mask. Please consult the *DSP/BIOS Bridge Reference Guide* for details.
2) An exception X occurs in a DSP application.
3) The DSP application notifies the DSP/BIOS Bridge exception handler of exception X through the BIOS function **`SYS_error`** (e.g. **`SYS_error("module_name", X)`**. By default, **`SYS_error`** is mapped to the user-customizable function **`USR_logError`**.

4) The default implementation of **USR_logError** calls **SHM_interrupt2(X)**, which causes the exception handler to transparently notify the GPP of the exception X.

5) The GPP application thread blocking on DSP exceptions will unblock and run. The GPP application can retrieve precise information about the exception through **DSPProcessor_GetState**.

For example, let us assume a GPP application wants to receive notification for DSP/BIOS exception SYS_EBADIO. The GPP creates a thread to wait for the exception as follows:

```
void main()
{
    . . .
    uNotifyType = DSP_SIGNALEVENT;
    /*
     * to receive notification for all DSP/BIOS exceptions,
     * register event mask type DSP_SYSERROR
     */
    uEventMask = DSP_SYSERROR;

    /* register to receive notification for event */
    status = DSPProcessor_RegisterNotify(testTask->hProcessor,
            uEventMask, uNotifyType, &testTask->hNotification);
    . . .
    /* create GppExceptionThread */
    . . .
}

static void GppExceptionThread(TEST_TASK testTask)
{
    DSP_PROCESSORSTATE   procState;
    DSP_HNOTIFICATION  aNotifications[1];
    DWORD index = -1;
    aNotifications[0] = & testTask->hNotification;

    . . .
    dwResult = DSPManager_WaitForEvents (aNotifications, 1, &index,
DSP_FOREVER);
    switch (index) {
    case 0:
        /*
         * retrieve the precise exception value;
         * value will be stored in procState.errInfo
         */
        DSPProcessor_GetState(testTask->hProcessor,
            &procState, sizeof(DSP_PROCESSORSTATE));
        break;
    default:
        /* unexpected result, take appropriate action */
    }
    . . .
}
```

A DSP application can trigger the SYS_EBADIO exception as follows:

```
/* an I/O error just occurred, notify the GPP. */
SYS_error("DSP_MODULE_NAME", SYS_EBADIO);
```

When the SYS_EBADIO exception gets triggered on the DSP, the DSP/BIOS Bridge exception handler will cause the DSP_SYSERROR event to be signaled on the GPP, thereby unblocking the GPP thread waiting on DSP_SYSERROR.

Currently, the DSP/BIOS Bridge exception handler accepts DSP/BIOS exceptions as well as user-defined exceptions. The range of valid values is defined as follows:

Table 7 DSP/BIOS Exception Vector

0 – 255	**DSP/BIOS exception values**
256 – 511	DSP/BIOS Bridge exception values
512 – 1023	User-defined exception values

In order for the GPP to properly receive DSP signaled exceptions, the DSP exception values must fall in the appropriate ranges defined above. Failure to define valid values will result in unpredictable system behavior. Furthermore, for a GPP application to receive proper notification of DSP/BIOS exceptions, it must use the correct event mask in **`DSPProcessor_RegisterNotify`**.

```
/* Set GPP to wait on to DSP/BIOS exceptions; use DSP_SYSERROR mask */
UEventMask = DSP_SYSERROR;

/* GPP will register for notification of any DSP/BIOS exceptions */
status = DSPProcessor_RegisterNotify(testTask->hProcessor,
    uEventMask, uNotifyType, &testTask->hNotification);

/*
 * to find out the value of the last exception,
 * use DSPProcessor_GetState(), then access
 * procState.errInfo to retrieve the exception value
 */
DSPProcessor_GetState(testTask->hProcessor, &procState,
    sizeof(DSP_PROCESSORSTATE));
```

Finally, the developer should be aware of customizations to **`SYS_error`**, as changes can disable the DSP/BIOS Bridge exception handler. For the TI OMAP 2420/2430/3430, ensure the following code (highlighted in bold) is part of the user-defined **SYS_error/USR_logError** implementation:

```
// user defined SYS_error() fxn
USR_logError(Exception X)
{
    // perform exception/error processing
    . . .
    // use the following fxn to notify GPP of exception
    SHM_interrupt2(X);
    . . .
}
```

4 DSP/BIOS Bridge Nodes

4.1 Overview

In DSP/BIOS Bridge, a *node* is an abstraction of a block of related program code and data. A node is not an individual hardware processing element, but one of probably many software-processing elements running on a single DSP.

As mentioned above, nodes can be dynamically created, executed, and deleted on a DSP at runtime, under control of a Resource Manager (RM) on an attached general-purpose processor (GPP). On the DSP, a Resource Manager Server facilitates the creation, execution, and deletion of nodes, and messaging between nodes and the GPP under control of the Resource Manager on the GPP. The relationship of the Resource Manager on the GPP and the RM Server on the DSP is shown in Figure 2 above. In the figure “video”, “audio”, and “speech” are signal-processing nodes that have been launched by the RM Server under Resource Manager control.

Nodes can be used for signal processing purposes as well as general control processing. Three types of nodes are currently defined. Each type of node has different capabilities and different uses.

Task Node – This is the basic processing element used in DSP/BIOS Bridge. A task node exists as an independent execution thread in DSP/BIOS. Task nodes can exchange messages with other nodes on the DSP or with the GPP. Task nodes can also stream large data blocks to/from other nodes and the GPP using a device-independent I/O interface.

XDAIS Socket Node – An XDAIS socket node is an enhanced task node that provides a framework, or housing, for a TMS320 DSP Algorithm Standard (hereafter referred to as XDAIS) compliant algorithm. The socket node facilitates data transfer from the algorithm to other nodes, or to the GPP.

Device Node – A device node manages either a DSP peripheral device or a communication path between two Task nodes or implements a software device. Device nodes that manage a peripheral device encapsulate low-level hardware and communication details.

All three node types follow a three-phase execution model, which enables efficient management of DSP resources. The three-phase model partitions node functionality in three blocks of code corresponding to the key periods of the node’s life cycle:

Create – all DSP resources needed by the node are allocated,

Execute – node’s real-time processing is performed,

Delete – all resources allocated for the node are freed

The three phases of task and XDAIS socket nodes are explicitly invoked at the appropriate time by the RM Server on `the DSP as commanded by the Resource Manager on the GPP. For device nodes, the three phases are invoked implicitly: as the streams are created/opened by client nodes, as data is streamed between nodes, and as streams are idled and closed as the client nodes are deleted, respectively.

Each type of node is accompanied by a set of configuration parameters that the Resource Manager can use as it schedules and manages DSP resources. Each type of node is further described below.

4.2 Task Node

A Task node runs as an independent execution thread in DSP/BIOS. Task nodes support device-independent streaming I/O with other nodes and with the GPP. Task node authors must provide three C-callable functions. These functions are called during the create, execute, and delete phases for the node, respectively, by the RM Server.

4.2.1 Communication Paths

In addition to the node-to-node and node-to-GPP messaging, task nodes can stream data to/from device nodes via stream (STRM) I/O.

Each task node can have up to 8 application-specific input streams and 8 application-specific output streams. A typical configuration is shown in Figure 11, with a task node inputting data from an ADC device and outputting data to a DAC device.

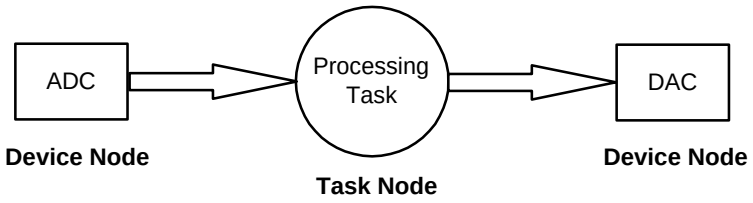

Figure 10 Task Node Between Two Devices

Task nodes do not communicate directly with other task nodes, but intermediary devices must be used, as shown in Figure 12. In this case a "pipe" device node is used to link two task nodes on the same DSP.

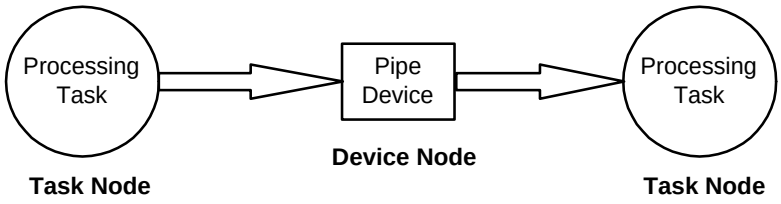

Figure 11 Task Node Communicating via Pipe Device

Task nodes can stream data to the GPP via the DSP-to-GPP link driver. The link driver supports multiple channels, and this is the same device driver through which the RM Server communicates with the Resource Manager.

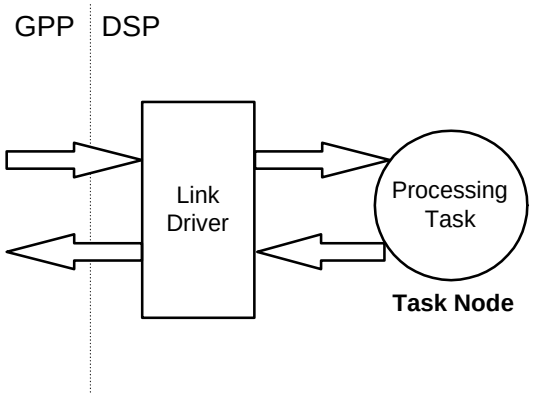

Figure 12 Task Node Streaming Data to GPP

4.2.2 Execution Timeline

The following table summarizes a timeline of the actions performed by the RM Server and node-specific functions during the create, execute, and delete phases of a task node.

Table 8 Execution Timeline for Task Node

Node Phase	RM Server Does:	Node-Specific Function Does:
Create	- allocates an environment structure for the node - creates a task for the node, in an inactive state - sets up messaging for the node - calls node's create function ⇒	
		- allocates an application-specific context structure to hold additional instance data for this instance of the node. - allocates **all** application-specific resources needed by the node, placing resource handles in the context structure
	- sends reply to Resource Manager, passing node's create function return value and a pointer to the task's environment structure - sets the node's execute function to that indicated by the RM - responds to the RM: going to launch the node - launches the node by changing the node's priority from 'inactive', to it's runtime priority ⇒	

Node Phase	RM Server Does:	Node-Specific Function Does:
Execute		- does real-time processing - monitors for messages from GPP, watching for a (RMS_EXIT) shutdown command; if shutdown command received, exit
	- sends message to Resource Manager indicating node's execute function return code - calls the node's delete-phase function ⇒	
		- releases all application-specific resources allocated in the create phase (e.g., deletes data streams) - releases context structure for the current instance of the node
Delete	- deletes the node's task - releases the node's environment and messaging resources - sends a reply to the Resource Manager, passing the node's delete function return value	

4.2.3 Environment Structure

One of the things that the RM Server does as it creates a task node on the DSP is to create an "environment structure" for the node. This structure is shown below.

```
typedef struct RMS_TskEnv {
    MSG_Obj message;
    Ptr     moreEnv;
    Ptr           self;
}
```

The first element is a messaging object (MSG_Obj), which contains the node's private message queue and a handle for the queue's companion semaphore. The second element of the environment structure is an arbitrary pointer (*moreEnv*) that the node can use to reference additional context data for the node. The third element is a handle to the node's thread in DSP/BIOS (actually a DSP/BIOS TSK_Handle).

A task node references its environment via a node environment pointer, `NODE_EnvPtr`; for a task node, `NODE_EnvPtr` is simply a pointer to an `RMS_TskEnv` structure.

4.2.4 Function Signatures

The code for each task phase is encapsulated in its own C-callable function. The signatures for these functions are listed below.

4.2.4.1 Create Function

The signature for a task node's create function is:

```
RMS_STATUS nodeCreate(Int argLength, Char * argData, Int numInStreams,
    RMS_WORD inDef[], Int numOutStreams, RMS_WORD outDef[], NODE_EnvPtr
node);
```

argLength and *argData* are used to pass a block of node arguments from the GPP application that allocates the node to the node on the DSP. [For example, a telephony application running on the GPP might pass a telephone number to the task node on the DSP, via the *argData* pointer.] *inDef[]* and *outDef[]* are used to define the application-specific STRM streams for the node. [These are arrays of pointers to STRM definition structures that define a stream's: buffer size, number of buffers, memory segment for the buffers, timeout for blocking calls, and the stream name.] The *node* environment pointer points to the node's environment structure created by the RM Server. The **nodeCreate** function can de-reference this pointer to get to the *moreEnv* element and set the element to point to additional context data for the task.

4.2.4.2 Execute Function

The signature for a task node's execute function is:

```
RMS_STATUS nodeExecute(NODE_EnvPtr node);
```

nodeExecute can de-reference the *node* environment pointer to access context data allocated in the node's create phase, as referenced by the *node->moreEnv* element.

4.2.4.3 Delete Function

The signature for a task node's delete function is:

```
RMS_STATUS nodeDelete(NODE_EnvPtr node);
```

nodeDelete should de-reference the *node* environment pointer and delete any node-specific context data allocated in the create-phase.

4.2.5 Execution Context

The following table summarizes the execution context for each Task Node function.

Table 9 Execution context for Task Node

Function	Execution Context
nodeCreate	RM Server
nodeExecute	Task created for node
nodeDelete	RM Server

4.3 XDAIS Socket Node

An XDAIS socket node allows an XDAIS-compliant algorithm to “plug-in” to a DSP/BIOS Bridge application. The socket node provides a housing for the algorithm, calling algorithm functions as appropriate, and passing data to/from the algorithm as parameters in the algorithm function calls.

In a typical scenario, the XDAIS algorithm is written by a DSP algorithm developer, and the “socket” is written by an OEM or system integrator to enable the XDAIS algorithm to be utilized within the DSP/BIOS Bridge framework.

This section describes the XDAIS socket concept. For detailed information on XDAIS, consult *TMS320 DSP Algorithm Standard Rules and Guidelines* and *TMS320 DSP Algorithm Standard API Reference*.

4.3.1 Socket Concept

A key benefit of XDAIS-compliant algorithms is that they are framework-independent. XDAIS defines rules to enhance portability, such as: algorithms must be reentrant within a preemptive environment, algorithm data references must be fully relocatable, algorithms must not directly access any peripheral device, etc. XDAIS algorithms can be viewed as data transducers or transformers; to use these transducers in a DSP/BIOS Bridge environment, a “socket” or housing must be created for the XDAIS algorithm to manage I/O between the algorithm and the rest of the DSP/BIOS Bridge application, and to allow scheduling within DSP/BIOS. The following figure illustrates the concept.

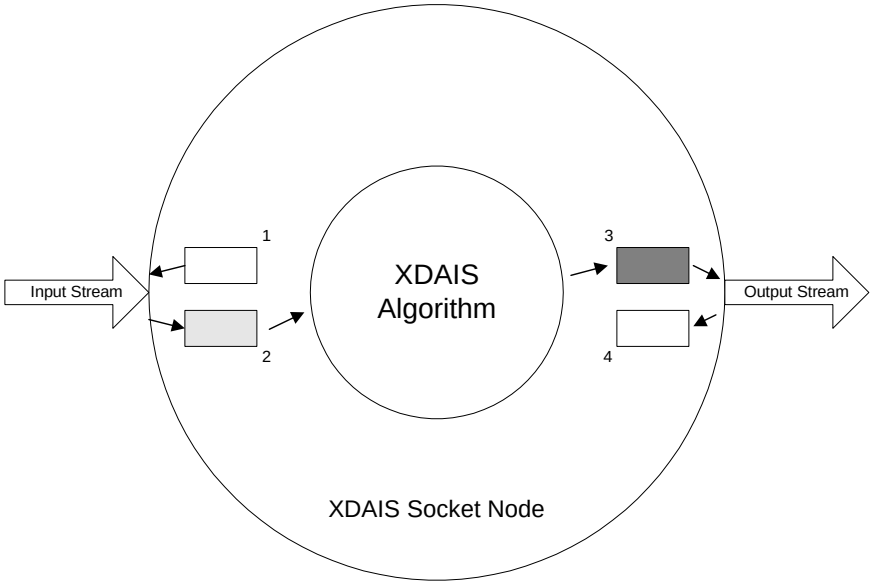

Figure 13 XDAIS Socket Concept

The XDAIS Socket will:

- submit an empty buffer (1) to an input stream,
- retrieve a filled data buffer (2) from the input stream,

- pass the buffer to the algorithm's processing function,
- send the resultant data buffer (3) to an output stream, and
- retrieve an empty buffer (4) back from the output stream.

Depending on the algorithm's processing model, the buffer on the left side of the socket may be the same as the buffer shown on the right side (i.e., the algorithm does in-place processing on the buffer, and the output is contained in the same buffer); or, the left and right side buffer will be different if the algorithm produces output data in a separate buffer (i.e., it does a 'transform and copy' operation). The *socket* will exist as a task in DSP/BIOS, allowing it to be scheduled along with all other DSP/BIOS Bridge nodes.

XDAIS-compliant algorithms can be used directly within DSP/BIOS Bridge, without modification. To integrate the algorithm into DSP/BIOS Bridge, the application developer (or OEM, or system integrator) must write three socket functions (create, execute, delete), corresponding to the three phases of the node's lifecycle. The socket's execute-phase function defines how/when the socket's task will wake up, how data is routed to/from the algorithm, and how the algorithm-specific process functions are called.

4.3.2 Communication Paths

The communication paths for an XDAIS socket are shown in the following figure. The socket communicates to device nodes via STRM streams (two are shown in the figure) and to the RM Server via message queues. The XDAIS algorithm itself does not communicate with the outside world, it is the socket that is scheduled in the RTOS and manages all I/O and messaging, passing buffers to/from the algorithm as appropriate.

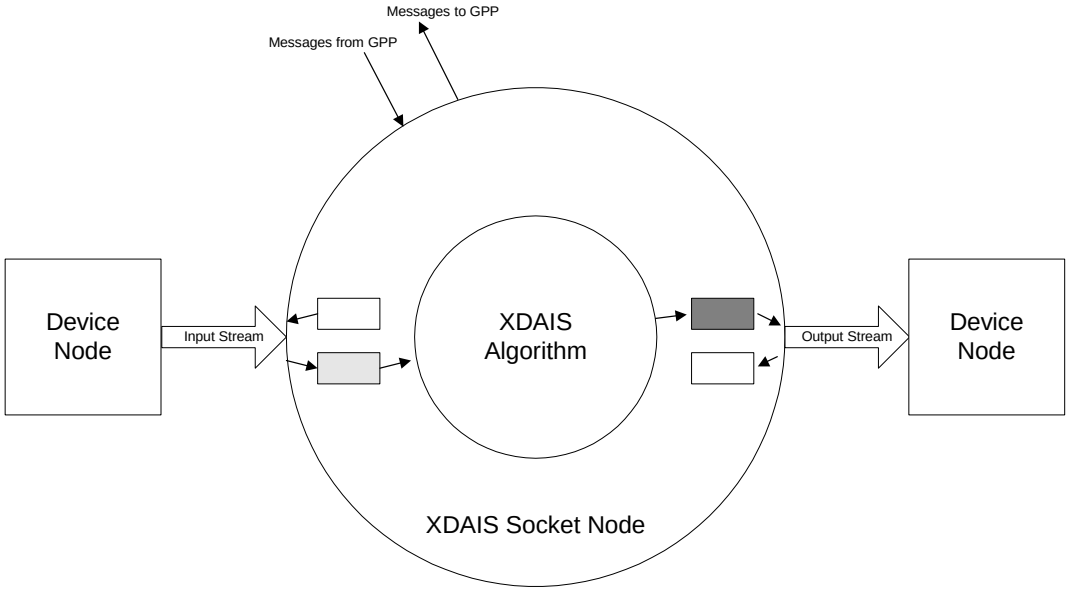

Figure 14 Socket Communication Paths

4.3.3 Standard XDAIS DMA Interface Support and DMA Resource Management

XDAIS-compliant algorithms are not allowed to *directly* access or control the hardware or peripherals, which include the DMA. The TMS320 DSP Algorithm Standard specifies two sets of API interfaces are required for accessing DMA resources: IDMA2/IDMA3 and ACPY2/ACPY3. These interfaces allow the client application and the algorithm to negotiate DMA resources and grant algorithms controlled access to DMA services. The client application must query the algorithm during instance creation about its DMA resource requirements and grant it the DMA handles to access DMA services.

XDAIS-compliant algorithms access DMA hardware using a "logical" DMA channel abstraction. Algorithms request and receive handles representing logical DMA channels from the client application. Algorithms submit DMA transfer requests on logical channels through the use of XDAIS ACPY2/ACPY3 APIs.

IDMA2/IDMA3. All algorithms that use DMA resources must implement the IDMA2/IDMA3 interface. This interface allows the algorithm to request and receive handles representing private "logical" DMA resources.

ACPY2/ACPY3. These functions are implemented as part of the client application and called by the algorithm (and possibly the client application). A client application must implement the ACPY2/ACPY3 interface (or integrate a provided ACPY2/ACPY3 interface) in order to use algorithms that use the DMA resource. The ACPY2/ACPY3 interface describes the comprehensive list of DMA operations an algorithm can access through the logical DMA channels it acquires through the IDMA2/IDMA3 protocol. The ACPY2/ACPY3 functions allow:

- Configuring channel DMA transfer parameters.
- Scheduling asynchronous DMA transfers.
- Synchronizing with scheduled transfers (both blocking and non-blocking).

The "Use of the DMA Resource" chapter in *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) describes the use of these interfaces. The *TMS320 DSP Algorithm Standard API Reference* (SPRU360) provides details on each function. Readers are advised to read ‘Joule DMA Reference Framework and Design’ for C64 specific ACPY3 and IDMA3 frame work details.

Collectively, IDMA2/IDMA3 and ACPY2/ACPY3 describe a flexible and efficient model that greatly simplifies management of system DMA resources and services by the client application and a simple and powerful mechanism for the algorithm to configure and access DMA services.

4.3.4 DMA Resource Management in DSP/BIOS Bridge

The DSP/BIOS Bridge DMA Manager (DMAN) is responsible for allocating and managing the DMA resources. DMAN is part of the DSP framework components. DMAN queries the algorithm about its resources and grants them. Once the resources are allocated, algorithms can call the DMAN’s ACPY API’s to do the data transfer

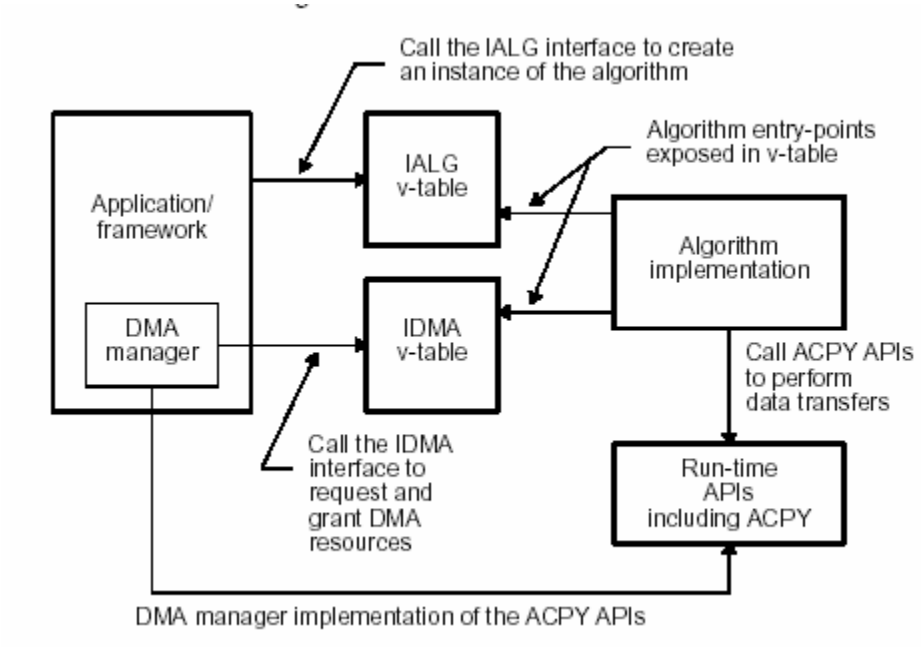

Figure 15 XDAIS DMA Resource Management

4.3.5 Logical DMA channels and Queue ID's

When XDAIS compliant algorithms request logical DMA channels they must specify a Queue ID for each channel. All DMA transfer requests submitted to logical channels sharing the same Queue ID must be carried out in a first-in first-out order. DSP/BIOS Bridge implementation of DMAN and ACPY2 libraries satisfy this FIFO ordering requirement by associating each logical channel with a unique physical DMA channel with a corresponding software queue. DSP/BIOS Bridge configuration tool allows the system integrator to utilize the Queue IDs when mapping requested logical channels to physical channels.

> The configurable DMAN properties of the base CDB are used to dedicate available physical channels to DMAN. NODE properties are used to configure the Queue ID to physical channels mappings for each individual algorithm.

Figure 18 shows the typical sequence of events during the lifetime of a XDAIS socket node using DMAN under DSP/BIOS Bridge environment. For simplicity, only one XDAIS node is shown.

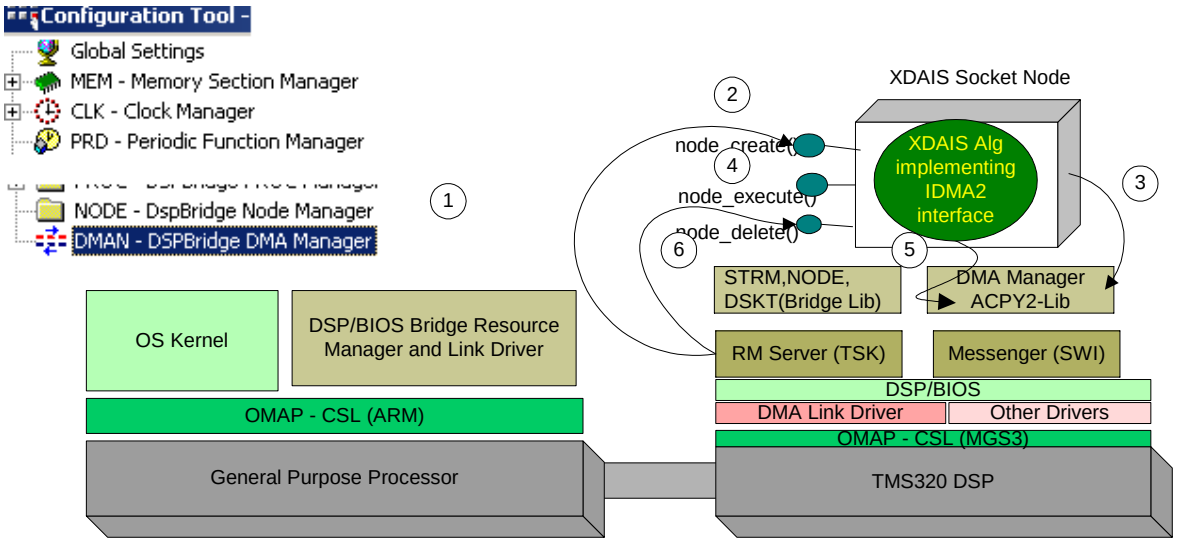

Figure 16 Flow of events

1. XDAIS algorithm specifies the Queue ID for each logical channel it requests.

2. RM server during node_create calls DSKT library APIs to instantiate a XDAIS node

3. Socket node calls DMAN to allocate and grant the algorithm handles for the logical channel it requested.
    - DMAN verifies alg and idma fxns are from the same implementation.
    - Calls alg's dmaGetChannelCnt to get the number of channel requests.
    - Calls alg's dmaGetChannels to query about its channel requirements.
    - Creates and initializes the logical dma handles.
    - Calls alg's dmaInit to grant the dma handles and resources.

4. RM Server calls node_execute. DSKT calls STRM API's to transfer buffers to/from GPP via the link driver.

5. Algorithm uses ACPY2 library to transfer data over granted DMA channels.

6. Upon node_delete call from the RM server, DSKT calls algorithm's IDMA2 dmaGetChannels and closes or releases the DMA resources via DMAN APIs.

4.3.6 Execution Timeline

The following table summarizes a timeline of the actions performed by the RM Server, the XDAIS socket, and the XDAIS algorithm functions.

Table 10 Execution Timeline for XDAIS Socket Node

Node Phase	RM Server Does:	xDIAS Socket Does:	XDAIS Algorithm:
Create	■ allocates an environment structure for the socket ■ creates a task for the socket in an inactive state ■ sets up messaging for the socket ■ calls socket's create function ⇒		
		■ allocates an additional structure to hold socket instance data ■ (optionally) initializes arrays to hold stream handles, buffer pointers, buffer arguments, and buffer sizes ■ opens all I/O streams for the socket ■ calls algNumAlloc to get count of data regions to allocate for the algorithm ⇒	
			algNumAlloc()
		■ allocates a memory descriptor table ■ calls algAlloc to fill the memory descriptor table ⇒	
			algAlloc()
		■ allocates memory, as described in the memory descriptor table ■ calls algInit to allow algorithm to initialize its memory ⇒	
			algInit()

Node Phase	RM Server Does:	xDIAS Socket Does:	XDAIS Algorithm:
		[Only when alg implements IDMA2 interface] ■ obtains the maximum number of logical DMA channels requested by the algorithm instance. ■ allocates table of dma records used to query the algorithm's DMA resources. ■ obtains algorithms dma channel requirements	
			dmaGetChannelCnt dmaGetChannels
		[Only when alg implements IDMA2 interface] ■ allocates and initializes the logical dma channel handles ■ grants the logical handles to the algorithm.	
			dmaInit
	■ sends reply to RM, passing socket's create function return value, and a pointer to the socket environment		
Execute	■ responds to RM that it's going to launch the socket ■ change socket's task priority to launch the socket		

Node Phase	RM Server Does:	xDIAS Socket Does:	XDAIS Algorithm:
		■ wait until either get message for node, or all input and output streams are ready for I/O ■ if got shutdown message (RMS_EXIT), exit ■ else if got control message, parse it, and (optionally) call algControl to pass the control information to the algorithm ■ else, if all streams ready, reclaim buffers from all input and output streams ■ call algorithm functions to process data ⇒	
			• algActivate() - call algorithm processing functions • algDeactivate()
		■ issue all empty buffers back to input streams and filled buffers to output streams ■ loop back, waiting for a message, or all streams ready for I/O	
	■ send message to Resource Manager indicating socket execute function return code		
Delete	■ calls socket delete function ⇒		
		■ delete all streams to socket ■ call algFree to get descriptors of memory to free ⇒	
			algFree()

Node Phase	RM Server Does:	xDIAS Socket Does:	XDAIS Algorithm:
		■ free algorithm memory ■ free all other resources allocated during socket create	
	■ delete the socket's task ■ release node environment and messaging resources ■ send reply to Resource Manager, passing socket delete function return value		

Note on Calling algActivate/algDeactivate

In the above timeline **`algActivate/algDeactivate`** calls bracket each call to the algorithm's processing function(s). This follows from the intent of the **`algActivate`** and **`algDeactivate`** functions, which allow algorithms to restore/save their scratch data upon each iteration. If the application is structured such that scratch data locations are not concurrently shared by algorithms (based upon allocation locations and/or scheduling), then for efficiency, the **`algActivate`** and **`algDeactivate`** calls can be moved from the execute phase to the socket create and delete phases, respectively.

4.3.7 Environment Structure

The environment structure for an XDAIS Socket node is identical to that of a normal task node, as shown below.

```
typedef struct RMS_TskEnv {
    MSG_Obj  message;
    Ptr   moreEnv;
    Ptr   self;
}
```

The additional context data required for the socket is attached to the environment in the socket's create function, via the *`moreEnv`* pointer.

4.3.8 Function Signatures

The code for each socket phase is encapsulated in its own C-callable function. The signatures for these functions are listed below.

4.3.8.1 Create Function

The signature for an XDAIS socket's create function is:

```
RMS_STATUS socketCreate(Int argLength, Char *argData,
    IALG_Fxns * algFxns, Int numInStreams, RMS_WORD inDef[],
    Int numOutStreams, RMS_WORD outDef[], NODE_EnvPtr node);
```

`argLength` and *`argData`* are used to pass a block of node arguments from the GPP application that allocates the node to the node on the DSP. *`algFxns`* is a pointer to the XDAIS algorithm's IALG_Fxns table. *`inDef[]`* and *`outDef[]`* are arrays of (32-bit) pointers to STRM definition structures that define the application-specific streams used to route data buffers between the socket, and the rest of the DSP application. The last parameter *`node`* points to the socket's environment structure created by the RM Server. The create function can de-reference this pointer to get to the *`moreEnv`* element and set this element to point to additional context data for the socket.

4.3.8.2 Execute Function

The signature for a socket's execute function is:

```
RMS_STATUS socketExecute(NODE_EnvPtr node);
```

`socketExecute` can de-reference the *`node`* environment pointer to access context data allocated in the node's create phase, as referenced by the *`node->moreEnv`* element.

4.3.8.3 Delete Function

The signature for a socket's delete function is:

```
RMS_STATUS socketDelete(NODE_EnvPtr node);
```

`socketDelete` should de-reference the *`node`* environment pointer, and free any node-specific context data allocated in the create-phase.

4.3.9 Execution Context

The following table summarizes the execution context for each XDAIS algorithm and socket function.

Table 11 Execution Context XDAIS Socket Node

Function	Execution Context
algInit	RM Server
algNumAlloc	RM Server
algAlloc	RM Server
algActivate	Task created for socket, or RM Server
algDeactivate	Task created for socket, or RM Server
algControl	Task created for socket
algFree	RM Server
algMoved	Not used
dmaGetChannelCnt	RM Server
dmaGetChannels	RM Server
dmaInit	RM Server
alg process functions	Task created for socket
socketCreate	RM Server
socketExecute	Task created for socket
socketDelete	RM Server

4.4 Device Node

A Device Node is used to either manage a DSP peripheral device, implement a software device, or to manage a communication path between two nodes. When used to manage a peripheral device, it encapsulates low-level hardware details and translates device-independent I/O requests (e.g., data buffer transfers) into appropriate actions on the physical device. When used to manage a communication path between two nodes, the device node serves as a "data pipe" through which device independent I/O transfers can be performed.

A new stream interface (STRM) has been defined for DSP/BIOS Bridge. STRM is *based* upon the SIO APIs provided by DSP/BIOS-II. The key differences between STRM and SIO are that STRM exports stream buffer allocation and free functions and only supports an issue/reclaim I/O model for a smaller code footprint. The STRM interface layer sits atop the same DEV APIs and model used in DSP/BIOS-II. Device drivers written for SIO in DSP/BIOS-II should work directly under the STRM interface of DSP/BIOS Bridge. *TMS320 DSP/BIOS User's Guide* describes the device model and DEV interface in great detail; it should be consulted for more detail.

4.4.1 Overview

Two types of devices are supported in DSP/BIOS Bridge: terminating, and stackable devices. Terminating devices source or sink data. An example is a driver for outputting data samples to a digital-to-analog converter. The following figure illustrates the flow of data buffers for a terminating device. For an output device, filled data buffers are submitted (issued) to a `todevice` queue, and empty buffers are retrieved (reclaimed) from the `fromdevice` queue. Likewise, for an input device, empty buffers are issued to the `todevice` queue, and filled buffers are reclaimed from the `fromdevice` queue.

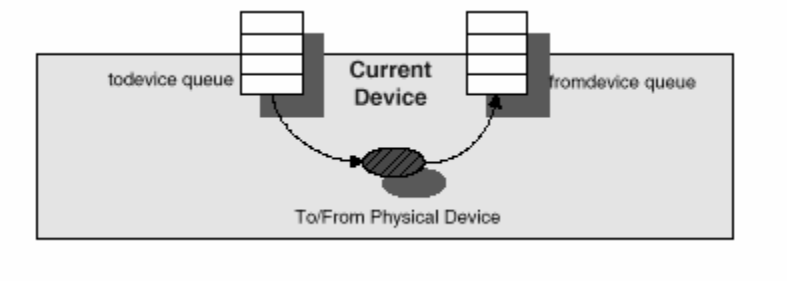

Figure 17 Buffer Flow for a Terminating Device

Stackable devices are "virtual I/O" devices that do not source or sink data but use generic device function calls to send/receive data to/from another device. An example of a stacking device is one that performs an intermediate scaling operation on data as it traverses through an I/O stream. The following figure illustrates terminating vs. stacking devices.

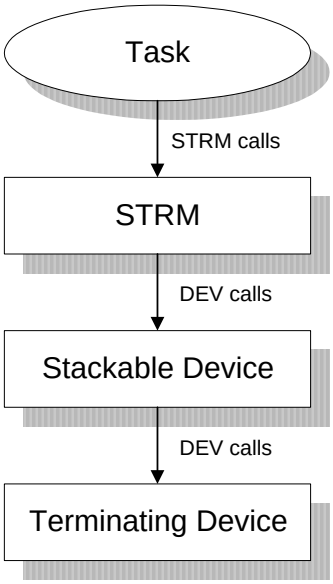

Figure 18 Relationship Between Stacking and Terminating Devices

There are two types of stackable devices: in-place and copying. An in-place stacking device performs in-place manipulation of data in buffers; a copying stacking device moves the data to another buffer while processing the data. Copying devices are needed for devices that produce more data than they receive, or because they require access to a whole buffer to generate output samples and cannot overwrite their input data. The following figure illustrates buffer flow in an in-place stacking device (denoted as "Current Device" in the figure).

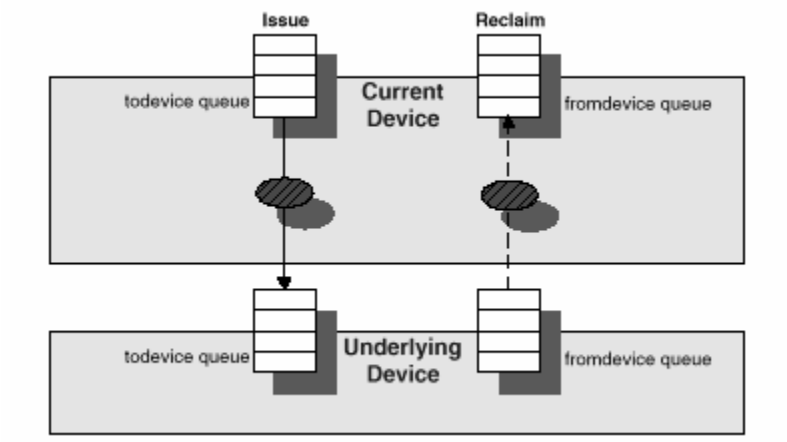

Figure 19 Buffer Flow for an In-Place Stacking Device

Buffer flow in a copying stacking device is more complicated and is illustrated in the following figure. Note that buffers that come from the task-side of the stream do not actually move to the device side of the

stream. Instead, the stacking device maintains a separate pool of buffers that are passed to the underlying device.

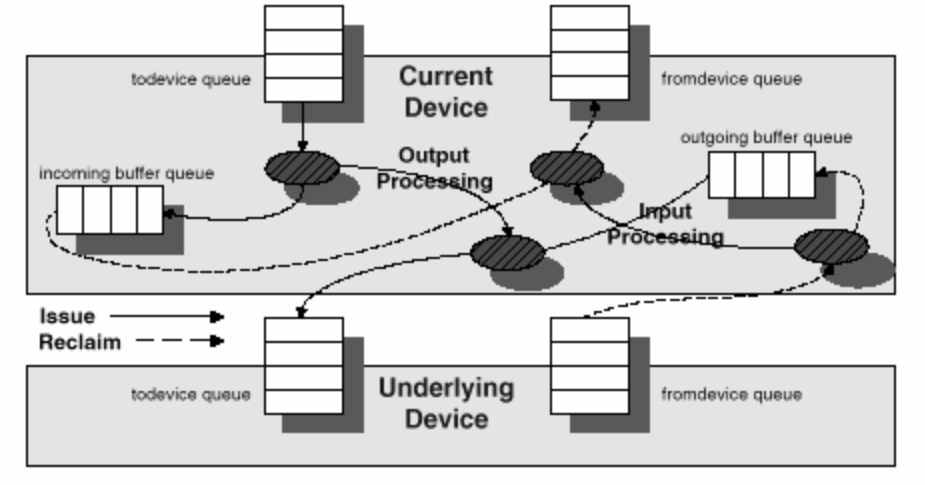

Figure 20 Buffer Flow for a Copying Stacking Device

In many cases it may be appropriate to partition signal processing functions within stackable devices, instead of task or XDAIS socket nodes. For example, a task may be generating data samples in a particular format and sending this data to a stack of devices. By simply swapping in a different stacking device into the stack (without modifying the source task), the data can be provided to the terminating device in a different format.

To create a new device node for a peripheral, the node developer must provide seven driver-specific functions, corresponding to: device open, close, idle, and control; issuing and reclaiming buffers to/from the device; and signaling when the device is ready for I/O. Often generic device functions can be used for some of the seven device functions.

DSP/BIOS Bridge does not impose any special requirement on device drivers; once a driver works in DSP/BIOS-II (using SIO), it will work without modification as a device node in DSP/BIOS Bridge (using STRM).

1.1.1 Communication Paths

Device nodes do not support node-to-node or node-to-GPP messaging. The communication paths between device and task and XDAIS socket nodes are shown above, in Figures 11, 12, and 15.

1.1.2 Execution Timeline

The execution model for a device node differs from that of other node types. Device node functions are not invoked explicitly by the RM Server; some are called indirectly as the RM Server calls a task node's create-phase function and the node opens a stream to the device, and others are invoked when nodes created by the RM Server perform device I/O in their execute phase. A device node's I/O functions are run

in the context of the task it is communicating with; device nodes do not have their own task context. For devices managing a physical device, some device operations can occur in an interrupt context.

The following table shows the execution timeline for a device node. The first column indicates the execution-phase of a task node, the second column indicates RM server actions for the task node, the third column indicates task node-specific operations, and the fourth column shows the related execution of device node functions (for the device node connected to the task node).

Table 12 Execution Timeline for Device Node

Node Phase	RM Server Does:	Node Does:	Device Function:
Create	■ allocates environment structure for node ■ creates task for node in inactive state ■ sets up messaging for node ■ calls node's create function ⇒		
		■ allocates an application-specific context structure to hold instance data for this instance of the node	
		■ allocates **all** application-specific resources needed by the node, placing resource handles in the context structure ■ calls STRM_create to open streams ⇒	
			Dxx_open()
	■ sends reply to RM, passing node's create function return value		
Execute	■ sets the node's execute function to that indicated by the RM ■ responds to RM that going to launch the node ■ launches the node by changing the node's priority from 'inactive', to it's runtime priority ⇒		

Node Phase	RM Server Does:	Node Does:	Device Function:
		■ does real-time processing, monitors for messages from GPP, calls STRM functions to stream data ⇒	
			Dxx_issue() Dxx_reclaim() Dxx_ctrl() Dxx_ready() Dxx_idle()
	■ sends message to RM indicating node's execute function return code		
Delete	■ calls the node's delete-phase function ⇒		
		■ releases all application-specific resources allocated in the create phase ■ deletes data streams ⇒	
			Dxx_idle() Dxx_close()
		■ releases the context structure for the current instance of the node	
	■ deletes the node's task ■ releases the node's environment and messaging resources ■ sends a reply to the RM, passing the node's delete function return value		

1.1.3 Function Signatures

Each device node must provide seven functions in a driver function table. Listed below are the signatures for these functions, along with a brief description of each function's duties. For more details, refer to the *TMS320 DSP/BIOS User's Guide*.

1.1.3.1 Dxx_open

The signature for the device open function is:

```
Int Dxx_open(DEV_Handle device, String name);
```

Dxx_open will be called when **STRM_create** is called by a device client to open a data stream. [Specifically, **STRM_create** will: match the specified device name via the table of configured devices, allocate and initialize a device object, create `todevice` and `fromdevice` queues, and then call the device's **Dxx_open** function.] **Dxx_open** parses any (optional) arguments from the device name string and allocates and initialize a device specific object to maintain the device state.

1.1.3.2 Dxx_close

The signature for the device close function is:

```
Int Dxx_close(DEV_Handle device);
```

Dxx_close is called by **STRM_delete** as a stream is being shutdown. After calling **Dxx_close**, **STRM_delete** will delete the device queues and the stream object. Note: since **STRM_delete** frees device queues and the stream object, it is imperative that the **Dxx_close** function inactivate the device, even if it detects problems during close of the device.

1.1.3.3 Dxx_idle

The signature for the device idle function is:

```
Int Dxx_idle(DEV_Handle device, Bool flush);
```

Dxx_idle is called by **STRM_idle** and **STRM_delete** to place the device in its idle state and recycle frames to the appropriate queue. The *flush* parameter determines what the device should do when idling an output stream: if *flush* is TRUE, all pending output data is discarded, if *flush* is FALSE, the **Dxx_idle** function should not return until all pending output data has been rendered.

1.1.3.4 Dxx_ctrl

The signature for the device control function is:

```
Int Dxx_ctrl(DEV_Handle device, Uns cmd, Arg arg);
```

Dxx_ctrl is called by **STRM_control** to perform a device-specific control operation, as designated by the *cmd* and *arg* parameters.

1.1.3.5 Dxx_issue

The signature for the device issue function is:

```
Int Dxx_issue(DEV_Handle device);
```

Dxx_issue is called by **STRM_issue** to notify the device that a new buffer frame has been submitted to the device on its `todevice` queue. For output streams, **Dxx_issue** performs (optional) device-specific processing on the data buffer; for input streams **Dxx_issue** simply submits empty buffers to the device. **Dxx_issue** must return without blocking.

1.1.3.6 Dxx_reclaim

The signature for the device reclaim function is:

```
Int Dxx_reclaim(DEV_Handle device)
```

Dxx_reclaim is called by **STRM_reclaim** to retrieve a buffer back from a device. For output devices, **Dxx_reclaim** waits until an output buffer frame has been emptied, and then it places the emptied frame on the `fromdevice` queue. For input devices, **Dxx_reclaim** waits until an input frame has been filled with the number of bytes of data requested, then it does (optional) device-specific processing on the input data, and places the filled frame on the `fromdevice` queue. **Dxx_reclaim** must block until a buffer frame is placed on the `fromdevice` queue, or until a stream timeout occurs.

1.1.3.7 Dxx_ready

The signature for the device ready function is:

```
Bool Dxx_ready(DEV_Handle device, SEM_Handle sem)
```

Dxx_ready is called by **NODE_wait** to determine if the device is ready for an I/O operation, i.e., a buffer can be reclaimed from the stream immediately, without blocking. In addition to this poll of the device ready state, **NODE_wait** uses the **Dxx_ready** call to register a single semaphore with a group of devices. The semaphore will be signaled by devices whey they are ready for I/O, allowing a **NODE_wait** client to be signaled when one of a set of streams is ready for I/O.

1.1.4 Execution Context

The following table summarizes the execution context for each Device Node function.

Table 13 Execution Context for Device Node

Function	Execution Context
Dxx_open	RM Server (node create phase)
Dxx_close	RM Server (node delete phase)
Dxx_issue	Controlling task for device
Dxx_reclaim	Controlling task for device
Dxx_idle	Controlling task for device; RM Server (node delete phase)
Dxx_ctrl	Controlling task for device
Dxx_ready	Controlling task for device

5 DSP/BIOS Bridge Configuration Database

5.1 Overview

This chapter summarizes the DSP/BIOS Bridge node configuration process. This process is applicable to two types of developers:

- The **node creator**. This is the developer who is writing the program code to implement the functionality of the node. In many cases the node creator will be an independent software vendor (ISV), developing code for a variety of target platforms.
- The **node integrator**. This is the original equipment manufacturer (OEM) who integrates different nodes together to form an application for their platform. Note that OEMs may take on the role of node creator as well, developing nodes specifically for their platform.

This chapter focuses on what node configuration information needs to be defined for a node to be usable by DSP/BIOS Bridge, and describes the process the developer uses to do this.

5.1.1 Definitions

First, some definitions:

DCD – The DCD is a data repository that resides on the GPP 'host' processor. This database is accessed at runtime to provide platform and node information to the Resource Manager. DCD stands for **D**ynamic **C**onfiguration **D**atabase.

DSP/BIOS Bridge Node Configuration – The procedure associated with the definition of DSP/BIOS Bridge node attributes.

CDB – CDB stands for **C**onfiguration **D**ata**B**ase, an integral part of DSP/BIOS. Code Composer Studio and its Graphical Configuration Tool (GCONF) uses CDB to manage the creation, configuration, and code generation of DSP/BIOS objects. (For more information, please refer to *TMS320 DSP/BIOS User's Guide*).

CDB Seed – A CDB seed, also known as a CDB Configuration File, is a file that describes a DSP/BIOS configuration.

UUID – A universally unique identifier. For details see:
http://www.opengroup.org/onlinepubs/009629399/apdxa.htm.

5.1.2 Purpose of the Node Configuration Process

5.1.2.1 For Node Creator

A node creator applies the node configuration process to define the attributes of DSP/BIOS Bridge nodes, such as an MP3 algorithm or a speech recognition engine. Node attributes may describe the resource requirements, setup information, or any other data relevant to a DSP/BIOS Bridge node component. Data created through the configuration process are encapsulated in the DCD, which is accessed by the Resource Manager at runtime. The Resource Manager, in turn, uses node configuration data retrieved from the database to instantiate DSP nodes required by user applications.

5.1.2.2 For Node Integrator

An integrator utilizes the node configuration process to integrate DSP/BIOS Bridge nodes into an embedded system, such as a smart-phone or a wireless appliance. The process enables the integrator to modify the properties of nodes to comply with requirements of the embedded system.

5.1.3 Usage Model

The following figure summarizes the workflow of node configuration within the DCD.

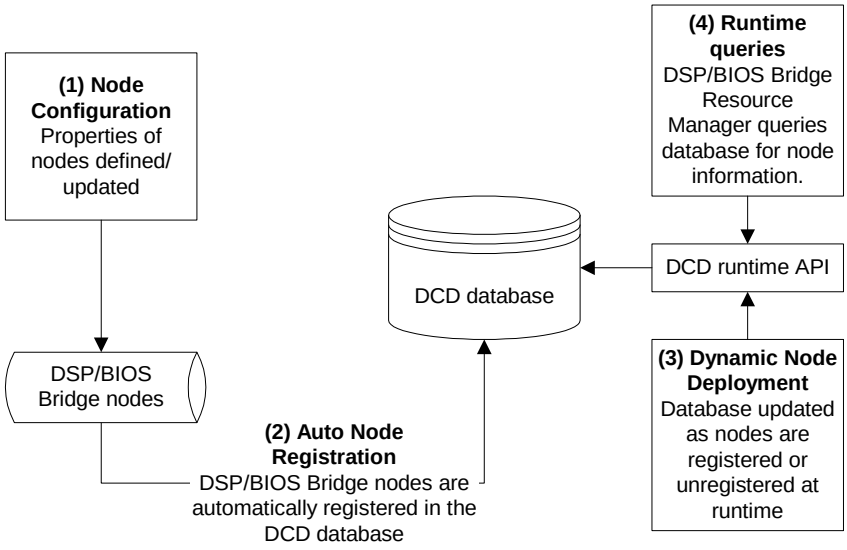

Figure 21 DCD Workflow

1. **Development:** Node creator configures DSP/BIOS Bridge nodes, and node integrator configures nodes into the embedded system.

2. **Runtime:** DSP/BIOS Bridge software framework automatically registers configured nodes into the DCD.

3. **Runtime (optional):** Integrator or Resource Manager dynamically deploys new nodes and updates the DCD.

4. **Runtime:** An application developer, through the Resource Manager, indirectly performs DCD queries to instantiate new nodes.

5.2 DSP/BIOS Bridge Node Configuration

The DSP/BIOS Bridge node configuration process gives a developer the ability to create, delete and modify attributes associated with DSP/BIOS Bridge node objects. For further details about the DSP/BIOS Bridge Node Configuration please refer to DSP-side Bridge documentation (to be written in the future).

5.2.1 DSP/BIOS Bridge Object Specification

For a node to be usable by DSP/BIOS Bridge, certain configuration parameters, or attributes, must be defined for the node. These attributes will enable the Resource Manager to properly create and launch the node on the DSP at runtime. The attributes that need to be defined for a node include those listed in the following table. The node type will determine which attributes from the set need to be defined. For example, for a device node only the first 3 attributes need to be defined (see *Example: Configuring a Device Node* below).

Table 14 Bridge Object Specification

Attribute	Description
DSP_UUID	A universally unique identifier for the node.

Attribute	Description
Node name	A descriptive name for the node.
Node type	The node type: TASKNODE, DAISNODE, or DEVNODE.
Runtime priority	The node's runtime priority within the DSP OS.
Stack size (words)	Stack size for node.
System stack size (words)	System stack size for node.
Stack segment	Memory segment for the node's stack.
Maximum messages queued to node	Max frames to reserve for the node.
# of node input streams	Number of input data streams used by the node.
# of node output streams	Number of output data streams used by the node.
Timeout value of blocking calls	The implicit GPP side timeout (in milliseconds) for blocking **`DSPNode_`** calls which communicate with the Resource Manager Server (**`DSPNode_Create`**, **`DSPNode_Run`**, **`DSPNode_Delete`**, **`DSPNode_Terminate`**, and **`DSPNode_ChangePriority`**).
Load type of node	Load type of node; static, overlay, or dynamic.
Name of structure containing iAlg interface	Defined by user when node type is DAISNODE.
Uses DMA	Selectable when node type is DAISNODE.
Create phase function name	Node's create function name, in format: <node>_<vendor>_create.
Execute phase function name	Node's execute function name, in format: <node>_<vendor>_execute.
Delete phase function name	Node'e delete function name, in format: <node>_<vendor>_delete.

Once the above attributes have been determined, the next step is to configure these attributes into the DCD using the DSP/BIOS Bridge Configuration Tool.

5.2.2 Node Configuration

For configuring Node using TCONF please refer to the DSP-side Bridge documentation (to be written in the future).

5.2.2.1 Node Creation

For detailed procedure about creating a new DSP/BIOS Bridge node please refer to the DSP-side Bridge documentation (to be written in the future).

5.2.2.2 Node Deletion

For detailed procedure about deleting a DSP/BIOS Bridge node please refer to the DSP-side Bridge documentation (to be written in the future).

5.2.2.3 Node Modification

For detailed procedure about modifying the attributes of existing DSP/BIOS Bridge node please refer to the DSP-side Bridge documentation (to be written in the future).

5.2.3 DMAN: DMA Manager Configuration.

OMAP 2420/2430/3430 DMA hardware (MGS3) supports six physical DMA channels. The DMAN module manages all physical DMA channel resources available for use by XDAIS algorithms. If any of the physical DMA channels are dedicated to system or application use, the system integrator has to configure DMAN. For more information about configuring the DMAN please refer to DSP-side Bridge documentation (to be written in the future).

5.2.4 Configuring a socket node that uses DMA resources

For detailed procedure to configure a socket node that uses DMA resources please refer to the DSP-side Bridge documentation (to be written in the future).

5.2.5 Example: Configuring a Device Node

There are two steps required to configure a device node: 1) configure the driver as a user-defined DSP/BIOS device, and 2) configure the device as a DSP/BIOS Bridge node.

5.2.5.1 Configuring a User-Defined Device

For detailed procedure to configure a user-defined device please refer to DSP-side Bridge documentation (to be written in the future).

5.2.5.2 Configuring a User-Defined Device as a Device Node

For detailed procedure to configure the new device as a device node please refer to DSP-side Bridge documentation (to be written in the future).

5.2.6 DSP Base Image Generation

When a configuration file is saved, the Configuration Tool automatically generates assembly source and header files, and a linker command file that contains all attributes for all nodes defined in the configuration. These files can be assembled and linked with other program files, to create the *base image* that will be loaded to the DSP at boot time.

The node configuration information is generated as COPY sections in the resulting COFF file; this configuration information is extracted from the file at runtime by the Resource Manager on the GPP, but is not actually loaded into DSP memory space.

5.2.7 DSP/BIOS Bridge Object Registration

All DSP nodes that are configured and built within a DSP base image are “auto-registered” into the DCD database during DSP boot-up.

Dynamic nodes (not in the base image) are registered to the DCD database via a separate utility.

(See section 8.2 Registering the Dynamic Libraries).

5.2.8 Node Configuration Constraints

During the configuration process, the node developer and/or integrator must specify valid values for each node property under the “General” and “Functions” tabs. The node configuration constraint table below lists possible values for each property in a DSP/BIOS Bridge node:

Table 15 Node parameters

Node property name	Valid values
DSP_UUID	A capitalized UUID with underscore as separators. Example: 7C68FD53_2E0C_11D5_B773_006008BDB66F
Node name	Node name with the format <name>_<vendors>. Example: PING_TI.
Node type	Node type. Valid values are TASKNODE, DAISNODE, DEVNODE.
Runtime priority	Node runtime priority. Valid values are 1 – 15.
Stack size (words)	Stack size for node. This value is dependent on the system state; should be fine-tuned.
System stack size (words)	System stack size for node. This value is dependent on the system state and should be fine-tuned.
Stack segment	Defaults to 0. Do not modify.
Maximum messages queued to node	Defaults to 1. Valid values are 1 – 128.
# of node input streams	Valid values are 0 – 8.
# of node output streams	Valid values are 0 – 8.
Timeout value of GPP blocking calls (milliseconds)	Defaults to 10000 milliseconds (10 seconds). Valid values are 0 – INFINITE.
Load type of node	Load type of node. Valid values are static, overlay, and dynamic.
Name of structure containing iAlg interface	Defined by user when node type is DAISNODE.
Uses DMA	Selectable when node type is DAISNODE.
Create phase function name	Must be of the format <node>_<vendor>_create. Example: PING_TI _create
Execute phase function name	Must be of the format <node>_<vendor>_execute. Example: PING_TI _execute.

Node property name	Valid values
Delete phase function name	Must be of the format <node>_<vendor>_delete. Example: PING_TI _delete

6 DSP/BIOS Bridge Examples

6.1 Examples Provided in Linux DSP/BIOS Bridge

The examples included with this release include applications demonstrating XDAIS algorithm integration, GPP/DSP messaging, and data streaming.

The GPP side examples are installed under the **\dspbridge\samples** sub-directory, the corresponding DSP side is stored under DSP sources **ti\dspbridge\dsp\samples** directory. The DSP task nodes for these examples are integrated into the sample DSP base image, which is built in a platform-specific directory.

6.2 The Ping Application

The Ping program illustrates the fundamental control and message passing mechanisms that are used between GPP applications and BIOS tasks. In particular, this example demonstrates:

- How to create a DSP task node with initial arguments.
- How to start, run, and finally terminate the DSP task node.
- How to generate DSP messages at a pre-specified periodic rate.
- How to retrieve periodic messages sent from the DSP node, using the DSP/BIOS Bridge Node event notifications.
- In addition, this example also introduces:
- How a node may be instantiated multiple times, and how node "class" data can be safely shared between the multiple instances.
- The Design by Contract[1] programming method (on DSP side) as a means to formally assert node function requirements.

The GPP files for this example reside under the **\samples\ping\linux** directory. The DSP files reside under the DSP sources **ti\dspbridge\dsp\samples** directory.

This section covers the details of the example's design, implementation, and usage.

6.2.1 DSP/BIOS Bridge APIs Demonstrated

The following DSP/BIOS Bridge APIs are used in this example:

6.2.1.1 GPP APIs

- **DSPProcessor_Attach**
- **DSPNode_Allocate**
- **DSPNode_Create**
- **DSPNode_RegisterNotify**
- **DSPNode_Run**
- **DSPNode_GetMessage**
- **DSPNode_Terminate**
- **DSPNode_Delete**
- **DSPProcessor_Detach**

6.2.1.2 DSP APIs

- **NODE_getMsg**
- **NODE_putMsg**

6.2.2 Ping Program Design

The purpose of the ping application is to demonstrate how a DSP task can send periodic messages to a controlling GPP program. Each message will convey some interesting bits of information from the DSP, and upon receipt of each message the GPP program will display the message to the console window. The ping program will run continuously until the user terminates it.

The DSP code is constructed within a generic DSP/BIOS Bridge task node and follows the 3-phase execution model for DSP/BIOS Bridge task nodes. This task node executes as a separate thread of execution on the DSP.

[1] See Bertrand Meyer, Object-Oriented Software Construction, 2nd Ed., Prentice Hall, 1997.

6.2.3 Implementing the DSP Task Node

This section reviews the DSP/BIOS Bridge DSP task model, and then discusses the implementation of the ping node task. See DSP source **\ti\dspbridge\dsp\samples\ping.c.**

6.2.3.1 DSP/BIOS Bridge Task Model

DSP applications (BIOS tasks) are controlled through the *Resource Manager Server (RM Server).* The RM Server is a special BIOS task created when the DSP platform boots up. It communicates with the DSP/BIOS Bridge driver on the GPP, sets up the BIOS environment on the platform, and then becomes the central controller for the other DSP tasks on the platform. The server responds only to a limited set of command messages from the GPP — the requests involving the creation or destruction of DSP tasks. The RM Server is the DSP/BIOS Bridge component that makes it possible for GPP applications to start and stop tasks on the DSP.

A task created by the RM Server is like any other BIOS task, with the following exceptions:

1. The task must supply three C-callable functions, each with a specific set of arguments particular to the node type;
2. DSP/BIOS Bridge automatically allocates a task environment structure to hold resources for the task. This environment structure is set as part of the task's environment. This environment data may include the node's message queue and frames, and a pointer to a user-defined node context.
3. The task is created *dynamically* by the RM Server, using parameters derived from the DSP/BIOS Bridge node configuration database.
4. The task parameters are configured under the DSP/BIOS Bridge Node Manager rather than under the TSK Manager in the DSP/BIOS Configuration tool.
5. Since several nodes from different vendors may be statically linked together, externally visible symbols, including the three C-callable node functions, must be prefixed by **<NODE_NAME>_<VENDOR>_** to help prevent name collision.
6. The BIOS task is created, started, terminated, and its task priority is set under the control of the GPP.

Once a DSP task node is created, all application specific message and streaming data traffic between the task and the GPP are handled without the intervention of the RM Server task. Also, with some restrictions in usage, all of the BIOS facilities are directly available to these tasks—including memory allocation, instrumentation, device independent stream I/O, and task scheduling.

6.2.3.2 Task Context Data

Each DSP/BIOS Bridge task node must provide three functions.

1. **`create`** allocates task resources, including memory and I/O, and initializes a task context data structure.
2. **`execute`** performs the actual signal processing functions using the resources set up by the create function
3. **`delete`** frees the resources allocated by the create function

The RM Server calls each function with a node environment argument (`NODE_ENVPTR` *node*), in which the application can store its own node context. Tasks are required to maintain and access their application specific data through this context structure. This context supports the ability of DSP/BIOS Bridge to create multiple instances of a task node. [R.D3]

In the case of the ping DSP task, we need to keep track of the delay argument initially passed from the GPP application and the instance number of this task node (anticipating that we will create multiple instantiations, and we are interested in distinguishing between them). We define a structure which will be passed between create, execute, and delete functions through the `NODE_ENVPTR:`

```
/*
 *  ======== PING_TI_Obj ========
 *  PING task node context.
 */
typedef struct PING_TI_Obj {
```

```
    Uns delay;         /* Time delay between messages in milliseconds.
*/
    Uns instance;      /* Instance number of this PING node.
*/
} PING_TI_Obj;
```

6.2.3.3 Task Setup and Teardown

The ping task context is created and initialized during the task's create phase, then stored in the node environment structure's `moreEnv` field. This includes the delay parameter and task node instance number.

6.2.3.3.1 *NODE INSTANCE AND NODE CLASS DATA*

The task node instance is determined by keeping a (global) count of the number of successful create calls made on this node. Note that this count *cannot* be kept in the node context, since it must be accessible from all ping node instances. Therefore, it must be global.

Such global data shared between multiple instances is similar to static member data in a C++ class, which can be shared between all instances of the class.

It is important to realize that such "node class" data should only be modified within the context of a critical section. The DSP/BIOS Bridge RM Server provides this context during both the create and delete phases, since both functions run in the context of this single, non-reentrant server thread. However, during the execute phase, which executes as a separate BIOS task, more care must be taken to safely modify node class data.

6.2.3.3.2 *CREATE PHASE*

```
/* Node 'class' data, shared between node instances: */
static Uns PING_TI_instances = 0; /* Since global, must use node prefix.
*/

/*
 *  ======== PING_TI_create ========
 */
RMS_STATUS PING_TI_create(Int argLength, Char * argData,
              Int numInStreams, RMS_WORD inDef[],
              Int numOutStreams, RMS_WORD outDef[], NODE_EnvPtr node)
{
    PING_TI_Obj     *pingPtr;
    Int             delay;
    RMS_STATUS      retval;

    /* Assert Requires:                   */
    DBC_require(argLength > 0);
    DBC_require(argData != NULL);

    /* Check input argument. */
    delay = atoi(argData);
    if (delay >= 1)  {

       /* Allocate task node context object. */
       if ((pingPtr = MEM_calloc(PINGMEMSEG, sizeof(PING_TI_Obj), 0))
            != MEM_ILLEGAL) {

            /* Initialize Ping node context structure. */
            pingPtr->delay = (Uns)delay;

            /*
             *  Set the instance number of this node.
```

```
                 */
                pingPtr->instance = ++PING_TI_instances;

                /* Save Ping node context for use in execute and delete
                phases. */
                node->moreEnv = pingPtr;
                retval = RMS_EOK;
            }
            else {
                /* If unable to allocate object, return out of memory error.
                */
                retval = RMS_EOUTOFMEMORY;
            }
        }
        else {
            LOG_printf(TRACE, "PING_TI_create: invalid delay = %d\n", delay);
            retval = RMS_EFAIL;
        }

        LOG_printf(TRACE,"PING_TI_create: retval = 0x%x; delay = %u\n",
        retval, delay);

        /* Assert Ensures */
        DBC_ensure((retval == RMS_EFAIL) || (retval == RMS_EOUTOFMEMORY) ||
         ((retval == RMS_EOK) && (node->moreEnv != NULL) && (pingPtr->delay
         >= 1)));

        return (retval);
    }
```

Finally, the create phase must return a success (`RMS_EOK`) or failure code back to the RM Server.

6.2.3.3.3 *DELETE PHASE*

The delete phase of the ping task is responsible for deallocating all objects allocated during the create phase.

The delete phase first accesses its node context through the `node->moreEnv` pointer and then destroys the node context. It is also responsible for decrementing the number of outstanding ping node instances by one:

```
    /*
     *  ======== PING_TI_delete ========
     *  Ensures:   Failed:  returns RMS_EFREE: couldn't free context object;
    or
     *             Success: returns RMS_EOK and PING_TI_Obj freed.
     */
    RMS_STATUS PING_TI_delete(NODE_EnvPtr node)
    {
        PING_TI_Obj  *pingPtr;
        RMS_STATUS    retval = RMS_EOK;

        /* Dereference context structure stored in the node variable. */
        pingPtr = node->moreEnv;

        if (pingPtr != NULL) {

            /* We now have one less PING node instance to kick around. */
            PING_TI_instances--;
```

```
        /* Free Ping context structure. */
        if (!MEM_free(PINGMEMSEG, pingPtr, sizeof(PING_TI_Obj)))  {
             retval = RMS_EFREE;
        }
    }
    LOG_printf(TRACE,"PING_TI_delete: retval = %x", retval);

    /* Assert Ensures */
    DBC_ensure((retval == RMS_EFREE) || (retval == RMS_EOK));

    return (retval);
}
```

Remember that DSP/BIOS Bridge is a dynamic task environment, and DSP tasks will be coming into and going out of existence while the device is in operation. To prevent memory leaks, it is imperative that all dynamically created objects are finally destroyed.

6.2.3.3.4 *A NOTE ON DESIGN BY CONTRACT*

Design by Contract (DBC) is a method of using assertions in one's code to explicitly document and validate the often implicit assumptions made by programmers. Benefits include:

1. Reduces redundant parameter checking, by pushing parameter validation up out of system modules to "untrusted" clients (typically, the user interface).
2. Makes explicit the programmer's implicit assumptions.
3. Helps prevent errors from being programmed in from the start by forcing the programmer to think about implicit assumptions *during the act of programming*.
4. Assertions catch bugs earlier during the product testing phase.

To use DBC in DSP/BIOS Bridge DSP programming, simply include the **dbc.h** header file, which contains the following macros:

```
#define DBC_assert(c) if (!(c)) { \
        SYS_abort("Assertion Failed: file=%s, line=%d.\n", __FILE__, \
        __LINE__); }
#define DBC_require DBC_assert
#define DBC_ensure DBC_assert
```

DBC is an especially important technique in embedded programming, where the drive to reduce program complexity and code size limits the amount of parameter validation programmers are willing to do. Careful use of DBC will alleviate this issue.

This technique is fully developed by Bertrand Meyer in Object-Oriented Software Construction, 2nd Ed., Prentice Hall, 1997.

6.2.3.3.5 *EXECUTE PHASE*

As previously mentioned, the create and delete functions of a task node are called in the context of the RM Server thread, which is a BIOS thread running at a pre-determined priority set by DSP/BIOS Bridge.

The execute function of a task node, however, is run as the part of the thread function to a BIOS **TSK_create** call, at a task priority determined by the priority field setting in the DSP/BIOS Bridge configuration database. As such, the execute phase of a node is a full-fledged BIOS thread *running independently* of the RM Server thread.

DSP/BIOS Bridge supports two methods of communication between DSP nodes and GPP tasks: 1) Streaming and 2) Messaging. Streaming is typically used to transfer large amounts of data, whereas messaging is typically used for short status and control messages. Since this ping node is basically providing periodic status messages to the GPP application, it is appropriate to use messaging in this example.

A requirement of this node is to run indefinitely until commanded by the GPP to terminate [R.D1]. Therefore, the execute thread must 'listen' for an exit signal from the GPP. The DSP/BIOS Bridge GPP API provides a function **DSPNode_Terminate** for this purpose. **DSPNode_Terminate** sends a system-defined message RMS_EXIT to the node's input message queue. Nodes which receive the RMS_EXIT command must exit the execute function immediately and therefore must remain responsive to this signal.

Applications must never send the RMS_EXIT message command directly to a node from the GPP using **DSPNode_PutMessage** or from another DSP node using **NODE_putMsg**. RMS_EXIT is reserved for use by the DSP/BIOS Bridge Resource Manager only.

Messages are sent to the GPP using **NODE_putMsg** passing the NODE_TOGPP constant as the destination. One solution would be to create a loop that sleeps for ever (NODE_FOREVER) until PING message or RMS_EXIT message from GPP is received.

A better approach would be to use the built-in delay parameter in **NODE_getMsg** itself, which will wait for a message to the node forever:

```
/* Wait for RMS exit message from GPP: */
NODE_getMsg(node, &msg, NODE_FOREVER);
```

Finally, the last requirement is to provide some status information on each message [R.D4].

```
RMS_STATUS PING_TI_execute(NODE_EnvPtr node)
{
    RMS_DSPMSG   msg;
    RMS_WORD cnt = 0;
    Uns          instance;
    MEM_Stat     memUsage;
    Bool         status;

    /* Assert Requires */
    DBC_require(node->moreEnv != NULL);

    LOG_printf(TRACE, "PING_TI_execute: Entered ...\n");
    PING_print("PING_TI_execute: Entered ...\n");

    /* Extract attributes from node context object: */
    instance = ((PING_TI_Obj *)node->moreEnv)->instance;

    msg.cmd = ~(RMS_EXIT);  /* Initialize */
    do {
       /* Wait (with delay timeout) for RMS exit messsage from GPP: */
       NODE_getMsg(node, &msg, NODE_FOREVER);
       LOG_printf(TRACE,
              "PING_TI_execute: Received ping from GPP: cmd = 0x%x; cnt
       = 0x%x\n",
              msg.cmd, msg.arg1);
```

```
            PING_print("PING_TI_execute: Received ping from GPP: cmd = 0x%x;
            cnt = 0x%x\n",
                   msg.cmd, msg.arg1);

            /*
             *  Unless a DSPNode_Terminate() call was made from GPP,
             *  NODE_getMsg will have timed out, with msg.cmd left as
            RMS_USER+1.
             */
            if (msg.cmd != RMS_EXIT) {
               MEM_stat(0, &memUsage);

               /* Setup message fields:  */
               msg.cmd  = RMS_USER + instance;   /* Node instance number */
               msg.arg1 = cnt++;                 /* Message count        */
               msg.arg2 = (RMS_WORD)(memUsage.size - memUsage.used); /* MAU's
               free */

               /* Send message, *without* blocking until message queue is
               ready.   */
               status = NODE_putMsg(node, NODE_TOGPP, &msg, 0);
               if (!status)  {
                 /*
                  *  Message queue not emptied, so GPP application hasn't
               pulled the
                  *  last message.  In this case, we drop it, and proceed.
                  *  If we do *not* wish to drop messages, set timeout
               parameter to
                  *  NODE_FOREVER instead of zero.  In this case, the GPP will
               flow
                  *  control the message rate.
                  */
                 LOG_printf(TRACE,
                      "PING_TI_execute: dropped acknowledgement msg to GPP:
                      cmd = 0x%x; cnt = 0x%x\n",
                      msg.cmd, msg.arg1);
                 PING_print("PING_TI_execute: dropped acknowledgement msg GPP:
                 cmd = 0x%x; cnt = 0x%x\n", msg.cmd, msg.arg1);
               }
            }

         }while(msg.cmd != RMS_EXIT);

         LOG_printf(TRACE, "PING_TI_execute: Exited\n");
         PING_print("PING_TI_execute: Exited\n");

         return (RMS_EOK);
      }
```

There are a few points worth noting about this code:

1. This code is designed to ping the GPP once and wait for a message back from GPP using **`NODE_putMsg and NODE_getMsg`**. Since the the *`timeout`* parameter is set to `NODE_FOREVER` instead of zero, **`NODE_getMsg`** waits until it receives a ping message or RMS_EXIT.

2. Message command IDs (values for `msg.cmd`) available to the user are in the range `{RMS_USER .. RMS_MAXUSERCODES}`. Other values are reserved for use by DSP/BIOS Bridge.

6.2.3.4 Configuring the Ping Task Node

Finally, before linking the ping task node into the base image, the node's configuration data must be entered via the DSP/BIOS Configuration tool. For futher information about configuring a Task Node please refer to DSP-side Bridge documentation (to be written in the future).

6.2.4 Implementing the GPP Program

The GPP program is an application named **ping.out**. For simplicity, the example is written as a console application and provides a minimal user interface. The GPP program requires the DSP PING task node to be loaded on the DSP. If not, the allocation of the PING task node will fail from the GPP side.

This section will describe the main actions of the code and highlight some important API usage points in the source code files **ping.c** in the **\dspbridge\samples** directory.

The example is organized into a set of subroutines, each function calling one or more DSP/BIOS Bridge APIs to accomplish some action with the DSP. The main functions are as follows, with a quick rundown:

`ProcessArgs`	Process command line arguments to pass into the setup routines.
`AttachProcessor`	Calls **`DSPProcessor_Attach`**.
`CreateNode`	Calls **`DSPNode_Allocate, DSPNode_Create`** (calls node's *create* function), sets up notifications to an event (via **`DSPNode_RegisterNotify`**), then creates a message box thread that signals the application to exit.
`RunNode`	Calls **`DSPNode_Run,`** then loops 'msgCount' number of times doing sequence of operations like sending a PING message by calling **`DSPNode_PutMessage,`** waiting for either the event handle or the message box thread for be signaled. If the event handle is signaled, calls **`DSPNode_GetMessage`** to retrieve and display the DSP message. The 'msgCount' is passed by the user from command line while executing ping sample. If the loop completes executing for 'msgCount' number of times, **`DSPNode_Terminate`** is called to signal the task node's *execute* function to exit.
`DestroyNode`	Calls **`DSPNode_Delete`** to call the node's *delete* function, and deallocate GPP side resources. Also frees up event and thread handles.
`DetachProcessor`	Calls **`DSPProcessor_Detach.`**

In addition, a common structure is defined (PING_TASK) to pass intermediate context between functions (preferable to global variables):

```
/* Ping task context data structure - passed between functions in this
module */
typedef struct {
    DSP_HPROCESSOR  hProcessor;     /* Handle to processor.
*/
    DSP_HNODE       hNode;          /* Handle to node.
*/
    HANDLE          hEvent;         /* Event to be signaled by DSP
*/
    UINT            msgCount;       /* Number of messages DSP sends
*/
} PING_TASK;
```

6.2.4.1 Node State Transitions are Synchronous

It is worth noting that except for the notification functions, all interactions with the Resource Manager Server by the **`DSPNode_`** APIs are synchronous. This implies, for example, that after returning from a successful call to **`DSPNode_Run,`** the DSP node has definitely *started* running. In fact, the Resource Manager will block any other requests to the RM Server until the current request is complete. Thus, there

is no possibility of a race condition caused by attempting to communicate with the node before the **DSPNode_Run** call completes. The same theory applies to **DSPNode_Create**, **DSPNode_Terminate,** and **DSPNode_Delete.**

Each node state transition function waits for a response from the RM Server with a timeout specified by a timeout value passed in through the DSP_NODEATTRIN structure in **DSPNode_Allocate,** or by the timeout value configured in the DCD for the node being controlled if the DSP_NODEATTRIN parameter is NULL.

6.2.4.2 Passing Initial Task Arguments

The **ping** program prepares initial arguments for the DSP task node. Initial arguments are passed in the form of a DSP_CBDATA structure (defined in the DSP/BIOS Bridge API header file) to **DSPNode_Allocate**:

```
/* The DSP_CBDATA structure.  */
typedef struct {
    ULONG   cbData;
    BYTE    cData[32];
} DSP_CBDATA, *DSP_HCBDATA;
```

The DSP_CBDATA structure is simply a declaration to be cast over a defined argument structure, whose first word is the length of the arguments, and whose next field is a BYTE array containing the arguments.

This essentially creates the ability to convey "command line" type arguments to the DSP node's create phase, which can be parsed on the DSP using standard C library string functions.

6.2.4.3 A Note on Exception Handling

The examples provided in this section do not provide any termination or exception handling capabilities. For example, a page fault, access violation, or divide by zero error would be caught by the system, and cause an immediate exit from the application. Although the GPP OS may take care of freeing threads and other kernel objects, allocated DSP/BIOS Bridge GPP resources may not be freed unless the appropriate **DSPNode_Delete, DSPStream_FreeBuffers, DSPStream_Close**, and **DSPProcessor_Detach** calls were made before exiting.

> You must eventually call **DSPNode_Delete** on a node handle after calling **DSPNode_Create** in order to free resources on *both* the GPP and DSP. If the node's **create** function fails, the resource manager server does *not* automatically call the node's **delete** function. Therefore, even though **DSPNode_Create** returns an error code, the GPP application is *still* responsible for calling **DSPNode_Delete**.

A robust application would take advantage of structured exception handling mechanisms provided by the GPP OS, or other methods to trap exceptions.

6.2.5 Running the Ping Application

The console application **ping.out** takes the following arguments:

[<message_count>]

The default value for the above argument is 50.

Once launched, the console application will start displaying message similar to the following:

- ping.out

```
Ping: Id 1.000000 Msg 1.000000 Mem 29712.000000
```

```
Ping: Id 1.000000 Msg 2.000000 Mem 29712.000000

Ping: Id 1.000000 Msg 3.000000 Mem 29712.000000

Ping: Id 1.000000 Msg 4.000000 Mem 29712.000000

Ping: Id 1.000000 Msg 5.000000 Mem 29712.000000

Ping: Id 1.000000 Msg 6.000000 Mem 29712.000000

 ...
```

A **ping** script can be used to invoke multiple instances of the ping application. The arguments to this script are the number of processes (default is 1), followed by the above argument

If an error occurs, a hexadecimal error code will be printed. Please see the **errbase.h** header file for a description of the error codes.

6.2.5.1 A Note on Data Flow Control

In this example, the DSP determines the rate of messages to the GPP.

This is because the I/O thread in the DSP/BIOS Bridge driver runs at a higher than normal priority in order to be responsive to events from the DSP. When many events are arriving at a high rate, the GPP OS scheduler will allow this driver I/O thread to run, at the possible exclusion of lower priority threads (like the user interface), which may also be ready to run.

In a more typical application, these DSP events would be “flow controlled” by the media API or application to ensure a specific data frame rate to meet the requirements of a particular DSP codec. In these cases, the load on the GPP should be reduced, allowing time for the user interface to respond.

6.2.6 Summary

The GPP console application and task node described in this chapter represents the code necessary for a GPP program using DSP/BIOS Bridge to control and receive messages from a DSP task node.

Though the example only uses messaging, its implementation as a task node means it can be used as a base for adding streaming capability as well.

The **strmcopy** example below will describe how to add data streaming capability to a task node.

6.3 The Stream Copy Application

The **strmcopy** example demonstrates simple data streaming between a GPP task and a DSP task node in DSP/BIOS Bridge. DSP programmers can also use the example as a test harness for running any single-input, single-output signal processing algorithm on the DSP, providing user-defined data from an input file, and capturing data to an output file. The **strmcopy** DSP task is a logical starting point for developing a typical signal processing application.

The GPP files for this example reside under the **\samples\strmcopy\linux** directory. The DSP files reside under the DSP side sources **\ti\dspbridge\dsp\samples** directory.

6.3.1 DSP/BIOS Bridge APIs Demonstrated

The following DSP/BIOS Bridge APIs are used in the application:

6.3.1.1 DSP APIs

- **NODE_getMsg**
- **NODE_wait**
- **STRM_create**
- **STRM_allocateBuffer**
- **STRM_issue**
- **STRM_reclaim**
- **STRM_idle**
- **STRM_freeBuffer**
- **STRM_delete**

6.3.1.2 Linux APIs

- **DSPProcessor_Attach**
- **DSPNode_Allocate**
- **DSPNode_Connect**
- **DSPNode_ConnectEx**
- **DSPNode_Create**
- **DSPNode_Run**
- **DSPStream_Open**
- **DSPStream_AllocateBuffers**
- **DSPStream_Issue**
- **DSPStream_Reclaim**
- **DSPStream_Idle**
- **DSPStream_FreeBuffers**
- **DSPStream_Close**
- **DSPNode_Terminate**
- **DSPNode_Delete**
- **DSPProcessor_Detach**

6.3.2 Strmcopy Program Design

The **strmcopy** example utilizes DSP/BIOS Bridge data streams to enable effective, flexible data transmission between the GPP and the DSP. On the GPP side, the console application opens an input

stream to the DSP and an output stream from the DSP. In addition, the application uses an input file to provide user-defined data to the DSP and an output file to capture data sent from the DSP. On the DSP side, the **strmcopy** task node has one input data stream and one output data stream; the input stream receives data from the GPP, the output stream sends data to the GPP.

The architecture of **strmcopy** is shown in the following figure:

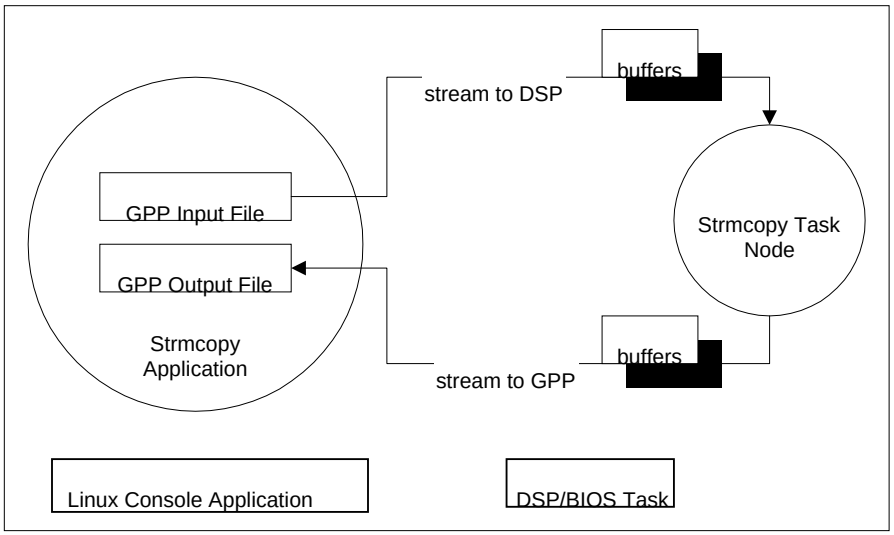

Figure 22 strmcopy Application Architecture

During program execution, the GPP application reads in fixed-length data buffers from the input file and sends them to the DSP on the DSP input data stream. When the DSP task node receives data buffers from the GPP, it performs a simple transfer to copy the data it received on its input data stream onto its output data stream. Finally, as the GPP application receives data from the DSP output data stream, it writes any received data buffers out to the GPP output file.

6.3.2.1 Note on using multiple buffers per data stream

In DSP/BIOS Bridge, the throughput on a given data stream can be dramatically affected by the number of buffers allocated for that stream. This section demonstrates how multi-buffering increases throughput on a data stream by allowing the GPP and DSP to issue and reclaim data more effectively.

In the issue/reclaim buffering model, read/write operations on a stream buffer are synchronous, which means while a process is reading from a buffer, another process cannot write to it, and vice versa. Let us call this the synchronicity rule. The temporal diagram below illustrates how the rule works.

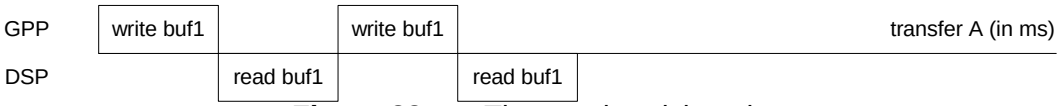

Figure 23 The synchronicity rule

As is evident in the diagram, simultaneous access of *buf1* never occurs.

Now, because the GPP and DSP can operate independently of one another, each processor can operate on multiple buffers at the same time, provided the synchronicity rule is not violated. In the diagram below, we compare two different multi-buffering transfers to the original single-buffer transfer.

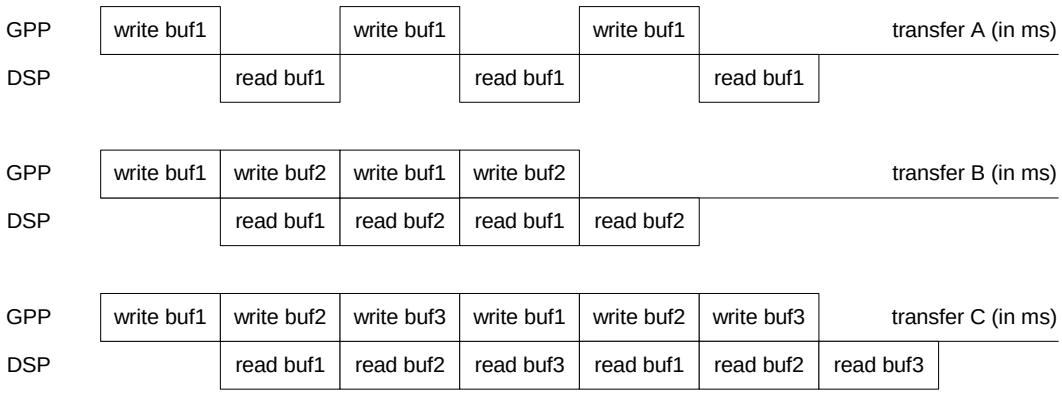

Figure 24 Single-buffering, double-buffering, and triple-buffering

Again, simultaneous access of a buffer never occurs in transfer B or C. In the time it took transfer A to transmit 3 buffers from the GPP to the DSP, transfer B transmitted 4 buffers, and transfer C transmitted 5 buffers. Clearly, multi-buffering results in better throughput than single-buffering.

6.3.3 Implementing the Strmcopy Task Node

Strmcopy, like **ping**, uses a generic DSP/BIOS Bridge task node based on the three-phase task model to perform data streaming between the GPP and the DSP.

A **strmcopy** node context object is created and initialized during the create phase. The node context information includes handles to input/output data streams, pointers to input/output data stream buffers, and size of input/output data stream buffers. A pointer to this context object is stored within the node's environment field `moreEnv`, which is accessible from the node's execute and delete phases.

```
typedef struct {
    STRM_Handle inStream;       /* A STRM input channel. */
    STRM_Handle outStream;      /* A STRM output channel. */
    Uns         *inBuf;         /* A data buffer. */
    Uns         *outBuf;        /* A data buffer. */
    Uns         inSize;         /* Size of input buffer. */
    Uns         outSize;        /* Size of output buffer. */
} StrmcopyObj;
```

6.3.3.1 Create Phase

In **strmcopy**'s create phase function, **`STRMCOPY_TI_create`**, the following occurs:

1. Input and output data streams are created through **`STRM_create`**.

2. Input and output data buffers are allocated on data streams through **`STRM_allocateBuffer`**.

3. A **strmcopy** context object is created and stored into the node environment variable.

```
/*
 *  ======== STRMCOPY_TI_create ========
 */
RMS_STATUS STRMCOPY_TI_create(Int argLength,
                              Char * argData,
                              Int numInStreams,
                              RMS_WORD inDef[],
                              Int numOutStreams,
                              RMS_WORD outDef[],
                              NODE_EnvPtr env)
{
    RMS_StrmDef     *strmDef;
    STRM_Attrs      attrs = STRM_ATTRS;
    StrmcopyObj     *copyObj;
    RMS_STATUS      status = RMS_EOK;

    /* Allocate context structure for this instance of the strmcopy
object. */
    if ((copyObj = MEM_calloc(0, sizeof(StrmcopyObj), 0)) != NULL) {
        /*
         *  Set env->moreEnv to copyObj to be used in execute and delete
         *  phase.
         */
        env->moreEnv = (Ptr)copyObj;
        /* Create input stream. */
        strmDef = (RMS_StrmDef *)inDef[0];
        attrs.nbufs = strmDef->nbufs;
        attrs.segid = strmDef->segid;
        attrs.timeout = strmDef->timeout;
        attrs.align = strmDef->align;

        LOG_printf(TRACE, "STRMCOPY_TI_create: input stream");
        LOG_printf(TRACE, "  bufsize 0x%x, nbufs %d,",
            strmDef->bufsize, attrs.nbufs);
        LOG_printf(TRACE, "  segid %d, timeout %d",
            attrs.segid, attrs.timeout);
        copyObj->inStream = STRM_create(strmDef->name, STRM_INPUT,
            strmDef->bufsize, &attrs);
        copyObj->inSize = strmDef->bufsize;

        /* Create output stream. */
        strmDef = (RMS_StrmDef *)outDef[0];
        attrs.nbufs = strmDef->nbufs;
        attrs.segid = strmDef->segid;
        attrs.timeout = strmDef->timeout;
        attrs.align = strmDef->align;

        LOG_printf(TRACE, "STRMCOPY_TI_create: output stream");
        LOG_printf(TRACE, "  bufsize 0x%x, nbufs %d,",
            strmDef->bufsize, attrs.nbufs);
        LOG_printf(TRACE, "  segid %d, timeout %d",
            attrs.segid, attrs.timeout);

        copyObj->outStream = STRM_create(strmDef->name, STRM_OUTPUT,
            strmDef->bufsize, &attrs);
        copyObj->outSize = strmDef->bufsize;
```

```
            /* Allocate data buffers for data streaming. */
            copyObj->inBuf = STRM_allocateBuffer(copyObj->inStream,
                copyObj->inSize);
            copyObj->outBuf = STRM_allocateBuffer(copyObj->outStream,
                copyObj->outSize);

            /* Verify allocations; if failure, free all allocated objects. */
            if ((copyObj->inStream == NULL) || (copyObj->outStream == NULL))
    {
                status = RMS_ESTREAM;
            }

            if ((copyObj->inBuf == MEM_ILLEGAL) ||
                (copyObj->outBuf == MEM_ILLEGAL)) {
                    status = RMS_EOUTOFMEMORY;
            }
        }
        else {
            /* If unable to allocate object, return RMS_E_OUTOFMEMORY. */
            status = RMS_EOUTOFMEMORY;
        }

        return (status);
    }
```

6.3.3.2 Execute Phase

In **strmcopy**'s execute phase function, **`STRMCOPY_TI_execute`**, the following occurs:

1. DSP task calls **`STRM_issue`** to put an empty buffer **inbuf** to its input data stream.

2. DSP task calls **`NODE_wait`** to wait for data or control messages to become available on its input data stream. **`NODE_wait`** blocks until data or message arrives; if **`NODE_wait`** unblocks due to data arrival, go to step 3.

3. DSP task calls **`STRM_reclaim`** to get a filled buffer **inbuf** from its input data stream. If this call succeeds, it implies the buffer **inbuf** has successfully received data transmitted from the GPP to the DSP.

4. At this point, the received data can be modified by a filter or algorithm. In the case of **strmcopy**, it is simply copied into an output buffer **outbuf**.

5. DSP task calls **`STRM_issue`** to put a filled buffer **outbuf** to its output data stream. If the DSP task is not in the first iteration of the program, then it must call **`STRM_reclaim`** to get an emptied buffer **outbuf** from its output data stream before it can call **`STRM_issue`**. If **`STRM_reclaim`** succeeds, it implies the previous content of buffer **outbuf** has been successfully transferred from the DSP to the GPP.

6. The DSP task repeats steps 1 – 5 for each round of data transfer between the GPP and the DSP.

```
/*
 *  ======== STRMCOPY_TI_execute ========
 */
```

```
RMS_STATUS STRMCOPY_TI_execute(NODE_EnvPtr env)
{
    StrmcopyObj     *copyObj;
    Uns             bytesRead;
    Arg             bufArg;
    RMS_DSPMSG      msg;
    Uns             msgReady;
    Uns             readyStreams;
    Bool            firstFrame = TRUE;

    /* Dereference context structure stored in the env variable. */
    copyObj = (StrmcopyObj*)env->moreEnv;
    LOG_printf(TRACE, "Waiting for first buffer from host...");

    /* Initialize msg structure. */
    msg.cmd = ~(RMS_EXIT);

    /* Prime the input stream with an empty buffer. */
    STRM_issue(copyObj->inStream, copyObj->inBuf, copyObj->inSize,
copyObj->inSize, 0);

    /* Until end of stream indicated, receive and send a host buffer: */
    while (1) {
        /* Wait until input stream is ready, or until GPP message
arrives. */
        readyStreams = NODE_wait(env, &copyObj->inStream, 1,
            NODE_FOREVER, &msgReady);

        /* If a message has arrived. */
        if (msgReady != 0) {
            /* Get message. */
            NODE_getMsg(env, &msg, 0);

            if (msg.cmd == RMS_EXIT) {
                LOG_printf(TRACE, "Got RMS_EXIT");
                break;
            }
        }

        /* If a buffer is ready on the input stream. */
        if (readyStreams != 0) {
            /* If this is the first iteration, use output buffer
directly. */
            if (firstFrame == TRUE) {
                firstFrame = FALSE;
            }
            else {
                /* Reclaim output buffer. */
                STRM_reclaim(copyObj->outStream, (Ptr)&copyObj->outBuf,
NULL, &bufArg);
            }

            /* Get the buffer from the input stream. */
            bytesRead = STRM_reclaim(copyObj->inStream, (Ptr)&copyObj-
>inBuf,
                NULL, &bufArg);

            /* Copy the buffer into the output buffer. */
```

```
            memcpy(copyObj->outBuf, copyObj->inBuf, bytesRead);

            /* Send an output buffer to the output stream. */
            STRM_issue(copyObj->outStream, copyObj->outBuf, bytesRead,
copyObj->outSize,
                0);

            /* Send an empty buffer to the input stream. */
            STRM_issue(copyObj->inStream, copyObj->inBuf, copyObj-
>inSize,
                copyObj->inSize, 0);
        }
    }

    /* Idle open streams before the delete phase. */
    STRM_idle(copyObj->inStream, TRUE);
    STRM_idle(copyObj->outStream, TRUE);

    /* Reclaim any outstanding buffers in the data streams. */
    STRM_reclaim(copyObj->inStream, (Ptr)&copyObj->inBuf, NULL, &bufArg);
    if (!firstFrame) {
        STRM_reclaim(copyObj->outStream, (Ptr)&copyObj->outBuf, NULL,
&bufArg);
    }

    return (RMS_EOK);
}
```

6.3.3.3 Delete Phase

In **strmcopy**'s delete phase function, **STRMCOPY_TI_delete**, the following occurs:

1. Input and output data stream buffers are freed through **STRM_freeBuffer**.
2. Input and output data streams are idled through **STRM_idle**, then deleted through **STRM_delete**.
3. The **strmcopy** context object created during the create phase is deallocated.

```
/*
 *  ======== STRMCOPY_TI_delete ========
 */
RMS_STATUS STRMCOPY_TI_delete(NODE_EnvPtr env)
{
    StrmcopyObj     *copyObj;

    /* Dereference context structure stored in the env variable. */
    copyObj = (StrmcopyObj*)env->moreEnv;

    if (copyObj) {
        if (copyObj->inStream != NULL) {
            /* Free stream buffer. */
            if (copyObj->inBuf != NULL) {
                STRM_freeBuffer(copyObj->inStream, copyObj->inBuf,
                    copyObj->inSize);
            }
```

```
            /* Free stream object. */
            STRM_delete(copyObj->inStream);
        }

        if (copyObj->outStream != NULL) {
            /* Free stream buffer. */
            if (copyObj->outBuf != NULL) {
                STRM_freeBuffer(copyObj->outStream, copyObj->outBuf,
                    copyObj->outSize);
            }

            /* Free stream object. */
            STRM_delete(copyObj->outStream);
        }

        MEM_free(0, copyObj, sizeof(StrmcopyObj));
    }
    return (RMS_EOK);
}
```

6.3.3.4 Configuring the Strmcopy Task node

The strmcopy node's configuration data must be entered via the DSP/BIOS Configuration tool. Note that all DCD information resides in special sections within the DSP base image executable file, and is accessed by the DSP/BIOS Bridge Resource Manager on DSP boot-up.

DSP/BIOS Bridge DSP tasks are configured under the **DSP/BIOS Bridge Node Manager** (not the TSK Manager) in the DSP/BIOS Configuration tool.

Configuration parameters for the **strmcopy** node include:

Table 16 Streamcopy Node Parameters

Node name	STRMCOPY_TI
Node type	TASKNODE
DSP_UUID	8D93706A_1F31_11D5_B314_00B0D0B50194
Runtime priority	3
Stack size (words)	1024
System stack size (words)	256
Stack segment	0
Maximum messages queued to node	1
# of node input streams	1
# of node output streams	1
Timeout value of GPP blocking calls (in milliseconds)	2000
Load type of node	**Static**
Create Phase function name	**STRMCOPY_TI_create**
Execute Phase function name	**STRMCOPY_TI_execute**
Delete Phase function name	**STRMCOPY_TI_delete**

Refer to the **ping** application and *DSP/BIOS Bridge Configuration Database* for more details.

6.3.4 Implementing strmcopy's GPP application

The GPP application for **strmcopy** works hand-in-hand with the DSP task node, and has three abstract functions that correspond to the DSP task node's three phase model: initialize (create), run (execute), and cleanup (delete). The sections below describe the GPP application implementation in more detail.

6.3.4.1 GPP Application Initialization

The strmcopy GPP application makes use of the following context structure:

```
typedef struct {
    DSP_HPROCESSOR      hProcessor;     /* Handle to processor. */
    DSP_UUID            hNodeID;        /* Handle to node ID GUID. */
    DSP_HNODE           hNode;          /* Handle to node. */
    DSP_HSTREAM         hInStream;      /* Handle to node input stream. */
    DSP_HSTREAM         hOutStream;     /* Handle to node output stream. */
    BYTE                **ppInBufs;     /* Input stream buffers. */
    BYTE                **ppOutBufs;    /* Output stream buffers. */
    DSP_NOTIFICATION    hNotification;  /* DSP notification object. */
    INT                 nStrmMode;      /* stream mode */
} STRMCOPY_TASK;
```

During initialization, the GPP console application uses functions **`InitializeProcessor`**, **`InitializeNode`**, and **`InitializeStreams`** to perform various DSP/BIOS Bridge related setup:

1. **`InitializeProcessor`** – Attach to a valid DSP through **`DSPProcessor_Attach`**. Then load and start a DSP image, if necessary.

```
/*
 *  ======== InitializeProcessor ========
 */
static DSP_STATUS InitializeProcessor(STRMCOPY_TASK *copyTask)
{
    DSP_STATUS  status     = DSP_SOK;
    INT      argc = 1 ;
    const CHAR *   argv = DEFAULTDSPEXEC ;

/* Attach to an available DSP (in this case, the 1st DSP). */
    if (DSP_SUCCEEDED(status)) {
        status = DSPProcessor_Attach(DEFAULTDSPUNIT, NULL,
            &copyTask->hProcessor);
        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPProcessor_Attach succeeded.\n");
        }
        else {
            fprintf(stdout, "DSPProcessor_Attach failed. Status =
0x%x\n",
                status);
        }
    }

    if (DSP_SUCCEEDED(status))
```

```
        {
            /* Load strmcopy image on DSP */
            status = DSPProcessor_Load(copyTask->hProcessor, argc, &argv,
    NULL) ;

            if (DSP_SUCCEEDED(status))
            {
               fprintf(stdout, "DSPProcessor_Load succeeded.\n") ;
            }
            else
            {
                fprintf(stdout, "DSPProcessor_Load failed. Status = 0x%x\n",
                    status);
            }
        }

        if (DSP_SUCCEEDED(status))
        {
            /* Run the DSP image */
            status = DSPProcessor_Start(copyTask->hProcessor) ;

            if (DSP_SUCCEEDED(status))
            {
              fprintf(stdout, "DSPProcessor_Start succeeded.\n") ;
            }
            else
            {
                fprintf(stdout, "DSPProcessor_Start failed. Status = 0x%x\n",
                    status);
            }
        }

        return (status);
    }
```

2. **InitializeNode** – Allocate memory for a DSP node through **DSPNode_Allocate**. Then create and run the DSP node on the attached DSP through **DSPNode_Create** and **DSPNode_Run**. In order to appropriately set up the input and output streams in this example, the stream attributes must be specified, which are then used when connecting two nodes through **DSPNode_Connect**. In the following function, the stream may be setup as a **Processor Copy**, **Zero Copy**, or **DMA** stream by assigning properties to the DSP_STRMATTR structure (See the **DSPNode_Connect** documentation for which copy modes are supported by the different GPP OS's). These different types of streams represent different ways of transporting the data given to a stream. The code for each type of transport mode is shown in the inline table below. It indicates the corresponding stream transport mode ID and memory segment ID that must be used. *Note: for Zero Copy, the memory segment ID must be specified as shared memory, indicated by **DSP_SHMSEG0**.*

```
    /*
     *  ======== InitializeNode ========
     */
    static DSP_STATUS InitializeNode(STRMCOPY_TASK *copyTask)
    {
        BYTE            argsBuf[ARGSIZE + sizeof(ULONG)];
```

```
        DSP_CBDATA     *pArgs;
        DSP_NODEATTRIN  nodeAttrIn;
        DSP_STRMATTR    attrs;
        DSP_UUID        uuid;
        DSP_STATUS      status = DSP_SOK;

        // Assign node unique ID from global variable
        uuid = nodeuuid;
```

Set up the DSP_STRMATTR strm attribute structure according to ONE of the following modes:
Processor Copy (STRMMODE_PROCCOPY)

attrs.uBufsize = 1024;
attrs.uNumBufs = 1;
attrs.uAlignment = 0;
attrs.uTimeout = DSP_FOREVER;
attrs.lMode = STRMMODE_PROCCOPY;
attrs.uSegid= 0;

```
           // Setup node attributes
        nodeAttrIn.uTimeout = 10000;
        nodeAttrIn.iPriority = 5;

        pArgs = (DSP_CBDATA *)argsBuf;
        pArgs->cbData = ARGSIZE;

        /* Allocate the strmcopy node. */
        if (DSP_SUCCEEDED(status)) {
            status = DSPNode_Allocate(copyTask->hProcessor, &uuid, pArgs,
                    &nodeAttrIn, &copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Allocate succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Allocate failed. Status = 0x%x\n",
                    status);
            }
        }

        /* Connect the strmcopy node to the host. */
        if (DSP_SUCCEEDED(status)) {
            status = DSPNode_Connect((DSP_HNODE)DSP_HGPPNODE, 0, copyTask-
    >hNode,
                    0, &attrs);
            if (DSP_FAILED(status)) {
                fprintf(stdout, "DSPNode_Connect failed. Status = 0x%x\n",
                        status);
            }
        }

        /* Connect the host to the strmcopy node */
        if (DSP_SUCCEEDED(status)) {
```

```
        status = DSPNode_Connect(copyTask->hNode, 0,
(DSP_HNODE)DSP_HGPPNODE,
            0, &attrs);
        if (DSP_FAILED(status)) {
            fprintf(stdout, "DSPNode_Connect failed. Status = 0x%x\n",
                    status);
        }
    }

    /* Create the strmcopy node on the DSP. */
    if (DSP_SUCCEEDED(status)) {
        status = DSPNode_Create(copyTask->hNode);
        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPNode_Create succeeded.\n");
        }
        else {
            fprintf(stdout, "DSPNode_Create failed. Status = 0x%x\n",
                status);
        }
    }

    /* Start the strmcopy node on the DSP. */
    if (DSP_SUCCEEDED(status)) {
        status = DSPNode_Run(copyTask->hNode);
        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPNode_Run succeeded.\n");
        }
        else {
            fprintf(stdout, "DSPNode_Run failed. Status = 0x%x\n",
                status);
        }
    }

    return (status);
}
```

3. **`InitializeStreams`** – Open data streams to/from the DSP task node through **`DSPStream_Open`**. Then allocate buffers on the newly opened streams through **`DSPStream_AllocateBuffers`**. The **strmcopy** example opens one *HostToDsp* stream and one *DspToHost* stream. A new set of stream attributes must be specified, not unlike those specified in the node initialization above, which can then used when opening a stream through **`DSPStream_Open`**. The stream may be setup as a **Processor Copy**, a **Zero Copy**, or a **DMA** stream by assigning properties to the `DSP_STRMATTRIN` structure. The stream transport mode must reflect the mode that was assigned during the node initialization phase above. The code for each type of transport mode is shown in the inline table below. *Note: for both the Zero Copy and DMA transport modes, the memory segment ID must be specified as shared memory, indicated by **DSP_SHMSEG0**. Note that this is slightly different than the attributes for DSPNode_Connect, as specified above during node initialization, because in that case, only the Zero Copy transport mode segment ID must be specified as Shared Memory.*

```
/*
 *  ======== InitializeStreams ========
 */
static DSP_STATUS InitializeStreams(STRMCOPY_TASK  *copyTask)
{
```

```
    DSP_STATUS          status = DSP_SOK;
    int                 i = 0;
    DSP_STREAMATTRIN    attrs;
```

Set up the **DSP_STREAMATTRIN** strm attribute structure according to **ONE** of the following modes:
Processor Copy (STRMMODE_PROCCOPY)

attrs.cbStruct = sizeof(DSP_STREAMATTRIN);
attrs.uTimeout = 5000;
attrs.uAlignment = 0;
attrs.uNumBufs = 1;
attrs.lMode = **STRMMODE_PROCCOPY;**
attrs.uSegment = **0;**

```
    /* Open an input data stream from the host to the strmcopy node. */
    if (DSP_SUCCEEDED(status)) {

        status = DSPStream_Open(copyTask->hNode, DSP_TONODE, 0,
            &attrs, &copyTask->hInStream);

        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPStream_Open of input stream
succeeded.\n");
        }
        else {
            fprintf(stdout,
                "DSPStream_Open of input stream failed. Status = 0x%x\n",
                status);
        }
    }

    /* Open an output stream from the strmcopy node to the host. */
    if (DSP_SUCCEEDED(status)) {
        status = DSPStream_Open(copyTask->hNode, DSP_FROMNODE, 0,
            &attrs, &copyTask->hOutStream);
        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPStream_Open of output stream
succeeded.\n");
        }
        else {
            fprintf(stdout,
                "DSPStream_Open of output stream failed. Status =
0x%x\n",
                status);
        }
    }

    /* Allocate and issue buffer to input data stream. */
    if (DSP_SUCCEEDED(status)) {
        copyTask->ppInBufs = (BYTE**)malloc(sizeof(BYTE*) *
DEFAULTNBUFS);

        status = DSPStream_AllocateBuffers(copyTask->hInStream,
            DEFAULTBUFSIZE, copyTask->ppInBufs, DEFAULTNBUFS);
```

```
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout,
                    "DSPStream_AllocateBuffers for input stream
    succeeded.\n");
            }
            else {
                fprintf(stdout,
                    "DSPStream_AllocateBuffers for input stream failed. "
                    "Status = 0x%x\n", status);
            }
        }

        /* Allocate and issue buffer to output data stream. */
        if (DSP_SUCCEEDED(status)) {
            copyTask->ppOutBufs = (BYTE**)malloc(sizeof(BYTE*) *
    DEFAULTNBUFS);

            status = DSPStream_AllocateBuffers(copyTask->hOutStream,
                DEFAULTBUFSIZE, copyTask->ppOutBufs, DEFAULTNBUFS);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout,
                    "DSPStream_AllocateBuffers for output stream
    succeeded.\n");
            }
            else {
                fprintf(stdout,
                    "DSPStream_AllocateBuffers for output stream failed. "
                    "Status = 0x%x\n", status);
            }
        }

        return (status);
    }
```

6.3.4.2 GPP Application Run

Data streaming between the GPP and the DSP takes place during the run stage of the GPP application. The streaming process is described below:

1. GPP application reads user-specified data from its input file into the buffer **outBuf**.
2. GPP application calls **`DSPStream_Issue`** to put the filled **outBuf** to the DSP input data stream.
3. GPP application calls **`DSPStream_Reclaim`** to get the emptied **outBuf** from the DSP input data stream. If this call succeeds, it implies **outBuf** has been successfully transferred from the GPP to the DSP. Otherwise, an error has occurred.
4. When the DSP task node receives **outBuf**, it immediately sends it back to the GPP on the DSP output data stream, making data available to the GPP.
5. GPP application calls **`DSPStream_Issue`** to put an empty buffer **inBuf** to the DSP output data stream.
6. GPP application calls **`DSPStream_Reclaim`** to reclaim the filled **inBuf** from the DSP output data stream. If this call succeeds, it implies **inBuf** has successfully received data transmitted from the DSP to the GPP. Otherwise, an error has occurred.

7. GPP application writes **inBuf** to its output file.

6.3.4.3 GPP Application Cleanup

At cleanup, the GPP application reverses the setup process through functions **`CleanupProcessor`**, **`CleanupNode`**, and **`CleanupStreams`**. These functions accomplish the following:

1. **`CleanupStreams`** – Idle opened streams through **`DSPStream_Idle`**; deallocate data buffers on existing data streams through **`DSPStream_FreeBuffers`**; then delete opened data streams through **`DSPStream_Delete`**.

```
/*
 *  ======== CleanupStreams ========
 */
static DSP_STATUS CleanupStreams(STRMCOPY_TASK *copyTask)
{
    DSP_STREAMINFO  streamInfo;
    DSP_STATUS      status = DSP_SOK;
    DWORD           dwArg = 0;
    DWORD           dwBufsize = DEFAULTBUFSIZE;
    BYTE            *pBuf;
    int             i = 0;


    /* Reclaim and free stream buffers. */
    if (copyTask->hInStream) {
        DSPStream_Idle(copyTask->hInStream, TRUE);
        status = DSPStream_GetInfo(copyTask->hInStream, &streamInfo,
            sizeof(streamInfo));

        if (copyTask->ppInBufs) {
            /* Reclaim the buffer if it's still in the stream */
            if (streamInfo.uNumberBufsInStream) {
                status = DSPStream_Reclaim(copyTask->hInStream, &pBuf, \
                        &dwBufsize, NULL, &dwArg);
            }
            status = DSPStream_FreeBuffers(copyTask->hInStream,
                    copyTask->ppInBufs, DEFAULTNBUFS);
            if (DSP_FAILED(status)) {
                fprintf(stdout, "DSPStream_FreeBuffer of input buffer
failed. "
                    "Status = 0x%x\n", status);
            }

            copyTask->ppInBufs = NULL;
            /* Close open streams. */
            status = DSPStream_Close(copyTask->hInStream);
            if (DSP_FAILED(status)) {
                fprintf(stdout, "DSPStream_Close of input stream failed.
"
                    "Status = 0x%x\n", status);
            }

            copyTask->hInStream = NULL;
        }
    }
```

```
        if (copyTask->hOutStream) {
            DSPStream_Idle(copyTask->hOutStream, TRUE);

            /* Need to determine the number of buffers left in the stream */
            status = DSPStream_GetInfo(copyTask->hOutStream, &streamInfo,
                sizeof(streamInfo));

            if (copyTask->ppOutBufs) {
                if (streamInfo.uNumberBufsInStream) {
                    status = DSPStream_Reclaim(copyTask->hOutStream, &pBuf, \
                            &dwBufsize, NULL, &dwArg);
                }

                /* Free buffer. */
                status = DSPStream_FreeBuffers(copyTask->hOutStream,
                    copyTask->ppOutBufs, DEFAULTNBUFS);
                if (DSP_FAILED(status)) {
                    fprintf(stdout,
                        "DSPStream_FreeBuffer of output buffer failed. "
                        "Status = 0x%x\n", status);
                }
                copyTask->ppOutBufs = NULL;
            }

            status = DSPStream_Close(copyTask->hOutStream);
            if (DSP_FAILED(status)) {
                fprintf(stdout, "DSPStream_Close of output stream failed. "
                    "Status = 0x%x\n", status);
            }

            copyTask->hOutStream = NULL;
        }

        return (status);
    }
```

2. **`CleanupNode`** – Terminate a running node through **`DSPNode_Terminate`**; deallocate a terminated node through **`DSPNode_Delete`**.

```
    /*
     *  ======== CleanupNode ========
     */
    static DSP_STATUS CleanupNode(STRMCOPY_TASK *copyTask)
    {
        DSP_STATUS      status;
        DSP_STATUS      exitStatus;

        if (copyTask->hNode) {

            // Terminate DSP node.
            status = DSPNode_Terminate(copyTask->hNode, &exitStatus);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Terminate succeeded.\n", status);
            }
            else {
                fprintf(stdout, "DSPNode_Terminate failed: 0x%x\n", status);
            }
```

```
            /* Delete DSP node. */
            status = DSPNode_Delete(copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Delete succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Delete failed: 0x%x\n", status);
            }

            copyTask->hNode = NULL;
        }

        return (status);
    }
```

3. **`CleanupProcessor`** – Detach from an attached DSP through **`DSPProcessor_Detach`**. This ensures the GPP application no longer references a valid DSP.

```
    /*
     *  ======== CleanupProcessor ========
     */
    static DSP_STATUS CleanupProcessor(STRMCOPY_TASK *copyTask)
    {
        DSP_STATUS      status;

        if (copyTask->hProcessor) {
            /* Detach from processor. */
            status = DSPProcessor_Detach(copyTask->hProcessor);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPProcessor_Detach succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPProcessor_Detach failed.\n");
            }

            copyTask->hProcessor = NULL;
        }

        return (status);
    }
```

6.3.4.4 Notes on Buffer Size

Note that in this example there are two GPP bytes per DSP MAU, so in the GPP-side code buffers are allocated as an even number of GPP bytes, to ensure the DSP can properly process the data.

The maximum communication buffer size is sometimes limited by the amount of physical shared memory between the GPP and DSP processors. Please refer to platform-specific release notes for information on maximum buffer size.

If a client application buffer exceeds the maximum size, the extra buffer bytes will not be transferred across the GPP/DSP boundary.

6.3.5 Running the Strmcopy Application

While the code example above indicates what the developer would ordinarily include to setup the stream transport modes appropriately, the GPP console application **strmcopy.out** will automatically set up the

stream attributes based on user input. Therefore, the developer does not need to edit the source to test the different transport modes.

The GPP console application **strmcopy.out** takes the following arguments:

<transport-id>	The stream transport ID used to set up the stream: 0 = Processor Copy 1 = DSP-DMA Transport 2 = Zero Copy Transport
<input file>	File containing input data.
<output_file>	File to store output data.

The console application will produce output similar to the following, and will wait for the user to press 'Enter' before exiting.

```
DSPProcessor_Attach succeeded.
DSPNode_Allocate succeeded.
DSPNode_Create succeeded.
DSPNode_Run succeeded.
DSPStream_Open of input stream succeeded.
DSPStream_Open of output stream succeeded.
DSPStream_AllocateBuffers for input stream succeeded.
DSPStream_AllocateBuffers for output stream succeeded.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 512 bytes to DSP.
Writing 512 bytes to output file.
Sending 46 bytes to DSP.
Writing 46 bytes to output file.
Sending 0 bytes to DSP.
RunTask succeeded.
DSPStream_FreeBuffers of input buffer succeeded.
DSPStream_FreeBuffers of output buffer succeeded.
DSPStream_Close of input stream succeeded.
DSPStream_Close of output stream succeeded.
DSPNode_Terminate succeeded.
DSPNode_Delete succeeded.
DSPProcessor_Detach succeeded.
Press 'Enter' to exit program:
```

6.3.6 Summary

The **strmcopy** console application and DSP task node in this chapter demonstrates the use of data streams in DSP/BIOS Bridge. It describes the GPP and API calls needed to make data exchange between the GPP and DSP possible.

In the next example, **scale**, the messaging features of **ping** and data streaming features of **strmcopy** are used together, in an XDAIS socket node example.

6.4 The XDAIS Socket Application

The **scale** program provides an example of how an off-the-shelf XDAIS algorithm can be integrated into the DSP/BIOS Bridge task node framework. In particular, this example demonstrates:

- How a GPP program can create and control an XDAIS socket node on a DSP.
- How the socket enables scheduling of the XDAIS algorithm, and how it manages I/O for the algorithm.
- How to send command messages to the XDAIS socket to affect state changes within the XDAIS algorithm.
- How to exchange data buffers with the algorithm, via the XDAIS socket.

This sample GPP program will create an XDAIS socket node on the DSP, exchange control messages with the socket, send input data buffers to the socket, and receive scaled output buffers back from the socket for display by the GPP console.

This example builds upon the task control, messaging, and data streaming concepts developed in the previous examples, and focuses on how an XDAIS-compliant algorithm can be easily integrated into the DSP/BIOS Bridge framework.

The GPP files for this example reside under the **\samples\scale\linux** directory. The corresponding DSP side sources can be found in DSP sources **ti\dspbridge\dsp\samples** directory.

6.4.1 DSP/BIOS Bridge APIs Demonstrated

The following DSP/BIOS Bridge APIs are used in this example:

6.4.1.1 DSP APIs

- **NODE_getMsg**
- **NODE_putMsg**
- **NODE_wait**
- **STRM_allocateBuffer**
- **STRM_freeBuffer**
- **STRM_idle**
- **STRM_issue**
- **STRM_reclaim**
- **STRM_delete**

6.4.1.2 GPP APIs

- **DSPProcessor_Attach**
- **DSPProcessor_Detach**
- **DSPNode_Allocate**
- **DSPNode_Connect**
- **DSPNode_ConnectEx**
- **DSPNode_Create**
- **DSPNode_Run**
- **DSPNode_PutMessage**
- **DSPNode_GetMessage**
- **DSPNode_Terminate**
- **DSPNode_Delete**
- **DSPStream_Open**
- **DSPStream_AllocateBuffer**
- **DSPStream_FreeBuffer**
- **DSPStream_Issue**

- **DSPStream_Reclaim**
- **DSPStream_Idle**
- **DSPStream_Close**

6.4.2 XDAIS Socket Overview

An XDAIS socket node allows an XDAIS-compliant algorithm to "plug-in" to a DSP/BIOS Bridge application. The socket node provides a housing for the algorithm, calling algorithm functions as appropriate, and passing data to/from the algorithm as parameters in the algorithm function calls. This section reviews the XDAIS socket concept; for more detailed information see the chapter *DSP/BIOS Bridge Nodes*, and for detailed information on XDAIS consult *The TMS320 DSP Algorithm Standard Rules and Guidelines* and *The TMS320 DSP Algorithm Standard API Reference*.

XDAIS algorithms can be viewed as data transducers or transformers, and they are restricted from making direct I/O or OS calls. To use these transducers in a DSP/BIOS Bridge environment, a "socket" or housing must be created for the XDAIS algorithm to manage I/O between the algorithm and the rest of the DSP/BIOS Bridge application and to allow scheduling within the DSP OS. The following figure illustrates the concept.

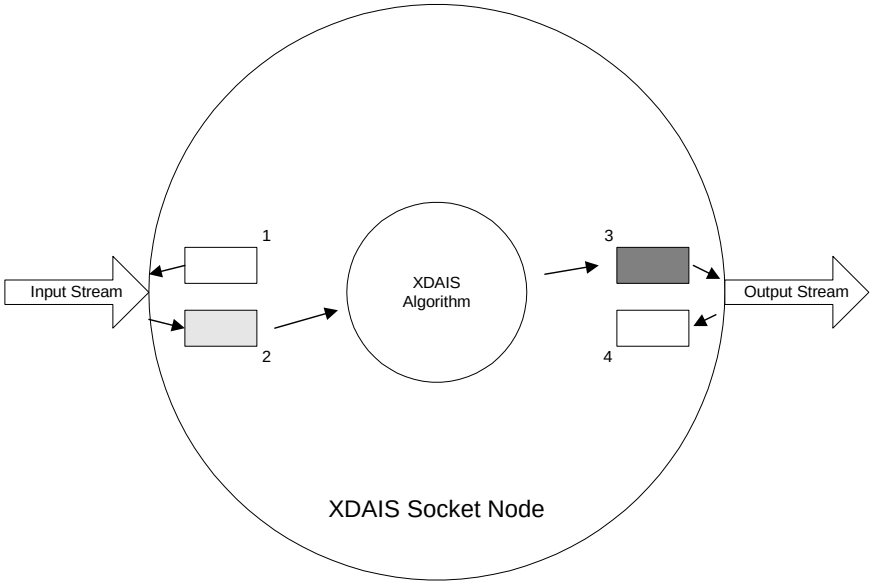

Figure 25 XDAIS Socket Concept

In this example the XDAIS Socket will:

- submit an empty buffer (1) to an input stream,
- retrieve a filled data buffer (2) from the input stream,
- pass the buffer to the algorithm's processing function,
- send the resultant data buffer (3) to an output stream, and
- retrieve an empty buffer (4) back from the output stream.

The *socket* will exist as a task in DSP/BIOS, allowing it to be scheduled along with all other DSP/BIOS Bridge nodes.

6.4.3 Program Design

scale demonstrates how a GPP application can create an XDAIS socket node on a DSP, send data buffers to the algorithm for scaling, and then receive the scaled buffers back for display. It also shows how algorithm control messages can be sent to the socket, which in turn translates the commands into algorithm control operations.

The application architecture is shown in the following figure:

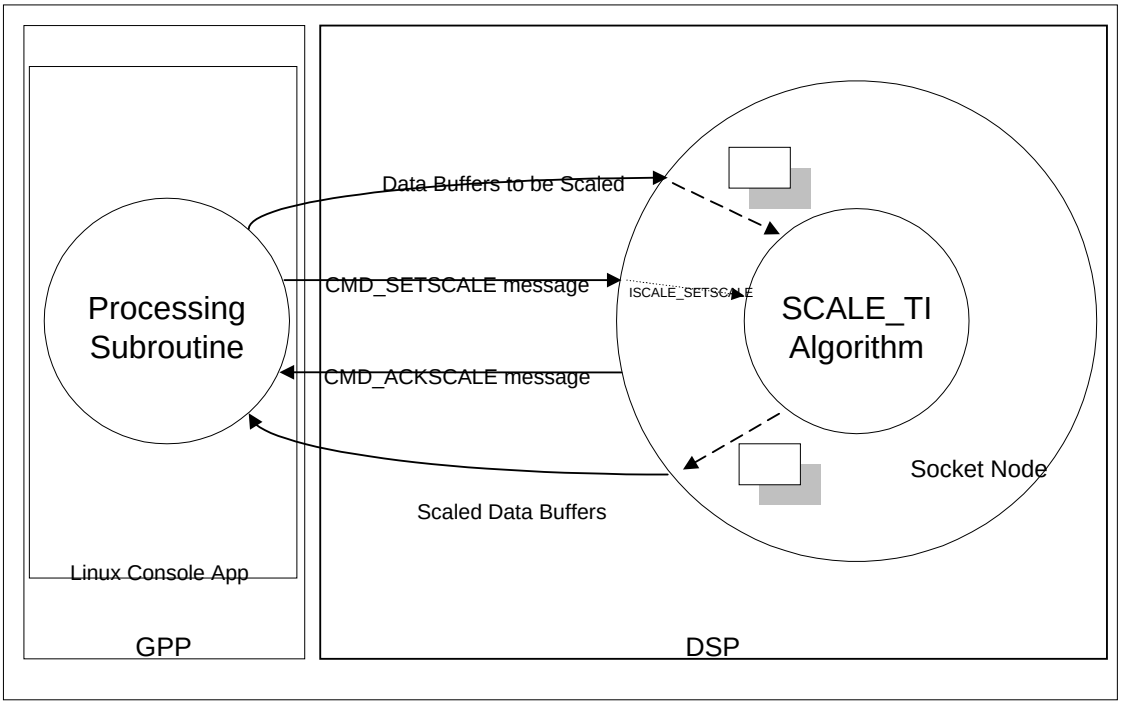

Figure 26 scale Application Architecture

The GPP program is constructed as a console application, which executes as a separate Linux process. The application has a single thread; subroutines are used to initialize communication with the DSP, allocate and create the socket node on the DSP, setup streaming, and finally initiate processing. The processing subroutine consists of a loop that executes a default number of iterations or a number of iterations as specified on the **scale** command line. In the processing loop a message will be sent to the socket indicating a change to the scale factor used by the algorithm. The GPP program will make a blocking call (**DSPNode_GetMessage**) to wait for an acknowledgement message from the socket that the algorithm scale factor was updated. Next a buffer will be issued to the stream to the socket, and the GPP program will call **DSPStream_Reclaim** to block until the buffer is returned by the DSP. The contents of the received buffer will be displayed on the console, and the next iteration of the loop will be started.

The DSP code consists of the XDAIS algorithm and three functions corresponding the create, execute, and delete phases of the socket. In its create-phase function, the socket allocates a node context structure, opens data streams, allocates data buffers, queries the XDAIS algorithm for its memory requirements, allocates memory for the algorithm, and initializes and activates the algorithm. In its execute-phase, function the socket runs a loop that does a **NODE_wait** blocking call to wait for either a data buffer or a message to arrive from the GPP. If a data buffer has arrived it will be passed to the algorithm for scaling, and then sent back to the GPP application. If a CMD_SETSCALE message has arrived, the socket will call the algorithm's control interface to pass the new scale factor to the algorithm, and then it will acknowledge the command to the GPP; if an RMS_EXIT message has arrived the socket will break out of its processing loop, idle streams, reclaim any outstanding buffers, and then exit. In its delete phase, the socket will deactivate the XDAIS algorithm, free the stream buffers that were allocated in the create phase, close the data streams, free the memory allocated for the algorithm, delete the algorithm instance object, and finally free the socket node's context structure.

6.4.4 The XDAIS Algorithm

The focus of this example application is how to integrate an XDAIS algorithm into a socket node, and how to control and pass data to it within a DSP/BIOS Bridge framework. As such, only a rudimentary XDAIS algorithm is used, so as to not distract from the main focus. The SCALE_TI algorithm consists of a basic XDAIS algorithm skeleton, plus a processing function that takes an input buffer and multiplies it by a scale factor to produce an output buffer. Two algorithm commands are defined (ISCALE_SETSCALE and ISCALE_GETSCALE), to allow the GPP application to control the scale factor used by the algorithm. The source for the algorithm is shown below. The banner comments for each function summarize its usage. For more details on XDAIS algorithms consult *The TMS320 DSP Algorithm Standard Rules and Guidelines*, and *The TMS320 DSP Algorithm Standard API Reference*.

```
/*
 *  ======== scale_ti.c ========
 *  Implementation of the SCALE_TI.h interface; TI's implementation
 *  of the ISCALE interface.
 *
 */
#pragma CODE_SECTION(SCALE_TI_alloc, ".text:algAlloc")
#pragma CODE_SECTION(SCALE_TI_free, ".text:algFree")
#pragma CODE_SECTION(SCALE_TI_initObj, ".text:algInit")
#pragma CODE_SECTION(SCALE_TI_init, ".text:init")
#pragma CODE_SECTION(SCALE_TI_exit, ".text:exit")

#include <std.h>
#include "iscale.h"

#include "scale_ti.h"

/*
 *  ======== SCALE_TI_Obj ========
 */
typedef struct SCALE_TI_Obj {
```

```
    IALG_Obj    alg;            /* MUST be first field of all SCALE objs
*/
    Uns         scaleFactor;
} SCALE_TI_Obj;


/*
 *  ======== SCALE_TI_activate ========
 *  Activate our object; e.g., initialize any scratch memory required
 *  by the SCALE_TI processing methods.
 */
Void SCALE_TI_activate(IALG_Handle handle)
{
}




/*
 *  ======== SCALE_TI_alloc ========
 *  Return a table of memory descriptors that describe the memory needed
 *  to construct a SCALE_TI_Obj structure.
 */
Int SCALE_TI_alloc(const IALG_Params *scaleParams, IALG_Fxns **fxns,
    IALG_MemRec memTab[])
{
    const ISCALE_Params *params = (Void *)scaleParams;

    if (params == NULL) {
        params = &ISCALE_PARAMS;        /* set default parameters */
    }

    /* Request memory for SCALE object */
    memTab[0].size = sizeof(SCALE_TI_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_EXTERNAL;
    memTab[0].attrs = IALG_PERSIST;

    return (1);
}

/*
 *  ======== SCALE_TI_control ========
 *  Execute a control operation.
 */
Void SCALE_TI_control(IALG_Handle handle, IALG_Cmd cmd, IALG_Status *
status)
{
    SCALE_TI_Obj *scale = (Void *)handle;

    if (cmd == ISCALE_SETSCALE) {
        scale->scaleFactor = (Uns) ((ISCALE_Status *)status)-
>scaleFactor;
    }
    else if (cmd == ISCALE_GETSCALE) {
        ((ISCALE_Status *)status)->scaleFactor = scale->scaleFactor;
    }
}
```

```
/*
 *  ======== SCALE_TI_deactivate ========
 *  Deactivate our object; e.g., save any scratch memory requred
 *  by the SCALE_TI processing methods to persistent memory.
 */
Void SCALE_TI_deactivate(IALG_Handle handle)
{
}

/*
 *  ======== SCALE_TI_exit ========
 *  Exit the SCALE_TI module as a whole.
 */
Void SCALE_TI_exit(Void)
{
}

/*
 *  ======== SCALE_TI_free ========
 *  Return a table of memory pointers that should be freed.  Note
 *  that this should include *all* memory requested in the
 *  SCALE_TI_alloc operation above.
 */
Int SCALE_TI_free(IALG_Handle handle, IALG_MemRec memTab[])
{
    SCALE_TI_Obj *scale = (Void *)handle;
    Int n;

    n = SCALE_TI_alloc(NULL, NULL, memTab);

    memTab[0].base = scale;

    return (n);
}

/*
 *  ======== SCALE_TI_init ========
 *  Initialize the SCALE_TI module as a whole.
 */
Void SCALE_TI_init(Void)
{
}

/*  ======== SCALE_TI_initObj ========
 *  Initialize the memory allocated for our instance.
 */
Int SCALE_TI_initObj(IALG_Handle handle, const IALG_MemRec memTab[],
    IALG_Handle p, const IALG_Params *scaleParams)
{
    SCALE_TI_Obj *scale = (Void *)handle;
    const ISCALE_Params *params = (Void *)scaleParams;

    if (params == NULL) {
        params = &ISCALE_PARAMS;        /* if NULL params, use defaults
*/
    }
```

```
        scale->scaleFactor = params->scaleFactor;

        return (IALG_EOK);
    }

    /*  ======== SCALE_TI_moved ========
     *  Fixup any pointers to data that has been moved by the client.
     */
    Void SCALE_TI_moved(IALG_Handle handle, const IALG_MemRec memTab[],
        IALG_Handle p, const IALG_Params *scaleParams)
    {
        /* NOT USED */
    }

    /*
     *  ======== SCALE_TI_scale ========
     *  TI's implementation of the scale operation.
     */
    Void  SCALE_TI_scale(ISCALE_Handle handle, Uns in[], Uns out[], Uns size)
    {
        SCALE_TI_Obj *scale = (Void *)handle;
        Int i;

        for (i = 0; i < size; i++) {
            out[i] = in[i] * scale->scaleFactor;
        }
    }
```

6.4.5 Implementing the Socket Node

An XDAIS socket node is a special type of DSP/BIOS Bridge task node. It exists as an independent execution thread in the DSP OS, and it also houses an XDAIS algorithm. Like the previous Ping and Strmcopy examples, three C-callable functions will be needed for the node, corresponding to the create, execute, and delete phases of its life cycle.

A socket context is created and initialized during the socket create phase, and a pointer to this context is stored within the node's environment, so that it can be accessed in the execute and delete phases. Socket context information for this example includes the XDAIS algorithm instance handle, input and output stream handles, input and output buffer pointers, and input and output buffer sizes:

```
    /*
     *  ======== SSKT_Obj ========
     *  Scale socket node context.
     */
    typedef struct SSKT_Obj {
       IALG_Handle algHandle;
       STRM_Handle inStream;
       STRM_Handle outStream;
       Uns * outBuf;
       Uns * inBuf;
       Uns outSize;
       Uns inSize;
    } SSKT_Obj;
```

6.4.5.1 Create Phase

The create-phase function for the socket is shown below. The function signature for the socket create function follows a standard format for all XDAIS socket nodes. This signature is similar to that used for 'normal' DSP/BIOS Bridge task nodes, with one additional parameter: a pointer to the IALG_Fxn table of the XDAIS algorithm.

The first thing the socket create function does is to allocate a socket context object (SSKT_Obj). For the socket's input and output streams the create function calls an XDAIS socket support function named **DSKT_createStream**, which parses a stream definition structure passed from the GPP and opens the stream. Both input and output stream handles are stored in the socket context object. The socket then allocates a data buffer for each of the streams and stores the buffer pointers in the socket context object. The create function is now ready to instantiate the algorithm, so it calls a support function named **DSKT_createAlgObject**. This function will query the XDAIS algorithm for its memory requirements, allocate memory for the algorithm, and call the algorithm's initialization function. Once the algorithm instance has been created the socket create function activates the algorithm by calling its activate function. Finally, the socket context is attached to the node environment, and the create function exits.

```
/*
 *  ======== SCALESOCKET_TI_create ========
 */
RMS_STATUS SCALESOCKET_TI_create(Int argLength, Char * argData,
    IALG_Fxns *algFxns, Int numInStreams, RMS_WORD inDef[], Int
numOutStreams,
    RMS_WORD outDef[], NODE_EnvPtr node)
{
    struct SSKT_Obj *dataPtr;
    STRM_Attrs attrs = STRM_ATTRS;

    if ((dataPtr = MEM_calloc(0, sizeof(SSKT_Obj), 0)) != MEM_ILLEGAL) {

        /* parse stream definition params, create input stream */
        dataPtr->inStream = DSKT_createStream((RMS_StrmDef *)inDef[0],
            STRM_INPUT, &dataPtr->inSize);

        /* parse stream definition params, create output stream */
        dataPtr->outStream = DSKT_createStream((RMS_StrmDef *)outDef[0],
            STRM_OUTPUT, &dataPtr->outSize);

        /* check for stream creation failure */
        if((dataPtr->inStream == NULL) || (dataPtr->outStream == NULL)) {
            return(RMS_ESTREAM);
        }

        /* allocate data buffers */
        dataPtr->inBuf = STRM_allocateBuffer(dataPtr->inStream,
            dataPtr->inSize);
        dataPtr->outBuf = STRM_allocateBuffer(dataPtr->outStream,
            dataPtr->outSize);

        /* check for buffer allocation failure */
        if((dataPtr->inBuf == NULL) || (dataPtr->outBuf == NULL)) {
            return(RMS_EOUTOFMEMORY);
        }

        /* create an algorithm instance object */
```

```
            dataPtr->algHandle = DSKT_createAlgObject(algFxns, (IALG_Params
    *)NULL);

            /* check for algorithm instance creation failure */
            if (dataPtr->algHandle == NULL) {
                return(RMS_EOUTOFMEMORY);
            }

            /* activate the algorithm */
            dataPtr->algHandle->fxns->algActivate(dataPtr->algHandle);

            /* attach socket's context structure to the node environment */
            node->moreEnv = dataPtr;

            return(RMS_EOK);
        }
        else {
            return(RMS_EOUTOFMEMORY);
        }
    }
```

6.4.5.2 Execute Phase

The execute phase of the socket will run as an independent execution thread of the DSP OS. It is in this execute-phase function (see below) that the socket will exchange data buffers with the GPP application and accept and respond to commands (sent as messages) from the GPP application. It is this function that will call XDAIS algorithm processing functions as appropriate. This function will also watch for an exit command from the Resource Manager on the GPP (sent as a message containing a pre-defined command `RMS_EXIT`). When the exit command is received, the function will break out of the continuous loop, idle the socket's streams, reclaim any outstanding buffers, and then exit.

The first thing the execute function does is to de-reference the node environment pointer to get a pointer to the socket context object initialized in the create phase.

The execute function uses the blocking API call **`NODE_wait`** to efficiently wait for either a message to arrive from the GPP, or for a data buffer to be ready on the socket's input stream. If a `CMD_SETSCALE` message has arrived, the algorithm's algControl function will be called, with the command set to `ISCALE_SETSCALE`, and the new scale factor specified in an IALG_Status structure. After performing the control operation the socket will acknowledge the command back to the GPP, using **`DSPNode_PutMessage`**.

If **`NODE_wait`** indicates that the input stream is ready, the execute function will reclaim data buffers from both the input and output streams, and call the algorithm's SCALE_TI_scale function. The 'empty' input buffer will then be issued back to the input stream, and the 'filled' output buffer will be issued to the output stream.

```
/*
 *  ======== SCALESOCKET_TI_execute ========
 */
RMS_STATUS SCALESOCKET_TI_execute(NODE_EnvPtr node)
{
    struct SSKT_Obj *dataPtr = node->moreEnv;
    Bool firstFrame = TRUE;
    ISCALE_Status status;
    Uns readyStreams;
    RMS_DSPMSG msg;
    Uns msgsReady;
    Arg arg;

    /* prime the input stream with an empty buffer */
```

```
    STRM_issue(dataPtr->inStream, dataPtr->inBuf, dataPtr->inSize,
dataPtr->inSize, 0);

    while(1) {
        /* wait until input stream is ready, or until GPP message arrives
*/
        readyStreams = NODE_wait(node, &dataPtr->inStream, 1,
NODE_FOREVER,
            &msgsReady);

        /* if a message has arrived: */
        if (msgsReady != 0) {

            /* get the message */
            NODE_getMsg(node, &msg, 0);

            /* if got CMD_SETSCALE, invoke alg's algControl function */
            if(msg.cmd == CMD_SETSCALE) {
                status.scaleFactor = (Int) msg.arg1;

                /* call algControl to change the alg scale factor */
                dataPtr->algHandle->fxns->algControl(dataPtr->algHandle,
                    (IALG_Cmd) ISCALE_SETSCALE, (IALG_Status *)&status);

                 msg.cmd = CMD_ACKSCALE;
                NODE_putMsg(node, NODE_TOGPP, &msg, 1000);
            }

            /* else if got RMS_EXIT then exit the execute-phase */
            else if(msg.cmd == RMS_EXIT) {
                break;
            }
        }

        /* if a buffer is ready on the input stream: */
        if (readyStreams != 0) {

            /* get the buffer from the input stream */
            STRM_reclaim(dataPtr->inStream, (Ptr *) &dataPtr->inBuf,
NULL, &arg);

            /* if this is the first iteration use the output buffer
directly */
            if (firstFrame == TRUE) {
                firstFrame = FALSE;
            }
            /* else reclaim empty buffer from output stream */
            else {
                STRM_reclaim(dataPtr->outStream, (Ptr *)&dataPtr->outBuf,
NULL, &arg);
            }

            /* call the alg's scale function to transform the input
buffer */
            ((ISCALE_Handle)(dataPtr->algHandle))->fxns->scale(
                (ISCALE_Handle)dataPtr->algHandle, dataPtr->inBuf,
                dataPtr->outBuf, dataPtr->inSize);
```

```
            /* send the output buffer to the output stream */
            STRM_issue(dataPtr->outStream, dataPtr->outBuf, dataPtr-
>inSize,
                dataPtr->inSize, 0);

            /* send the empty input buffer back to the input stream */
            STRM_issue(dataPtr->inStream, dataPtr->inBuf, dataPtr-
>inSize,
                dataPtr->inSize, 0);
        }
    }

    /* idle both streams */
    STRM_idle(dataPtr->inStream, TRUE);
    STRM_idle(dataPtr->outStream, TRUE);

    /* reclaim any buffers remaining in the streams */
    STRM_reclaim(dataPtr->inStream, (Ptr *) &dataPtr->inBuf, NULL, &arg);
    if (firstFrame == FALSE) {
        STRM_reclaim(dataPtr->outStream, (Ptr *) &dataPtr->outBuf, NULL,
&arg);
    }

    /* done */
    return(RMS_EOK);
}
```

6.4.5.3 Delete Phase

The delete phase of the socket is responsible for deallocating all objects allocated during the create phase.

The socket delete function first accesses it's node context through the `node->moreEnv` pointer. It then deactivates the algorithm, frees stream data buffers, and deletes the streams. A socket support function **`DSKT_algFreeObject`** is called to free memory allocated for the algorithm and to free the algorithm instance object. Finally, the socket context object is freed, and the function exits.

```
/*
 *  ======== SCALESOCKET_TI_delete ========
 */
RMS_STATUS SCALESOCKET_TI_delete(NODE_EnvPtr node)
{
    struct SSKT_Obj *dataPtr = node->moreEnv;

    if ((node != NULL) && (dataPtr != NULL)) {

        /* deactivate the algorithm */
        if (dataPtr->algHandle != NULL) {
            dataPtr->algHandle->fxns->algDeactivate(dataPtr->algHandle);
        }

        /* delete stream data buffers */
        if (dataPtr->inBuf != NULL) {
            STRM_freeBuffer(dataPtr->inStream, dataPtr->inBuf, dataPtr-
>inSize);
        }
        if (dataPtr->outBuf != NULL) {
            STRM_freeBuffer(dataPtr->outStream, dataPtr->outBuf,
```

```
            dataPtr->outSize);
        }

        /* delete streams */
        if (dataPtr->inStream != NULL) {
            STRM_delete(dataPtr->inStream);
        }
        if (dataPtr->outStream != NULL) {
            STRM_delete(dataPtr->outStream);
        }

        /* free the algorithm instance object */
        DSKT_freeAlgObject(dataPtr->algHandle);

        /* free the socket node's context structure */
        MEM_free(0, dataPtr, sizeof(SSKT_Obj));
    }
    return(RMS_EOK);
}
```

Remember that DSP/BIOS Bridge is a *dynamic task* environment, and DSP tasks will be coming into and going out of existence while the device is in operation. To prevent memory leaks, ***it is imperative that all dynamically created objects are finally destroyed***.

6.4.5.4 Configuring the Socket Node

The socket node's configuration data must be entered via the DSP/BIOS Configuration tool. Note that all DCD information resides in special sections in the DSP base image executable file, and is accessed by the DSP/BIOS Bridge Resource Manager when booting the DSP.

Configuration parameters for the socket node include:

Table 17 Socket Node Parameters

Node name	SCALESOCKET_TI
Node type	DAISNODE
DSP_UUID	D91A01BD_D215_11D4_8626_00105A98CA0B
Runtime priority	8
Stack size (words)	1024
System stack size (words)	256
Stack segment	0
Maximum messages queued to node	4
# of node input streams	1
# of node output streams	1
Timeout value of GPP blocking calls (in milliseconds)	2000
Load type of node	**Static**
Create Phase function name	**SCALESOCKET_TI_create**
Execute Phase function name	**SCALESOCKET_TI_execute**
Delete Phase function name	**SCALESOCKET_TI_delete**

Name of structure containing iAlg interface	**SCALE_TI_ISCALE**

Refer to the chapter *DSP/BIOS Bridge Configuration Database* for more details.

6.4.6 Implementing the GPP Program

The GPP program is named **scale.out**. For simplicity, it is written as a console application and provides a minimal user interface. The source for the program resides in the file **scale.c** in **\dspbridge\samples** directory.

The example is organized into a set of subroutines, with each routine calling one or more DSP/BIOS Bridge APIs to accomplish some action with the DSP. The main functions are as follows:

InitializeProcessor	Attaches the Linux application to the DSP (**DSPProcessor_Attach**).
InitializeNode	Allocates data structures for the socket node (**DSPNode_Allocate**), defines stream connections to the GPP application (**DSPNode_Connect**), creates the node on the DSP (**DSPNode_Create**), and starts the node running on the DSP (**DSPNode_Run**).
InitializeStreams	Opens data streams to and from the socket on the DSP (**DSPStream_Open**), and allocates data buffers for the stream (**DSPStream_AllocateBuffers**).
RunTask	The main processing loop for **scale**: sends a message to the socket defining a new algorithm scale factor (**DSPNode_PutMessage**), and waits for an acknowledgement (**DSPNode_GetMessage**); issues a buffer to the stream to the DSP (**DSPStream_Issue**), and reclaims a scaled buffer back from the DSP (**DSPStream_Reclaim**); displays the contents of the scaled buffer, and loops back for the next iteration.
CleanupStreams	Frees stream buffers (**DSPStream_FreeBuffers**), idles streams **DSPStream_Idle**), and closes streams (**DSPStream_Close**).
CleanupNode	Signals the end of the node's execute phase (**DSPNode_Terminate**), invokes the node's delete-phase function, and frees all DSP resources used by the node (**DSPNode_Delete**).
CleanupProcessor	Detaches the console application from the DSP (**DSPProcessor_Detach**).

A common structure is defined (SCALE_TASK) to pass intermediate context between the subroutines:

```
/* scale application context data structure: */
typedef struct {
    DSP_HPROCESSOR  hProcessor;                 /* processor handle */
    DSP_HNODE       hNode;                      /* node handle */
    DSP_HSTREAM     hInStream;                  /* input stream handle */
    DSP_HSTREAM     hOutStream;                 /* output stream handle
*/
    BYTE            *pInBuf;                    /* input stream buffer */
    BYTE            *pOutBuf;                   /* output stream buffer
*/
    UINT            uIterations;                /* number of loop
iterations */
} SCALE_TASK;
```

6.4.7 Running the Application

The GPP console application **scale.out** takes the following arguments:

<iterations> Number of iterations.

The console application will produce output similar to the following, and will wait for the user to press 'Enter' before exiting.

```
DSPProcessor_Attach succeeded.
DSPNode_Allocate succeeded.
DSPNode_Create succeeded.
DSPNode_Run succeeded.
DSPStream_Open of output stream succeeded.
DSPStream_Open of input stream succeeded.
DSPStream_AllocateBuffers for input stream succeeded.
DSPStream_AllocateBuffers for output stream succeeded.
Buffer to be scaled by DSP: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Setting alg scale factor to 0…DSP ACK.
Iteration 0 completed, received: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Setting alg scale factor to 1…DSP ACK.
Iteration 1 completed, received: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Setting alg scale factor to 2…DSP ACK.
Iteration 2 completed, received: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Setting alg scale factor to 3…DSP ACK.
Iteration 3 completed, received: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Setting alg scale factor to 4…DSP ACK.
Iteration 4 completed, received: 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
Setting alg scale factor to 5…DSP ACK.
Iteration 5 completed, received: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
Setting alg scale factor to 6…DSP ACK.
Iteration 6 completed, received: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
Setting alg scale factor to 7…DSP ACK.
Iteration 7 completed, received: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
RunTask succeeded.
DSPStream_FreeBuffers of input buffer succeeded.
DSPStream_Idle of input stream succeeded.
DSPStream_Close of input stream succeeded.
DSPStream_FreeBuffers of output buffer succeeded.
DSPStream_Idle of output stream succeeded.
DSPStream_Close of output stream succeeded.
DSPNode_Terminate succeeded.
DSPNode_Delete succeeded.
DSPProcessor_Detach succeeded.
Press 'Enter' to exit program:
```

6.4.8 Summary

The **scale** console application and corresponding DSP-side code demonstrate how an XDAIS-compliant algorithm can be integrated into a DSP/BIOS Bridge application, within a “socket node”. Data streaming is used to exchange buffers between the console application and the algorithm on the DSP, and messaging is used to send control commands to the algorithm and receive acknowledgements back from the DSP.

Integrating the XDAIS algorithm into the DSP/BIOS Bridge application requires that three functions be written, one each for the create, execute, and delete phases of the socket housing the algorithm. These three functions are very similar to those needed for a ‘normal’ DSP/BIOS Bridge task node, (and they enable an XDAIS algorithm to be used “off-the-shelf”, with no special modifications for DSP/BIOS Bridge).

6.5 The DMMCOPY Application

The **dmmcopy** example demonstrates basic usage of the Dynamic Memory Mapping (DMM) APIs. For specific usage information about Dynamic Memory Mapping, please refer to the *DSP/BIOS Bridge Application Note for Dynamic Memory Mapping*.

The **dmmcopy** example maps two buffers allocated on the ARM to two regions in the DSP virtual address space using the DMM APIs. It then reads data from an input file specified on the command line into the first buffer. Using messaging to synchronize buffer access with the DSP, the ARM sends a notification message to the DSP, which prompts the DSP to copy all of the contents in the first buffer into the second buffer. The DSP then sends a notification message to the ARM, which prompts the ARM to read the contents from the second buffer and write the data to the output file, which is also specified on the command line. This process is repeated until the entire input file has been copied.

The GPP files for this example reside under the **\dspbrige\samples\linux\dmmcopy** directory. The DSP files reside under the DSP sources **\ti\dspbridge\dsp\samples** directory.

6.5.1 DSP/BIOS Bridge APIs Demonstrated

The following DSP/BIOS Bridge APIs are used in the application:

6.5.1.1 DSP APIs

- **NODE_getMsg**
- **NODE_putMsg**

6.5.1.2 Linux APIs

- **DSPProcessor_Attach**
- **DSPNode_Allocate**
- **DSPNode_Connect**
- **DSPNode_ConnectEx**
- **DSPNode_Create**
- **DSPNode_Run**
- **DSPProcessor_ReserveMemory**
- **DSPProcessor_Map**
- **DSPNode_PutMessage**
- **DSPNode_GetMessage**
- **DSPProcessor_FlushMemory**
- **DSPProcessor_InvalidateMemory**
- **DSPProcessor_UnMap**
- **DSPProcessor_UnReserveMemory**
- **DSPNode_Terminate**
- **DSPNode_Delete**
- **DSPProcessor_Detach**

6.5.2 Dmmcopy Program Design

The **dmmcopy** example demonstrates how to properly use Dynamic Memory Mapping (DMM) to leverage the power of the DSP MMU and large buffers without having to rely on data streaming. On the GPP side, the application allocates two buffers and maps them each to a respective DSP virtual memory address space. These addresses are sent to the DSP. The GPP then reads in a fixed-length amount of data from the input file and writes this data into the first buffer. After the GPP has written data to the first buffer, the DSP is able to access the same buffer as well as the contents of the buffer. The DSP application then copies the data from one buffer into another buffer, and after it has completed, the GPP writes the data from the second buffer to an output file.

This example uses DSP/BIOS Bridge messaging to synchronize buffer access between the GPP and the DSP.

6.5.3 Implementing the Dmmcopy Task Node

Dmmcopy, like **ping**, uses a generic DSP/BIOS Bridge task node based on the three-phase task model to perform data streaming between the GPP and the DSP.

Since **dmmcopy** does not rely on streams to communicate, it does not need a DSP context object to keep stream information between phases.

6.5.3.1 Create Phase

The **dmmcopy** create phase, **DMMCOPY_TI_create**, essentially does nothing since there are no streams to create.

```
/*
 *  ======== DMMCOPY_TI_create ========
 */
RMS_STATUS DMMCOPY_TI_create(Int argLength,
                             Char * argData,
                             Int numInStreams,
                             RMS_WORD inDef[],
                             Int numOutStreams,
                             RMS_WORD outDef[],
                             NODE_EnvPtr env)5104408412
{
    LOG_printf(TRACE, "DMMCOPY_TI_create");

    /* No streams to setup! */

    return (RMS_EOK);
}
```

6.5.3.2 Execute Phase

In **dmmcopy**'s execute phase function, **`DMMCOPY_TI_execute`**, the following occurs:

1. DSP task calls NODE_getMsg to wait for control messages to before it can setup its DMM buffers or access them.
2. DSP receives a message from the GPP indicating that GPP buffers have been mapped. The developer determines that this type of message should identify itself by a special **DMM_SETUPBUFFERS** command. The message contains two arguments, each containing a DSP virtual address pointer that has been mapped to a GPP buffer. The DSP stores these pointers and then waits for another message. (**Note**: It is up to the developer to determine how to synchronize access to the buffers between the GPP and the DSP. For messaging, the developer may create meaningful commands to be sent in each message between the GPP and the DSP).
3. DSP receives a message, determined by the developer to be a special **DMM_WRITEREADY** command, from the GPP indicating that it may access the mapped buffers.
4. DSP retrieves the number of words to copy and then copies the contents of one buffer into the other buffer. The DSP then notifies the GPP that it has finished using the buffers and waits for another message.
5. The DSP task repeats steps 3 and 4 for each round of data indicated by the GPP.
6. DSP receives an **RMS_EXIT** message, which indicates that the GPP is terminating execution of the node. At this time, the DSP exits the execute phase.

```
/*
 *  ======== DMMCOPY_TI_execute ========
 */
```

```
RMS_STATUS DMMCOPY_TI_execute(NODE_EnvPtr env)
{
    Uns             size;
    RMS_DSPMSG      msg;
    Uns             *fromGPP;
    Uns             *toGPP;

    LOG_printf(TRACE, "DMMCOPY_TI_execute");

    LOG_printf(TRACE, "Waiting for first buffer from host...");

    /* Initialize msg structure. */
    msg.cmd = ~(RMS_EXIT);

    while (1) {

        /* Get message. */
        NODE_getMsg(env, &msg, NODE_FOREVER);

        if (msg.cmd == RMS_EXIT) {

            /* Exit if GPP says we should */
            LOG_printf(TRACE, "Got RMS_EXIT");
            break;

        } else if (msg.cmd == DMM_SETUPBUFFERS) {

            /* Setup buffer addresses */
            fromGPP = (Uns *)(msg.arg1);
            toGPP = (Uns *)(msg.arg2);
            LOG_printf(TRACE, "Buffer addresses stored");

        } else if (msg.cmd == DMM_WRITEREADY) {

            /* How many words to copy? */
            size = (Uns)(msg.arg1);

            /* Copy words from input buffer to output buffer */
            memcpy((Uns *)toGPP, (Uns *)fromGPP, size);

            LOG_printf(TRACE, "Copied %d words...", size);

            /* Tell GPP that we're done copying */
            NODE_putMsg(env, NODE_TOGPP, &msg, 0);
            LOG_printf(TRACE, "GPP notified of completion");
        }
    }

    return (RMS_EOK);
}
```

6.5.3.3 Delete Phase

Like the create phase, the **dmmcopy** delete phase, **DMMCOPY_TI_delete**, also does nothing, since there are no streams to tear down.

```
/*
 *  ======== DMMCOPY_TI_delete ========
```

```
 */
RMS_STATUS DMMCOPY_TI_delete(NODE_EnvPtr env)
{
    LOG_printf(TRACE, "DMMCOPY_TI_delete");

    /* No streams to tear down! */

    return (RMS_EOK);
}
```

6.5.3.4 Configuring the Dmmcopy Task node

The dmmcopy node's configuration data must be entered via the BIOS Configuration tool. Note that all DCD information resides in special sections within the DSP base image executable file, and is accessed by the DSP/BIOS Bridge Resource Manager on DSP boot-up.

DSP/BIOS Bridge DSP tasks are configured under the **DSP/BIOS Bridge Node Manager** (not the TSK Manager) in the BIOS configuration tool.

Configuration parameters for the **dmmcopy** node include:

Table 18 DMMcopy Node Parameters

Node name	DMMCOPY_TI
Node type	TASKNODE
DSP_UUID	28BA464F_9C3E_484E_990F_48305B183848
Runtime priority	3
Stack size (words)	512
System stack size (words)	256
Stack segment	0
Maximum messages queued to node	1
# of node input streams	0
# of node output streams	0
Timeout value of GPP blocking calls (in milliseconds)	1000
Load type of node	**Static**
Create Phase function name	**DMMCOPY_TI_create**
Execute Phase function name	**DMMCOPY_TI_execute**
Delete Phase function name	**DMMCOPY_TI_delete**

Refer to other samples and the section on *DSP/BIOS Bridge Configuration Database* for more details.

6.5.4 Implementing dmmcopy's GPP application

The GPP application for **dmmcopy** works hand-in-hand with the DSP task node, and has three abstract functions that correspond to the DSP task node's three phase model: initialize (create), run (execute), and cleanup (delete). The sections below describe the GPP application implementation in more detail.

6.5.4.1 GPP Application Initialization

The dmmcopy GPP application makes use of the following context structure:

```
typedef struct {
```

```
        DSP_HPROCESSOR      hProcessor;     /* Handle to processor. */
        DSP_HNODE           hNode;          /* Handle to node. */
    } DMMCOPY_TASK;
```

During initialization, the GPP console application uses functions **InitializeProcessor**, and **InitializeNode** to perform various DSP/BIOS Bridge related setup:

1) **InitializeProcessor** – Attach to a valid DSP through **DSPProcessor_Attach**. Then load and start a DSP image, if necessary.

```
/*
 *  ======== InitializeProcessor ========
 */
static DSP_STATUS InitializeProcessor(DMMCOPY_TASK *copyTask)
{
    DSP_STATUS          status = DSP_SOK;
    TCHAR               tszEventName[] = TEXT("testevent1");
    HANDLE              hThread = NULL;

    /* Attach to an available DSP (in this case, the 1st DSP). */
    if (DSP_SUCCEEDED(status)) {
        status = DSPProcessor_Attach(DEFAULTDSPUNIT, NULL,
            &copyTask->hProcessor);
        if (DSP_SUCCEEDED(status)) {
            fprintf(stdout, "DSPProcessor_Attach succeeded.\n");
        }
        else {
            fprintf(stdout, "DSPProcessor_Attach failed. Status =
0x%x\n",
                status);
        }
    }

    return (status);
}
InitializeNode – Allocate memory for a DSP node through DSPNode_Allocate.
Then create and run the DSP node on the attached DSP through
DSPNode_Create and DSPNode_Run

/*
 *  ======== InitializeNode ========
 *  Perform node related initialization.
 */
static DSP_STATUS InitializeNode(DMMCOPY_TASK *copyTask)
{
    BYTE            argsBuf[ARGSIZE + sizeof(ULONG)];
    DSP_CBDATA      * pArgs;
    DSP_NODEATTRIN  nodeAttrIn;
    DSP_UUID        uuid;
    DSP_STATUS      status = DSP_SOK;

    uuid = DMMCOPY_TI_uuid;

    nodeAttrIn.uTimeout = DSP_FOREVER;
    nodeAttrIn.iPriority = 5;

    pArgs = (DSP_CBDATA *)argsBuf;
    pArgs->cbData = ARGSIZE;
```

```
        /* Allocate the dmmcopy node. */
        if (DSP_SUCCEEDED(status)) {
            status = DSPNode_Allocate(copyTask->hProcessor, &uuid, pArgs,
                &nodeAttrIn, &copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Allocate succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Allocate failed. Status = 0x%x\n",
                    status);
            }
        }

        /* Create the dmmcopy node on the DSP. */
        if (DSP_SUCCEEDED(status)) {
            status = DSPNode_Create(copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Create succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Create failed. Status = 0x%x\n",
                    status);
            }
        }

        /* Start the dmmcopy node on the DSP. */
        if (DSP_SUCCEEDED(status)) {
            status = DSPNode_Run(copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Run succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Run failed. Status = 0x%x\n",
                    status);
            }
        }

        return (status);
    }
```

6.5.4.2 GPP Application Run

3) **RunTask** - Data streaming between the GPP and the DSP takes place during the run stage of the GPP application. The streaming process is described below:

1. GPP application allocates two buffers of size 0x10000 bytes.

2. GPP application invokes **DSPProcessor_ReserveMemory** to reserve a DSP virtual address space for each allocated buffer using a size that is 4K greater than the allocated buffer to account for automatic page alignment. The total reservation size must also be aligned along a 4K page boundary.

3. GPP application invokes **DSPProcessor_Map** to map each allocated buffer to the DSP virtual address spaces reserved in the previous step.

4. GPP application prepares a message to notify the DSP execute phase of the base addresses of each virtual address space, each of which have been mapped to a buffer allocated on the GPP. GPP application uses **DSPNode_PutMessage** to send the message to the DSP.

5. GPP application then reads fixed-length data from the input file and stores data in the first allocated buffer.

6. GPP application invokes **DSPProcessor_FlushMemory** to ensure that the data cache has been flushed.

7. GPP application prepares a message to notify the DSP execute phase that it has finished writing to the buffer and the DSP may now access the buffer. The message also contains the number of words written to the buffer so that the DSP will know just how much data to copy. The GPP uses **DSPNode_PutMessage** to send the message to the DSP and then invokes **DSPNode_GetMessage** to wait to hear a message back from the DSP.

8. GPP application receives a message back from the DSP execute phase which indicates that the DSP has finishing copying the data from the first buffer to the second buffer. GPP invokes **DSPProcessor_InvalidateMemory** to invalidate the memory range in data cache and then writes the data contained in the second buffer to the output file.

9. Steps 5 – 7 repeat until all of the data from the input file has been copied and written to the output file.

10. GPP application calls **DSPProcessor_UnMap** for each virtual base address to unmap the reserved DSP virtual address space beginning at each address from each of the allocated buffers.

11. GPP application then invokes **DSPProcessor_UnReserveMemory** to unreserve the virtual address space that had been previously reserved using **DSPProcessor_ReserveMemory**. At this point, it becomes available to be reserved again at a later time.

12. GPP application frees each allocated buffer at their original base addresses.

6.5.4.3 GPP Application Cleanup

At cleanup, the GPP application reverses the setup process through functions **`CleanupProcessor`**, **`CleanupNode`**, and **`CleanupStreams`**. These functions accomplish the following:

4) **`CleanupNode`** – Terminate a running node through **`DSPNode_Terminate`**; deallocate a terminated node through **`DSPNode_Delete`**.

```
/*
 *  ======== CleanupNode ========
 */
static DSP_STATUS CleanupNode(DMMCOPY_TASK *copyTask)
{
    DSP_STATUS      status;
    DSP_STATUS      exitStatus;

    if (copyTask->hNode) {
```

```
            // Terminate DSP node.
            status = DSPNode_Terminate(copyTask->hNode, &exitStatus);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Terminate succeeded.\n", status);
            }
            else {
                fprintf(stdout, "DSPNode_Terminate failed: 0x%x\n", status);
            }

            /* Delete DSP node. */
            status = DSPNode_Delete(copyTask->hNode);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPNode_Delete succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPNode_Delete failed: 0x%x\n", status);
            }

            copyTask->hNode = NULL;
        }

        return (status);
    }
```

5) **`CleanupProcessor`** – Detach from an attached DSP through **`DSPProcessor_Detach`**. This ensures the GPP application no longer references a valid DSP.

```
    /*
     *  ======== CleanupProcessor ========
     */
    static DSP_STATUS CleanupProcessor(DMMCOPY_TASK *copyTask)
    {
        DSP_STATUS      status;

        if (copyTask->hProcessor) {
            /* Detach from processor. */
            status = DSPProcessor_Detach(copyTask->hProcessor);
            if (DSP_SUCCEEDED(status)) {
                fprintf(stdout, "DSPProcessor_Detach succeeded.\n");
            }
            else {
                fprintf(stdout, "DSPProcessor_Detach failed.\n");
            }

            copyTask->hProcessor = NULL;
        }

        return (status);
    }
```

6.5.4.4 Notes on Buffer Size

The size of the virtual address space that is reserved should be at least 4K larger than the size of the allocated buffer. This is to account for automatic page alignment of the buffer.

The maximum buffer size can be limited to 13MB for C55L and it is much larger for C64P.

Please refer to platform-specific release notes for information on maximum buffer size.

If the specified input file is an odd number of bytes, the resulting output file will consist of an extra byte. This is because the DSP must copy using the number of words instead of the number of bytes.

6.5.5 Running the Dmmcopy Application

While the code example above indicates what the developer would ordinarily include to setup the stream transport modes appropriately, the GPP console application **dmmcopy.out** will automatically set up the stream attributes based on user input. Therefore, the developer does not need to edit the source to test the different transport modes.

The GPP console application **dmmcopy.out** takes the following arguments:

```
<input_file>            File containing input data.
<output_file>           File to store output data.
```

6.5.6 Summary

The **dmmcopy** console application and DSP task node in this chapter demonstrate the use of Dynamic Memory Mapping in DSP/BIOS Bridge. It describes the GPP and API calls needed to reserve and map buffers allocated on the GPP to a DSP virtual address space.

6.5.7 A Note on Dynamic Memory Mapping

Dynamic Memory Mapping allows GPP applications to allocate large buffers which are then mapped to the DSP virtual address space using the MMU. For specific information on how to properly use Dynamic Memory Mapping in your application, please refer to the *DSP/BIOS Bridge Application Note for Dynamic Memory Mapping.*

7 Overlaying DSP/BIOS Bridge Nodes

For performance reasons, it may be desirable to run several nodes in on-chip memory, but they may not all fit in the same space all at the same time. If the nodes do not need to run at the same time, they can share regions of on-chip memory, and DSP/BIOS Bridge will manage the overlaying of these nodes in the memory regions they share. Node phases are 'loaded' (node sections copied to overlay memory) when calls to **`DSPNode_Create`**, **`DSPNode_Run`**, and **`DSPNode_Delete`** are made, and DSP/BIOS Bridge will prevent the overlay memory from being overwritten by other nodes while it is in use. What is overlayed when the node phase is loaded may include the node code, or data used in the node phase.

Bridge will not prevent the loading of multiple instances of the same overlay node. In this case, the code and data sections are not reloaded but are merely shared between all instances of a node as global. The node developer should keep this model in mind when determining how data sections are overlayed and how the data is managed by the node.

7.1 Defining Overlay Sections

Sections of nodes to be overlayed can be specified using the UNION statement within a SECTIONS directive of a linker command file to specify separate load and run addresses. Example linker command files will be shown in examples, but for a complete description of linker command file syntax, please refer to the *TMS320C55x Assembly Language Tools User's Guide*. As an example, suppose we have two nodes, node0 and node1, that we would like to load in external memory, and run in internal memory. We would decide which of the nodes' code or data sections we want loaded in external memory, but run in internal memory. Then we would write a linker command file that uses UNION statements to declare the *run addresses* of these sections to be in internal memory segments, and the *load addresses* to be in external memory segments. The program is then linked with this linker command file. When DSP/BIOS Bridge loads the program, the overlayed node sections are loaded at their *load addresses*. When one of the nodes is to be run, DSP/BIOS Bridge handles the overlaying from external to internal memory.

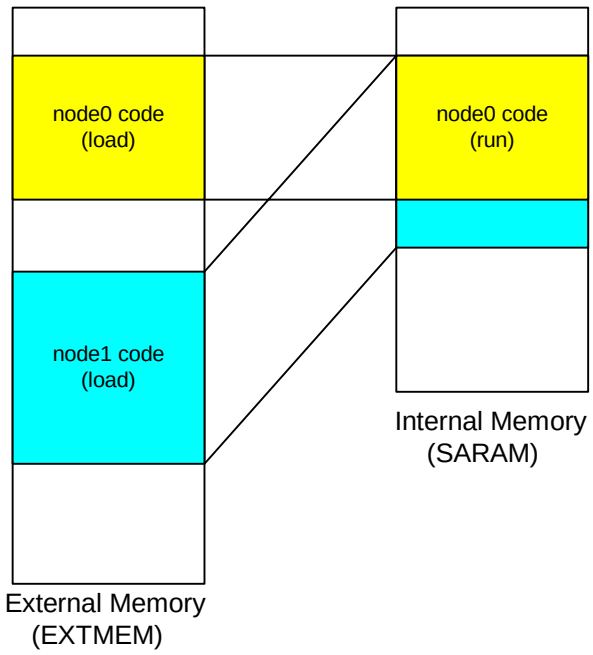

Figure 27 Overlay of two nodes. DSP/BIOS Bridge loads node0 and node1 code in external memory,
and copies each of the nodes' code to internal memory, when the node code is to be run.

The following statement in a linker command file will allow code sections of node0 and node1 to be run in the same region of memory.

```
SECTIONS {
        /*
            *  Set alignment to 4 byte boundary and pad sections with
extra 4 bytes,
            *  since we must do a 32-bit copy from external memory.
            */
            UNION: run = SARAM, align = 4 {
                .node0:text: load = EXTMEM, align = 4, {node0.o64P
(.text) . += 4;}
                .node1:text: load = EXTMEM, align = 4, {node1.o64P
(.text) . += 4;}
            }
        }
```

In this example, code for node0 and node1 is loaded into a region of external memory called EXTMEM. In order for node0 to be run, the section '.node0:text' must be copied from its load address in EXTMEM to its run address in SARAM. When node0 completes, and we want to run node1, the code for node1 can then be copied to internal memory, overwriting the code for node0. The copying of node0 and node1 code from external to internal memory is handled transparently by DSP/BIOS Bridge. When a call to **DSPNode_Create** is made for node0, DSP/BIOS Bridge will copy the section '.node0:text' to its run address, and then call the node's create phase function. DSP/BIOS Bridge will also manage the overlay memory, so that in this example, node1 cannot be created until node0 is deleted.

The name of a section to be overlayed must have the following format.

.node_name[:phase][:section_name]

The fields are defined as

- *node_name* – This is the name of the node that is specified in the configuration (.cdb) file under the *Name* field of the node's properties. (To view or set the node's name in the .cdb file, right click on the node in the Configuration Tool and select properties.)
- *phase* – This field is optional and if present, must be one of **create**, **delete**, or **execute**. This field indicates the phase that the section will be overlayed. If *phase* is **create** or **delete**, DSP/BIOS Bridge will overlay the section during a **DSPNode_Create** or **DSPNode_Delete** call, and will 'unload' the section when the call completes. If *phase* is **execute**, the section will be overlayed in **DSPNode_Run**, and 'unloaded' when the node terminates. If the *phase* field is not specified, the section will be overlayed in **DSPNode_Create**, and will be 'unloaded' when the node is deleted. Not specifying the phase allows the section to be shared by all three node phases.
- *section_name* – This can be any name to further identify the section, and is not used by DSP/BIOS Bridge. For example, *section_ name* could be *data*, *table1*, or *code*.
- The character ':' separates the fields. Also the character '.' must precede the node name.

The following are valid overlay section names, where node0 is the name of a node. The names 'data' and 'table1' have been chosen arbitrarily.

```
.node0
.node0:data
.node0:execute:table1
```

For more information about the build procedure please refer to DSP-side Bridge documentation (to be written in the future)

7.2 Example Linker Command Files for Overlays

7.2.1 Overlay Example 1

This example shows the overlaying of the execute phases of three nodes from external to internal memory. The delete and create phases are not overlayed. This may be useful when the execute phases of the nodes are time critical and need to run in internal memory, but the delete and create phases are not and can run in external memory. Note that the execute phase of only one of the three nodes may be running at one time.

In the linker command file below, the execute phase code of the three nodes is specified to run in internal memory (SARAM) but are loaded in external memory (EXTMEM). In this scenario, calling **DSPNode_Run**(node0) will cause the sections node0:execute:text and node0:execute:data to be copied from their load addresses to their run addresses. Calling **DSPNode_Run** for node1 or node2 will fail until node0 terminates.

Note that overlay sections must be 32-bit aligned for accesses to external memory. Each overlay section is padded with an extra 4 bytes, since we do 32-bit copies from external memory.

```
/*
 *  ======== ovly0_55.cmd ========
 */

-l ovlycfg.cmd55l  /* Generated DSP/BIOS linker command file */

SECTIONS {
    /* Overlay execute phases of 3 nodes. */
    UNION: run = SARAM, align = 4 {
        /*
         *  These sections will be loaded before running the node's
         *  execute phase, and unloaded when the node terminates.
         */
        .node0:execute:text: load = EXTMEM, align = 4,
            {node0execute.o55l(.text) . += 4;}
        .node1:execute:text: load = EXTMEM, align = 4,
            {node1execute.o55l(.text) . += 4;}
        .node2:execute:text: load = EXTMEM, align = 4,
            {node2execute.o55l(.text) . += 4;}
    }

    /*
     *  Overlayed data sections. These sections will be copied
     *  to DARAM when DSPNode_Run() is called.
     */
    UNION: run = DARAM, align = 4 {
        .node0:execute:data: load = EXTMEM, align = 4,
            {*(.node0data) . = align(4);}
        .node1:execute:data: load = EXTMEM, align = 4,
            {*(.node1data) . = align(4);}
        .node2:execute:data: load = EXTMEM, align = 4,
            {*(.node2data) . = align(4);}
    }
}
```

The following figure illustrates sections copied to internal memory for this example when **DSPNode_Run**(node1) is called.

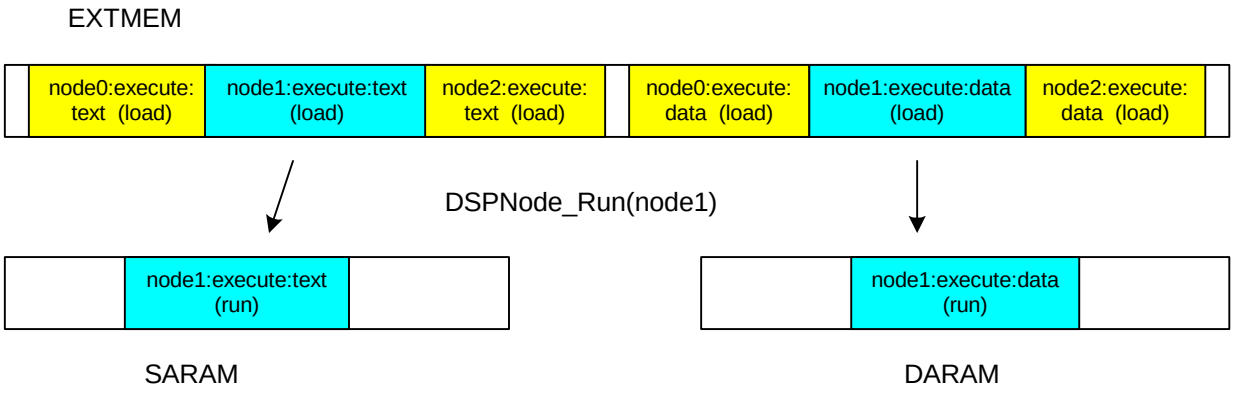

Figure 28 DSPNode_Run(node1) causes 2 sections to be copied from external to internal memory

The table below shows an example sequence of DSPNode calls in the left column. The right column shows DSPNode calls that will fail for this example overlay scenario due to the overlay memory protection mechanism of DSP/BIOS Bridge. The calls in the right column will return an error code of `DSP_EOVERLAYMEMORY` if called within the example sequence in the left column.

Table 19 DSPNode Sequence calls

Example sequence of DSPNode calls	Disallowed DSPNode calls
DSPNode_Create(node0)	
DSPNode_Create(node1)	
DSPNode_Create(node2)	
DSPNode_Run(node0)	
	DSPNode_Run(node1), **DSPNode_Run**(node2)
DSPNode_Terminate(node0)	
DSPNode_Run(node1)	
	DSPNode_Run(node2)
DSPNode_Terminate(node1)	
DSPNode_Run(node2)	

7.2.2 Overlay Example 2

In this example, the execute phase code of three nodes is again overlayed, but we have changed the overlayed data of nodes so that the phase when the data is to be loaded is not specified. This will cause the data to be loaded at the node's create time, and to stay in internal memory until the node is deleted. This may be useful if the data is needed by more than one of the node's phases.

Note the name change of the node data sections, for example, from node0:execute:data to node0:data, in the linker command file below. In the previous example, we would have been able to call **DSPNode_Create**(node0) and **DSPNode_Create**(node1) sequentially, without having to call **DSPNode_Delete**(node0) in between. That is not the case in this example, since the section node0:data cannot be overwritten by node1:data until node0 has been deleted.

```
/*
 *  ======== ovly1_55.cmd ========
 */

-l ovlycfg.cmd55l /* Generated DSP/BIOS linker command file */

SECTIONS {
UNION: run = SARAM, align = 4 {
        /* Overlay execute phase code of three nodes. */
        .node0:execute:text: load = EXTMEM, align = 4,
            {node0execute.o55l(.text) . += 4;}
        .node1:execute:text: load = EXTMEM, align = 4,
            {node1execute.o55l(.text) . += 4;}
        .node2:execute:text: load = EXTMEM, align = 4,
            {node2execute.o55l(.text) . += 4;}
    }

    /*
     *  Data sections loaded in create phase, unloaded in delete phase. Only
     *  instances of one of the 3 nodes can be run at once, because these
     *  sections overlap all 3 node phases.
     */
    UNION: run = DARAM, align = 4 {
              .node0:data: load = EXTMEM, align = 4, {*(.node0data) . =
      align(4);}
              .node1:data: load = EXTMEM, align = 4, {*(.node1data) . =
      align(4);}
              .node2:data: load = EXTMEM, align = 4, {*(.node2data) . =
      align(4);}
          }
      }
```

The following figure illustrates sections copied to internal memory when **DSPNode_Create**(node1) and **DSPNode_Run**(node1) are called.

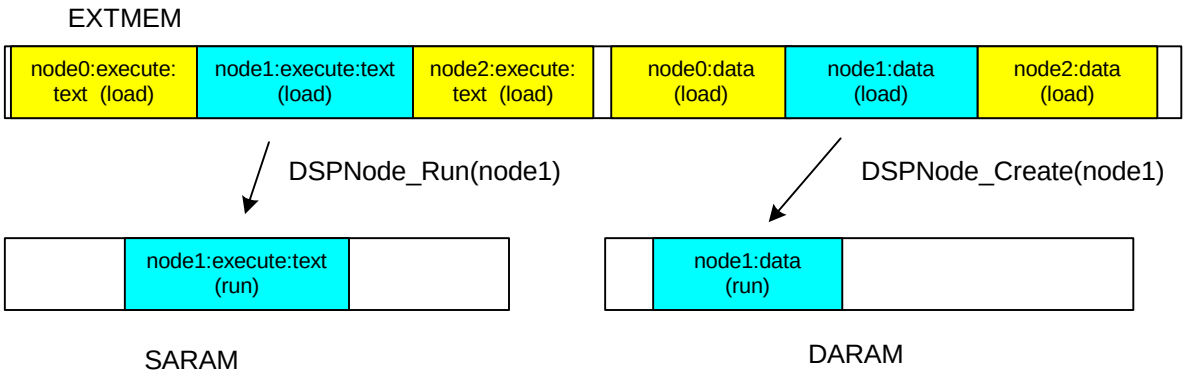

Figure 29 DSPNode_Create(node1) causes data to be copied from external to internal memory. DSPNode_Run(node1) causes execute phase code to be copied to internal memory.

The table below shows a sample sequence of DSPNode calls in the left column for this overlay example. The calls in the right column will fail, returning an error code of DSP_EOVERLAYMEMORY if called within the example sequence in the left column.

Table 20 DSPNode Sequence calls2

Example sequence of DSPNode calls	Disallowed DSPNode calls
DSPNode_Create(node1)	
	DSPNode_Create(node0), **DSPNode_Create**(node2)
DSPNode_Run(node1)	
	DSPNode_Create(node0), **DSPNode_Create**(node2)
DSPNode_Terminate(node1)	
	DSPNode_Create(node0), **DSPNode_Create**(node2)
DSPNode_Delete(node1)	
DSPNode_Create(node0)	
	DSPNode_Create(node1), **DSPNode_Create**(node2)

7.2.3 Overlay Example 3

In the next linker command file the create, delete, and execute phases of a node are overlayed. This example is given to show the limitations of an application with this overlay configuration.

```
/*
 *  ======== ovly2_55.cmd ========
 */
-l ovlycfg.cmd55l  /* Generated DSP/BIOS linker command file */
SECTIONS {
    UNION: run = SARAM, align = 4 {
        /*
         *  Overlay create, delete, and execute phases of a node.
         */
        .node0:create:text: load = EXTMEM, align = 4,
            {node0create.o55l(.text) . += 4;}
        .node0:execute:text: load = EXTMEM, align = 4,
            {node0execute.o55l(.text) . += 4;}
        .node0:delete:text: load = EXTMEM, align = 4,
            {node0delete.o55l(.text) . += 4;}
    }
}
```

The figure below illustrates sections copied to internal memory when **DSPNode_Create**(node0), **DSPNode_Run**(node0), and **DSPNode_Delete**(node0) are called for this overlay configuration. Note that in this example, unlike the previous examples, we cannot create a second instance of node0 if an instance is already running its execute phase. This is because we will not be allowed to overlay the node's create

phase on top of the execute phase, until the first instance of the node terminates. We may still have multiple instances of node0, but only if all instances are created up front.

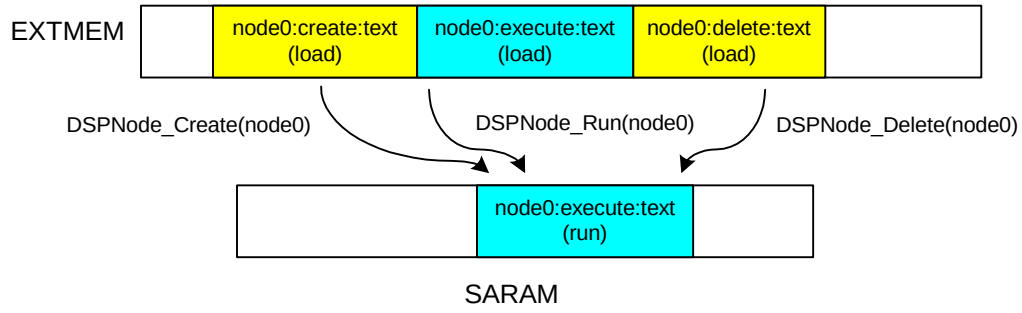

Figure 30 Create, delete, and exeucte phase of a node overlayed, after a call to DSPNode_Run().

The table below shows a sample sequence of DSPNode calls in the left column for this overlay example. The calls in the right column will fail, returning an error code of `DSP_EOVERLAYMEMORY` if called within the example sequence in the left column. Here, we assume that the execute phase will only terminate when **`DSPNode_Terminate`** is called.

Table 21 DDSPNode Sequence Calls

Example sequence of DSPNode calls	Disallowed DSPNode calls
`DSPNode_Create`(node0)	
`DSPNode_Create`(node0) (second instance)	
`DSPNode_Run`(node0)	
	`DSPNode_Create`(node0), **`DSPNode_Delete`**(node0)
`DSPNode_Run`(node0) (second instance)	
	`DSPNode_Create`(node0), **`DSPNode_Delete`**(node0)
`DSPNode_Terminate`(node0)	
	`DSPNode_Create`(node0), **`DSPNode_Delete`**(node0)
`DSPNode_Terminate`(node0) (second instance)	

DSPNode_Create(node0) (third instance)	
DSPNode_Delete(node0) (first instance)	
DSPNode_Delete(node0) (second instance)	

Note that for this example, we cannot delete the node if it is in the running state. This means that if the node does not terminate properly, it cannot be deleted. Therefore, it may not necessarily be a good idea to overlay the execute and delete phases of a node.

7.3 Debugging Considerations

For information about debugging capability for overlays please refer to DSP_BUILD_DOC.doc

8 Dynamically Loading DSP/BIOS Bridge Nodes

This chapter includes the following topics:

- Methods for configuring dynamically loadable DSP/BIOS Bridge nodes
- Creating sharable dynamic DSP/BIOS Bridge libraries
- Advanced techniques for run-time placement of named code/data sections
- Building a base image to support dynamically loadable nodes
- Debugging dynamically loaded DSP code

All of the work necessary to create a system using dynamically loadable DSP/BIOS Bridge libraries is in the configuration and build steps. For further information of configuration and build setup please refer to DSP-side Bridge documentation (to be written in the future).

There are actually no DSP code changes required to use dynamic loading.

8.1 Creating Dynamically Loadable Bridge Nodes

In this section, we will describe the process of creating and integrating dynamically loadable DSP libraries for DSP/BIOS Bridge. This information is applicable to the independent software vendor (ISV) that develops DSP/BIOS Bridge libraries, as well as to the original equipment manufacturer (OEM) that integrates these libraries into the system for the OEM's platform.

A dynamically loadable node has its code and data residing in one or more relocatable object files on the GPP. When the node is run, these object files are linked and loaded onto the DSP on top of an already running DSP base image without any modification to the base image itself. This means that nodes can be loaded onto the DSP only when they are needed, allowing the DSP memory to be easily shared by nodes. The automated scheme reduces the complexity that would typically involve doing the same thing manually, adding to the overall flexibility of a DSP system. New libraries may also be downloaded onto the system during runtime.

Dynamic Loading differs from overlays in that dynamic nodes are not part of the image that is initially loaded onto the DSP.

DSP/BIOS Bridge has the following dynamic loading features.

- You can configure regions of memory for dynamic loading of code only, data only, or for both code and data.
- You can specify a *preferred* dynamic loading memory region where you prefer to have the DSP/BIOS Bridge loader load the node's code or data. In this case, the DSP/BIOS Bridge loader will attempt to load in that region first. If sufficient memory is not available in the preferred memory region, the loader will attempt to find another region for loading the node.
- You can designate a *required* dynamic loading memory region where you require the DSP/BIOS Bridge loader to load the node's code or data. In this case, the DSP/BIOS Bridge loader will attempt to load in that region first. If sufficient memory is not available in the required memory segment, the load will fail.
- You can attain simple memory and phase library granularity by building the node into multiple, independent dynamic phase libraries associated with each node phase and can assign each phase to be loaded to a *preferred* or *required* memory region.
- You may attain further memory granularity by specifying named sections and assigning each specified named section to be loaded to a *preferred* or *required* memory region, irrespective of the default memory region designated for the phase or node as a whole.
- The functionality of the DSP/BIOS Bridge dynamic loader is transparent to the application. Simply calling DSPNode_Create(), DSPNode_Run(), or DSPNode_Delete() will cause any files containing the node's code and data to be dynamically loaded.

Definitions

Terms used throughout this section are defined below.

- **DCD** – A data repository residing on the GPP that provides platform, node, and library information to the Resource Manager.
- **UUID** – This is a *universally unique identifier.* DSP/BIOS Bridge nodes and libraries are all associated with a UUID. These identifiers are used in the DCD for mapping DSP/BIOS Bridge nodes with the DSP object files that contain the nodes' information.
- **Base Image (.dof64P)** – This is a statically linked DSP executable that the OEM will create, to be loaded on the DSP when the system boots up. The base image contains DSP/BIOS, DSP/BIOS Bridge, device drivers, and any other statically linked DSP/BIOS Bridge nodes. Base images are built as *.dof55l* DOFF format executables.
- **BIOS CDB File** – A configuration file for the management of DSP/BIOS objects. Since the CDB file is an integral part of DSP/BIOS, more information may be found in the *TMS320 DSP/BIOS User's Guide.* This file can be manipulated through the Graphical Configuration Tool, GCONF, or through the Textual Configuration Tool, TConf. When a BIOS CDB file is *saved,* assembly files and a linker command file are generated that go into the build of a DSP program. The base image is built with a BIOS CDB file. In like manner, each dynamically loadable node library is built with a CDB file containing at least one node configuration.
- **Dynamic Library (.dll64P)** – A relocatable DSP object file that can be downloaded to virtually any address on the DSP. A dynamic library must not contain any hard coded addresses, but may contain references to undefined symbols. Dynamic libraries are built as *.dll55l* DOFF format object files.
- **Dynamic Node Library (.dll64P)** – A dynamic library that contains the configuration for one or more DSP/BIOS Bridge nodes, and may contain node code and/or data. A dynamically loadable node will always be associated with a dynamic node library; this is the library that the DSP/BIOS Bridge dynamic loader will attempt to load first when loading code and data for the node.
- **Dynamic Phase Library (.dll64P) –** A dynamic library that only contains code and/or data for one node phase. There can be three phase libraries per node, corresponding to the Create, Execute, and Delete phases.
- **Dynamic Dependent Library (.dll64P)** – A dynamic library with global symbols that are referenced by another dynamic library. If a dynamic library has dependent libraries (libraries containing the symbols it references), the dependent libraries must be loaded before the library is loaded.
- **Dynamic Persistent Dependent Library (.dll64P)** – A dependent dynamic library that is loaded with a dynamic phase library and is kept persistent in memory for the entire lifetime of the node until the Delete phase has completed execution.

A DSP/BIOS Bridge dynamic node library can contain configurations, code, and data for either just one or more than one DSP/BIOS Bridge node. The ISV provides a *.dll64P* dynamic library to the OEM. The OEM has configured a *.dof55l* base image onto which the provided library will be loaded dynamically.

The base image and the actual dynamic node are built from two different CDB seed files that are each configured with GCONF/TCONF.

Before describing the specific procedures for building a dynamic library, however, we must first describe the base image memory configuration needed for dynamic loading.

8.1.1 Dynamic Loading Memory Configuration

For detailed procedure about the Dynamic Loading Memory Configuration please refer to DSP-side Bridge documentation (to be written in the future).

8.1.2 Creating and Configuring Dynamic Nodes

You may create and build a base image as you would any DSP image. Your base image may include any static and overlay nodes. However, the configuration of dynamic nodes takes place independently of the OEM base image and its memory configuration. For more information about creating and configuring dynamic nodes please refer to DSP-side Bridge documentation (to be written in the future).

8.1.2.1 Configuring a Node with Phase Library Granularity

In order to achieve finer loading and memory granularity, you may optionally build the node into multiple phase libraries, enabling you to specify the *preferred* or *required* memory region to load the code and data for each phase library, which are only kept loaded for the duration of each phase's execution.

For more information about Configuring a node to support this type of granularity please refer to DSP-side Bridge documentation (to be written in the future)

8.1.2.2 Configuring a Node with Memory Section Granularity

You may wish to direct specific named code and data sections to be loaded in specific memory regions, regardless of the memory settings for the phase library or node library that contains them. For more information about configuring a Node with Memory Section Granularity please refer to DSP-side Bridge documentation (to be written in the future).

8.1.3 Creating and Configuring Dynamic Dependent Libraries

A dynamic dependent library offers you the capability of sharing global symbols between multiple dynamic libraries. These libraries reference a dependent library that is build separately from a node or phase library. Such a dependent library may contain contain a shared library, socket algorithm, or merely code that must, for some reason, be developed separately. A dependent library may even depend upon other dependent libraries.

A library that references a dependent library must link with the dependent library during the dynamic load process. Prior to this linking step, the dynamic loader must first load the dependent library. Because the dynamic loader must know what dependent libraries to load and how to find them, the dependent libraries must be specifically configured with their own UUIDs. This is because they are not directly affiliated with any particular node library. The UUID must be known not only to the dependent library itself, but also by all of the libraries that reference it. A chart is used to keep track of this information, as described below.

There are two types of dependent libraries:

- Standard
- Persistent

8.1.3.1 Standard Dependent Libraries

A standard dependent library is a library that is referenced by any node or phase library. The dynamic loader loads the dependent library when it is needed in order to link with a parent library. In this case, the dependent library is kept loaded as long as the library that references it is kept loaded. If a single node library that contains the code and data for all three node phases references it, the dependent library is kept loaded for the entire lifetime of the node. However, in the case where it is referenced by an individual phase library, the dependent library is only kept loaded for the duration of the execution of that particular phase. The loader may later reload the dependent library if it is referenced by another phase library.

During the time that a dependent library is loaded, and another node instance or library is loaded that references it, it is never loaded twice since its corresponding symbols already exist within the symbol table.

8.1.3.2 Persistent Dependent Libraries

For standard dependent libraries, the same dependent library may be loaded to different locations in memory from phase to phase. There are situations, however, that require that the locations of the symbols in memory should not change from phase to phase. Such scenarios include DAIS socket algorithms that should only be loaded once during the Create phase and should remain loaded for all phases. Another scenario is when multiple instances of a node may reference *static* data variables, the locations of which must not change during the lifetime of each node. A persistent dependent library is a dependent library that is only loaded once. It is kept *persistent* at the same memory location throughout the lifetime of the node.

8.1.3.3 Configuring Dependent Libraries

To configure a dependent library, create a dependent library chart. This chart is a simple text file. Each line of this chart identifies a dependent library by name and lists its UUID, its persistence status, and all libraries that reference it, according to the following format:

```
# DEPENDENT LIB NAME, DEPENDENT LIB UUID, PERSISTENT, REFERENCED BY
<deplibname>,<deplib uuid>,<0 | 1>,<list of reference libraries>
```

An example, named **pingdeplibschart.txt,** is shown below:

```
# DEPENDENT_LIB_NAME, DEPENDENT_LIB_UUID, PERSISTENT, REFERENCED_BY
pingdyndata,12A3C3BF_D015_11D4_9F69_00C04F3A59AF,1,pingCreatedyn
pingdynlib,12A3C3C8_D015_11D4_9F69_00C04F3A59AF,0, pingCreatedyn,
pingExecutedyn, pingDeletedyn
```

This chart identifies ***pingdyndata.dll55l (.dll64P)*** as a dependent library (omitting the file extension) that has a UUID of ***12A3C3BF_D015_11D4_9F69_00C04F3A59AF.*** The '1' that follows the UUID indicates that this library is to be loaded persistently, and ***pingCreatedyn.dll55l (.dll64P),*** a phase library, is the only known library that references this dependent library. Since the library is persistent, when **pingCreatedyn.dll55l *(.dll64P)*** is loaded, the dynamic loader will load **pingdyndata.dll55l *(.dll64P)*** and keep it loaded for all node phases.

In contrast, the chart also identifies ***pingdynlib.dll55l (.dll64P)*** as a dependent library that is not persistent. It is referenced by ***pingCreatedyn.dll55l (.dll64P), pingExecutedyn.dll55l (.dll64P),*** and **pingDeletedyn.dll55l *(.dll64P)*,** which means that the dynamic loader loads **pingdynlib.dll55l *(.dll64P)*** when each phase library is loaded and unloads it when each phase library is unloaded.

It is important to note here that when building a dependent library by itself, without the knowledge of what libraries will be referencing it or whether it will be loaded persistently, the chart may omit the **PERSISTENT** and **REFERENCED_BY** fields. The line in the chart would then resemble the following:

```
pingdyndata,12A3C3BF_D015_11D4_9F69_00C04F3A59AF
```

However, when building the libraries that will reference this dependent library, the chart must contain all of the necessary fields. This is because the reference libraries must be rebuilt with the knowledge of their dependent libraries' UUIDs.

8.1.4 Building the Dynamic Node

There are three methods that pertain to building dynamic nodes. The first method concerns the building of a single dynamic node library, which contains not only the node configuration, but also the node's code and data for all three phases. The second method concerns the building of multiple dynamic libraries, which together belong to one node. This would include individual libraries containing the node configuration and the code and data for individual phases. The third method concerns the building of a dynamic library with memory granularity support.

8.1.4.1 Building a Node with a Single Dynamic Node Library

The **ping** node may be built in three easy steps. The first step is compilation of the source files. The second step involves linking of the libraries to produce the final, relocatable COFF object library. The third step involves converting the final library into a Dynamic Object File Format (DOFF), which can be interpreted by the DSP/BIOS Bridge loader. Below steps explains the procedure to build a node with a single dynamic node library, for detailed information please refer to DSP-side Bridge documentation (to be written in the future).

8.1.4.1.1 COMPILATION

For example, to build the **ping** node library, we could compile the file *ping.c* as follows:

```
cl55 –ml –I\ti\c5500\bios\include –I\ti\c5500\cgtools\include ping.c

cl55 –ml –I\ti\c5500\bios\include –I\ti\c5500\cgtools\include
pingdllcfg.s55
```

We have used the large model (-ml) option to be compatible with DSP/BIOS Bridge.

8.1.4.1.2 LINKING

After compilation, the object files must then be linked with the linker command file into one relocatable object file, for more information about the Linking please refer to DSP-side Bridge documentation (to be written in the future)

```
lnk55 –r –cr –o pingdyn.o55l ping.obj pingdllcfg.obj –l pingdllcfg.cmd
```

(For C55x DSPs)

OR

```
lnk6x –r –cr –o pingdyn.o64P ping.obj pingdllcfg.obj –l pingdllcfg.cmd
```

(For C64x DSPs)

The –r option keeps the relocation information. The input files to the linker are the ping source object file, **ping.obj,** the CDB object file, **pingdllcfg.obj,** and the linker command file, **pingdllcfg.cmd.** The resulting relocatable object file is **pingdyn.o55l (o64P),** the resulting library will be named **pingdyn.lib.** Omission of this option will lead to a symbol linking failure.

The –cr option ensures that cinit processing and initialization occurs at load time. The dynamic loader takes care of this step. Omission of this option may lead to unpredictable runtime behavior on the DSP.

8.1.4.1.3 COFF->DOFF CONVERSION

The relocatable COFF object library need to be converted into into a Dynamic Object File Format, for more informaton about COFF->DOFF conversion please refer to DSP-side Bridge documentation (to be written in the future)

8.1.4.2 Building a Node with Phase Library Granularity

Building a node with phase library granularity means that the node is divided into at least three different libraries, each containing code/data particular to a phase of the node. Another library may contain the node's configuration information, although it is possible that this could be built into one of the phase libraries .For more information about building a Node with Phase Library Granularity please refer to DSP-side Bridge documentation (to be written in the future)

8.1.5 Adding DSP/BIOS Bridge Symbols to the Base Image

Since the base image does not contain any of the dynamic node's code, it may not reference some of the BIOS or DSP/BIOS Bridge functions that are referenced in the dynamic node's code. As a result, when the base image is built, these functions will not be linked into it.

For more information about Adding DSP/BIOS Bridge Symbols to the Base image please refer to DSP-side Bridge documentation (to be written in the future)

8.2 Registering the Dynamic Library

Before we can load and use the **pingdyn.dll55l/pingdyn.dll64P** node library or any phase libraries with a base image, the library and corresponding node must first be registered with the DCD. You may

accomplish this using the exported **DSPManager_RegisterObject()** and **DSPManager_UnregisterObject()** APIs.

Provided within this release is a command-line utility called **dynreg.** This utility is implemented using the APIs provided and provides an easy interface for the management of dynamic nodes and libraries, for which this utility will automate registration and unregistration. The utility will accept all types of dynamic libraries. For more information on the functionality of **dynreg,** please refer to the *DSP/BIOS Bridge Reference Guide.* You may also implement your own utility using the API provided.

To *register* the node and the library that contains the node:

1. If you are using **dynreg:**

   ```
   dynreg –r pingdyn.dll55l
   ```

   This will cause the **ping** node and library to be registered within the DCD database with the UUID configured with the node. DSP/BIOS Bridge will extract the node information from the library.

   Additionally, all dynamic phase libraries and dynamic dependent libraries must also be registered with the DCD database. Dependent libraries are registered using their own assigned UUIDs.

2. If you are implementing your own utility using the APIs provided:

   a. Register the node:

   ```
   status = DSPManager_RegisterObject(&PING_TI_uuid, DSP_DCDNODETYPE,``pingdyn.dll55l``);
   ```

   The UUID specified here must be the same as that which was specified in the node's configuration.

   b. Register the library or libraries that contains the node or phase code/data:

   ```
   status = DSPManager_RegisterObject(&PING_TI_uuid,DSP_DCDLIBRARYTYPE,``pingdyn.dll55l``);
   ```

   This will register the node's pertinent libraries within the DCD database where the DSP/BIOS Bridge loader may know where to access them upon the event of a load.

   Register any dependent libraries using the same DSP_DCDLIBRARYTYPE with their own UUID.

   If you are using dynamic phase libraries, simply register them using the node's UUID and the appropriate type:

   Create phase:

   ```
   status =
DSPManager_RegisterObject(&PING_TI_uuid,DSP_DCDCREATELIBTYPE,``pingCreatedyn.dll55l``);
   ```

   Execute phase:

   ```
   status =
   DSPManager_RegisterObject(&PING_TI_uuid,DSP_DCDEXECUTELIBTYPE,``pingExecutedyn.dll55l``);
   ```

   Delete phase:

   ```
   status =
DSPManager_RegisterObject(&PING_TI_uuid,DSP_DCDDELETELIBTYPE,``pingDeletedyn.dll55l``);
   ```

If you later wish to *unregister* the node and node library:

1. If you are using **dynreg:**

   ```
   dynreg –u ``pingdyn.dll55l``
   ```

This will cause the **ping** node and library to be unregistered from the DCD database with the UUID configured with the node. DSP/BIOS Bridge will extract the node information from the library.

Additionally, all dynamic phase libraries and dynamic dependent libraries must also be unregistered with the DCD database. Dependent libraries are unregistered using their own assigned UUIDs.

2. If you are implementing your own utility using the APIs provided:

a. Unregister the library or libraries that contains the node or phase code/data:

```
status = DSPManager_UnregisterObject(&PING_TI_uuid, DSP_DCDLIBRARYTYPE);
```

b. Unregister the node:

```
status = DSPManager_UnregisterObject(&PING_TI_uuid, DSP_DCDNODETYPE);
```

The UUID specified here must be the same as that which was specified in the node's configuration.

Unregister any dependent libraries using the same DSP_DCDLIBRARYTYPE with their own UUID.

If you are using dynamic phase libraries, simply register them using the node's UUID and the appropriate type:

Create phase:

```
status = DSPManager_UnregisterObject(&PING_TI_uuid, DSP_DCDCREATELIBTYPE);
```

Execute phase:

```
status = DSPManager_UnregisterObject(&PING_TI_uuid, DSP_DCDEXECUTELIBTYPE);
```

Delete phase:

```
status = DSPManager_UnregisterObject(&PING_TI_uuid, DSP_DCDDELETELIBTYPE);
```

8.3 DSP/BIOS Bridge Library Version Control

The flexibility offered by the dynamic loading technology allows a situation where DSP/BIOS Bridge libraries can be developed independently of other components. Without discipline in maintenance of the interfaces between components, a possibility exists where a later version of a DSP library may not link with an existing base image or parent DSP/BIOS Bridge node.

Future versions of TI DSP technology may assist with run time checks on version compatibility, for example by checking version compatibility before attempting a dynamic link. However, in the current version of DSP/BIOS Bridge, maintaining control of library interface specifications, the contract between components, is the responsibility of the system integrator.

8.4 Dynamic Loading Examples Provided in Linux DSP/BIOS Bridge

The dynamic loading examples included with this release include dynamic loading versions of the original static examples that demonstrate XDAIS algorithm integration, GPP/DSP messaging, and data streaming. These examples are Ping, Strmcopy, and XDAIS Scale.

For each of the example nodes below, a sample base image, named **dynbase_tiomap2430.dof64P** is used that does not include any node configurations.

The Linux files for these examples reside under the **\dspbridge\samples** directory, and the DSP side ofiles are found in DSP Sources **\ti\dspbridge\dsp\samples\ ,** For more information about building Dynamic Loading Examples please refer to DSP-side Bridge documentation (to be written in the future).

Build all the new dynamic libraries and copy into the Linux target file system. Run the sample applications exactly the same wat as previously described. The actual loading and unloading of the node is completely transparent to the user.