DSP/BIOS™ BRIDGE REFERENCE

Making**Wireless**

Table of Contents

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

Purpose

This document is a reference guide for the DSP/BIOS Bridge application interfaces. It discusses the GPP-side Bridge APIs and provides a reference to the document discussing the DSP-side Bridge APIs.

2. DSP/BIOS Bridge API Reference

GPP-Side Bridge APIs

2.1.1. Modules and Naming Conventions

The DSP/BIOS Bridge GPP-side API is partitioned into four hierarchical modules and their corresponding objects (data structures):

Manager Module. This module is the highest level module and is primarily used to obtain DSP processor and node configuration information.

Processor Module. This module is used to manipulate DSP processor objects, which represent particular DSP subsystems linked to the GPP. Processor object handles are used to create, execute, and delete signal processing nodes on a particular DSP subsystem.

Node Module. This module is used to manipulate node objects, which represents signal processing elements running on a particular DSP.

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

Each module has a prefix identifying it, and this prefix is used in the name of each function in that module. The prefixes are: **DSPManager**_ for the Manager Module, **DSPProcessor**_ for the Processor Module, **DSPNode**_ for the Node Module, and **DSPStream**_ for the Stream Module.

2.1.2. Data Type Names

The following table summarizes the data types used in the GPP-side API definitions. Instead of using fundamental C-type definitions, (such as int or char), a set of portable types are defined.

Table 1 Data Type Names

Type	Description
BOOL	Boolean value.
BYTE	Unsigned character value.
CHAR	An ANSI text character.

Type	Description
CHARACTER	An ANSI or Unicode text character. The type depends upon the GPP OS being used.
DWORD	Unsigned long integer value.
HANDLE	An opaque handle, i.e., a VOID *.
INT	Signed integer value.
PSTRING	Pointer to a text string. The character type may be ANSI or Unicode, depending upon the GPP OS.
UINT	Unsigned integer value.
ULONG	Unsigned long integer value.
VOID	Empty type.

2.1.3. Status Codes

All GPP-side API functions return the type **DSP_STATUS**, which is a signed long integer. **DSP_STATUS** can identify general errors, e.g., out-of-memory conditions, invalid pointers, etc., as well as specific errors, such as an invalid data format, or a hardware resource is busy. Return codes are partitioned so that it is easy to determine whether something succeeded or failed.

- the leftmost bit indicates success (0) or failure (1),
- the next leftmost 4-bits are a reserved field,
- followed by an 11-bit facility code,
- followed by a 16-bit error or success code.

There can be different “types” of success and failure. In most cases the DSP_SUCCEEDED and DSP_FAILED macros should be used to simplify application code (see *Macros* below).

The API error codes are summarized below in *GPP-side API Return Codes*. A typical use of an API function and the associated **DSP_STATUS** return value is shown in the code fragment below:

```
DSP_STATUS   lStatus;

lStatus = DSPNode_Delete(hNode);
if (DSP_FAILED(lStatus)) {
    ‘ report error to user’
}
```

2.1.4. Handles

The DSP/BIOS Bridge GPP-side API makes use of 32-bit “handles”, which are opaque references to internal objects. DSP/BIOS Bridge handles are *not* interchangeable with other handles in the GPP OS. DSP/BIOS Bridge handles can be passed between GPP OS processes, but as a general rule, the process that creates an object must also be the process that deletes the object.

2.1.5. cbStruct fields

Several DSP/BIOS Bridge GPP-side structures contain a **`cbStruct`** field, which is used as a means of versioning data structures passed between DSP/BIOS Bridge runtime and client software. In all cases, the client should set the **`cbStruct`** field to the size of the containing structure.

So, for example, when setting up the following structure, one should always set **cbStruct** to **sizeof(DSP_BUFFERATTR)**.

```
typedef struct {
    DWORD      cbStruct;
    UINT       uSegment;
    UINT       uAlignment;
} DSP_BUFFERATTR, * DSP_HBUFFERATTR;
```

2.1.6. DSPManager Interface

Include File

```
#include <DSPManager.h>
```

Table 2 DSPManager APIs

API	Description
DSPManager_Close	Closes a handle to the DSP/BIOS Bridge driver.
DSPManager_EnumNodeInfo	Enumerate and get information about nodes configured in the node configuration database
DSPManager_EnumProcessorInfo	Enumerate and get information about available DSP processors
DSPManager_Open	Opens a handle to the DSP/BIOS Bridge module
DSPManager_RegisterObject	Register object with the DCD
DSPManager_UnregisterObject	Unregister object from the DCD
DSPManager_WaitForEvents	Wait for events until specified timeout.

2.1.6.1. DSPManager_Close

```
DBAPI DspManager_Close( UINT  argc,
                        PVOID argp
                      )
```

Description

Closes a handle to the DSP/BIOS Bridge driver.

Parameters

argc	Reserved, set to zero
argp	Reserved, set to NULL

Return Value

DSP_SOK	Success
DSP_EFAIL	An error occurred while closing the handle to the DSP Bridge driver.

Comments

DSPManager_Close closes the handle to the DSP/BIOS Bridge module. This API should be called as soon as the application finishes using DSP/BIOS Bridge. In a multi-process application, **`DSPManager_Close`** should be called from every process needs a handle to the DSP/BIOS Bridge module.

See Also

DSPManager_Open

2.1.6.2. DSPManager_EnumNodeInfo

```
DBAPI DSPManager_EnumNodeInfo ( UINT                uNode,
                                OUT DSP_NDBPROPS *  pNDBProps,
                                UINT                uNDBPropsSize,
                                OUT UINT *          puNumNodes
                              )
```

Description

Enumerate and get configuration information about nodes configured in the node configuration database.

Parameters

uNode	The (arbitrary) numeric node ID.
pNDBProps	Pointer to the DSP_NDBPROPS structure in which the node information will be returned.
uNDBPropsSize	Size of the DSP_NDBPROPS structure.
puNumNodes	Location where the number of nodes configured in the database will be returned.

Return Value

DSP_SOK	Success.
DSP_EINVALIDARG	Parameter uNode is out of range.
DSP_EPOINTER	Parameter pNDBProps or puNumNodes is invalid.
DSP_ECHANGEDURINGENUM	During the enumeration there has been a change in the node configuration database.
DSP_EFAIL	Unable to get node information.
DSP_ESIZE	The size of the specified DSP_NDBPROPS structure is too small to hold all node information, (i.e., uNDBPropsSize is too small).

Comments

DSPManager_EnumNodeInfo is used to retrieve information about nodes configured in the node configuration database.

An API client can call **DSPManager_EnumNodeInfo** to get basic information about a node, such as the type of DSP required, MIPS and memory requirements, etc.

The client starts enumeration by requesting information about the first node in the database, by specifying *uNode* as 0. **DSPManager_EnumNodeInfo** will fill in the specified DSP_NDBPROPS structure, and will also return the number of number of nodes configured, to the location specified by *puNumNodes*. The client can then call **DSPManager_EnumNodeInfo** with incrementing values of *uNode*, to get information about other nodes. If there is a change in the configuration database between the time the first node was enumerated, (i.e., *uNode* = 0), and the current call to **DSPManager_EnumNodeInfo**, then the value DSP_ECHANGEDURINGENUM will be returned. Usually DSP_ECHANGEDURINGENUM will be a signal to the application that it should restart the enumeration.

DSPManager_EnumNodeInfo provides static information about a node, as configured in the DCD. As such, **DSPManager_EnumNodeInfo** can be called prior to allocating the node in the system. Alternatively, **DSPNode_GetAttr** provides dynamic information about a node that has been allocated in the system.

This API returns information on both DSP nodes and IVA1 nodes if applicable.

Note

Individual node properties or the actual set of configured nodes can change during subsequent enumeration calls, resulting in an inconsistent data set, and a return value of `DSP_ECHANGEDURINGENUM`.

The `DSP_NDBPROPS` structure pointed to by *`pNDBProps`* must be allocated before **`DSPManager_EnumNodeInfo`** is called.

Currently, the configuration database is system-wide; that is, it contains information of all the nodes registered in the system, regardless of which processor they can run on. In systems with multiple DSP's DSPManager_EnumNodeInfo() will **not** report which processor each node is capable of running on.

See Also

DSPNode_GetAttr
DSP_NDBPROPS

2.1.6.3. DSPManager_EnumProcessorInfo

```
DBAPI DSPManager_EnumProcessorInfo ( UINT                    uProcessor,
                                     OUT DSP_PROCESSORINFO * pProcessorInfo,
                                     UINT                    uProcessorInfoSize,
                                     OUT UINT *              puNumProcs
                                   )
```

Description

Enumerate and get configuration information about available processors.

)

Parameters

uProcessor	The (arbitrary) numeric processor ID.
pProcessorInfo	Pointer to the DSP_PROCESSORINFO structure in which the processor information will be returned.
uProcessorInfoSize	Size of the DSP_PROCESSORINFO structure.
puNumProcs	Location where the number of processors configured in the database will be returned.

Return Value

DSP_SOK	Success.
DSP_EINVALIDARG	Parameter uProcessor is out of range.
DSP_EPOINTER	Parameter pProcessorInfo or puNumProcs is invalid.
DSP_EFAIL	Unable to get processor information.
DSP_ESIZE	The size of the specified DSP_PROCESSORINFO structure is too small to hold all processor information, (i.e., uProcessorInfoSize is too small).

Comments

DSPManager_EnumProcessorInfo is used to retrieve information about available processors.

An API client can call **DSPManager_EnumProcessorInfo** to get basic information about a processor, such as the speed of the processor, the amount of RAM visible to the processor, etc. The client can then use this information to determine if the processor is a good candidate for the tasks that need to be allocated to a processor, (or whether a different processor is a better candidate). Once a processor is chosen, clients can attach to it with **DSPProcessor_Attach**.

The client can get information about the first processor by specifying *uProcessor* as 0. **DSPManager_EnumProcessorInfo** will return the number of processors registered in the configuration database in the location specified by *puNumProcs*. The client can then call **DSPManager_EnumProcessorInfo** with incrementing values of *uProcessor*, to get information about the other processors.

DSPManager_EnumProcessorInfo provides static information about a processor. Alternatively, **DSPProcessor_GetResourceInfo** provides dynamic information about a processor's resources.

Note

The DSP_PROCESSORINFO structure pointed to by *pProcessorInfo* must be allocated before **DSPManager_EnumProcessorInfo** is called.

See Also

DSPProcessor_GetResourceInfo
DSP_PROCESSORINFO

2.1.6.4. DSPManager_Open

```
DBAPI DspManager_Open ( UINT  argc,
                        PVOID argp
                      )
```

Description

Opens a handle to the DSP/BIOS Bridge module.

Parameters

argc	Reserved
argp	Reserved

Return Value

DSP_SOK	Success.
DSP_EFAIL	Error occurred while opening the hanlde to the bridge module.

Comments

DSPManager_Open should be the first function called before calling any other DSP/BIOS Bridge API. In a multi-process application, **DSPManager_Open** should be called from each process. In a multi-threaded application, it is sufficient to call **DSPManager_Open** only once.

See Also

DSPManager_Close

2.1.6.5. DSPManager_RegisterObject

```
DBAPI DSPManager_RegisterObject( DSP_UUID *     pUuid,
                                 DSP_DCDOBJTYPE objType,
                                 CHAR *         pszPathName
                               )
```

Description

Register a library object with the DSP/BIOS Bridge Configuration Database (DCD).

Parameters

PUuid	Input Pointer to a DSP_UUID structure.
ObjType	Where the object type can be:
DSP_DCDNODETYPE	A node object.
DSP_DCDPROCESSORTYPE	A processor object.
DSP_DCDLIBRARYTYPE	A library object.
DSP_DCDCREATELIBTYPE	A create-phase library object
DSP_DCDEXECUTELIBTYPE	An execute-phase library object
DSP_DCDDELETELIBTYPE	A delete-phase library object
pszPathName	Location of library/object in filesystem.

Return Value

DSP_SOK	Success.
DSP_EFAIL	Unable to register library.

Comments

DSPManager_RegisterObject is used to register dynamic nodes and their dependent libraries with the DCD. It may also be used to register dynamic libraries that do not contain node information.

Note that statically linked DSP nodes are automatically registered into the DCD at build time, by checking the "Register in DCD" checkbox in the DSP node properties dialog box in the DSP/BIOS Bridge configuration. Therefore, there is no need to call this API for statically linked nodes.

DSPManager_RegisterObject is designed to be an OEM function, and should be called from a OEM defined DCD maintenance utility on behalf of applications.

During development, developers should use the **dynreg** utility supplied with the DSP/BIOS Bridge.

Note

The pathname of the library **must** not be changed once it is registered with DCD. This is particularly true for dynamically loaded libraries where the library is accessed repeatedly during runtime.

This API is only available for platforms that support dynamic loading of DSP libraries.

This API is not supported for IVA1 nodes.

See Also

DSP_UUID

DSPManager_UnregisterObject

2.1.6.6. DSPManager_UnregisterObject

```
DBAPI DSPManager_UnregisterObject ( DSP_UUID *     pUuid,
                                    DSP_DCDOBJTYPE objType
                                  )
```

Description

Unregister library object from DCD.

Parameters

pUuid	Pointer to a DSP_UUID structure
ObjType	Where the object type can be:
DSP_DCDNODETYPE	A node object.
DSP_DCDPROCESSORTYPE	A processor object.
DSP_DCDLIBRARYTYPE	A library object.
DSP_DCDCREATELIBTYPE	A create-phase library object
DSP_DCDEXECUTELIBTYPE	An execute-phase library object
DSP_DCDDELETELIBTYPE	A delete-phase library object

Return Value

DSP_SOK	Success.
DSP_EFAIL	Unable to get node information.

Comments

DSPManager_UnregisterObject is used to unregister dynamic nodes and their dependent libraries from the DCD. It may also be used to unregister dynamic libraries that do not contain node information.

DSPManager_UnregisterObject is designed to be an OEM function, and should be called from an OEM defined DCD maintenance utility on behalf of applications.

During development, developers should use the **dynreg** utility supplied with the DSP/BIOS Bridge DDK.

Note

This API is only available for platforms that support dynamic loading of DSP libraries.

This API is not supported for IVA1 nodes.

See Also

DSP_UUID

DSPManager_RegisterObject

2.1.6.7. DSPManager_WaitForEvents

```
DBAPI DSPManager_WaitForEvents ( DSP_HNOTIFICATION * aNotifications,
                                 UINT                uCount,
                                 OUT UINT *          puIndex,
                                 UINT                uTimeout
                               )
```

Description

Block on any Bridge event(s).

Parameters

aNotifications	Array of pointers to notification objects.
uCount	Number of elements in aNotifications array.
puIndex	Index of signaled notification object.
uTimeout	Timeout interval in milliseconds.

Return Value

DSP_SOK	Success.
DSP_ETIMEOUT	Unable to get node information.
DSP_EINVALIDARG	Invalid argument passed.

Comments

This function allows users to wait on a single, or multiple Bridge notification objects, after they've been registered with **`DSPProcessor_RegisterNotify`**, **`DSPNode_RegisterNotify`**, or **`DSPStream_RegisterNotify`**.

When **`DSPManager_WaitForEvents`** returns successfully, the caller is provided with a single index corresponding to a single event. As a result, the user is only passed the index of one event satisfying the wait, even if multiple events have been signaled.

It is important to note that the events within the notification objects are auto-resetting. This means that user applications are not responsible for manually resetting the events.

The events within the notification objects are not counting events. Remember to take into account that your event may have been signaled multiple times (by a higher priority thread, for example). The event will only change state one time before it is reset.

Please see the description of the `DSP_NOTIFICATION` structure for more details.

Note

Applications cannot wait on non-DSP/BIOS Bridge events using this API. At most one thread can wait on a notification object at a time, so the same notification object must not be used in more than one wait call simultaneously. If an application needs to wait for the same notification simultaneously in multiple threads, then it should register for that same notification multiple times using different notification objects.

See Also

DSPProcessor_RegisterNotify
DSPNode_RegisterNotify
DSPStream_RegisterNotify
DSP_NOTIFICATION

2.1.7. DSPProcessor Interface

Include File

```
#include <DSPProcessor.h>
```

Table 3 DSPProcessor APIs

API	Description
DSPProcessor_Attach	Reserve (GPP-side) resources for a processor; get a handle for the processor
DSPProcessor_Ctrl	Pass control information to the processor's GPP device driver
DSPProcessor_Detach	Free (GPP-side) resources dedicated to a processor
DSPProcessor_EnumNodes	Enumerate nodes currently allocated for a processor.
DSPProcessor_FlushMemory	Flushes a buffer from the GPP data cache
DSPProcessor_GetResourceInfo	Get processor resource information
DSPProcessor_GetState	Report the (execution) state of a processor
DSPProcessor_InvalidateMemory	Invalidates Memory range from GPP data cache
DSPProcessor_Load	Reset a processor and load a base program image
DSPProcessor_Map	Map a GPP buffer to a reserved virtual address
DSPProcessor_RegisterNotify	Register API client to be notified on specific events
DSPProcessor_ReserveMemory	Reserve memory from the DSP virtual address space
DSPProcessor_Start	Start a processor, running the code loaded by DSPProcessor_Load
DSPProcessor_UnMap	Unmap a GPP buffer from a reserved virtual address
DSPProcessor_UnReserveMemory	Free memory reserved from the DSP virtual address space

The **`DSPProcessor_`** functions provide an interface to specific processor objects. Each **`DSPProcessor_`** function call requires a processor handle, which identifies the individual processor. This processor handle is returned by a successful invocation of the **`DSPProcessor_Attach`** function.

Each function is described below, in alphabetic order, in manual-page format.

2.1.7.1. DSPProcessor_Attach

```
DBAPI DSPProcessor_Attach ( UINT                                uProcessor,
                            OPTIONAL CONST DSP_PROCESSORATTRIN * pAttrIn,
                            OUT DSP_HPROCESSOR *                phProcessor
                          )
```

Description

Prepare for communication with a particular processor, and return a handle to the processor object.

Parameters

uProcessor	Specifies the index of the processor to attach to. This arbitrary index is the same as the processor ID (uProcessor) for DSPManager_EnumProcessorInfo calls.
pAttrIn	Pointer to the DSP_PROCESSORATTRIN structure that contains the attributes to be applied to the processor. If the value of this parameter is NULL, a default set of attributes will be assigned to the processor.
phProcessor	Pointer to the location where the processor handle will be returned.

.

Return Value

DSP_SOK	Success.
DSP_EPOINTER	Parameter phProcessor is not valid.
DSP_EINVALIDARG	Parameter uProcessor is invalid.
DSP_EFAIL	Unable to attach the processor.
DSP_SALREADYATTACHED	Successful attach; note that another GPP client was already attached to this processor.

To use other **DSPProcessor_** functions, you must first select the desired processor with **DSPProcessor_Attach**, and then use the returned processor handle in subsequent **DSPProcessor_** function calls. When your application no longer needs to use a particular processor, you should call **DSPProcessor_Detach** to release the GPP-side resources allocated by **DSPProcessor_Attach**.

DSPProcessor_Attach and **DSPProcessor_Detach** do not affect the execution state of the processor.

uProcessor is the numeric processor ID, corresponding to that used by **DSPManager_EnumProcessorInfo**.

The caller must check the return value of **DSPProcessor_Attach** for validity. A return value of DSP_SALREADYATTACHED indicates a valid handle has been returned to *phProcessor*, and that another application or driver on the GPP is already attached to the specific processor. If an error occurs, the value at **phProcessor* is unspecified.

Note

If a GPP client is already attached to a processor when **DSPProcessor_Attach** is called on the same processor again, then the attributes specified in DSP_PROCESSORATTRIN will be ignored, i.e., DSP_PROCESSORATTRIN is only useful on the first attach to a processor.

See Also

DSP_PROCESSORATTRIN
DSPProcessor_Detach

2.1.7.2. DSPProcessor_Ctrl

```
DBAPI DSPProcessor_Ctrl( DSP_HPROCESSOR           hProcessor,
                         ULONG                    dwCmd,
                         IN OPTIONAL DSP_CBDATA * pArgs
                       )
```

Description

Pass control information to the GPP device driver managing processors. This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
dwCmd	An implementation-specific command code.
pArgs	A pointer to an arbitrary argument structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	The parameter hProcessor is invalid.
DSP_ETIMEOUT	A timeout occurred before the control information could be sent.
DSP_EFAIL	Unable to send control information.

Comments

This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

2.1.7.3. DSPProcessor_Detach

```
DBAPI DSPProcessor_Detach ( DSP_HPROCESSOR  hProcessor )
```

Description

Close a processor and de-allocate all (GPP) resources reserved for it by **DSPProcessor_Attach**.

Parameters

hProcessor	Handle of the processor to be released.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EFAIL	A failure occurred, unable to detach.

Comments

DSPProcessor_Detach releases the GPP-side resources used to control and communicate with a particular processor.

DSPProcessor_Detach does not affect the actual execution state of the processor; it merely releases GPP-side resources.

Note

hProcessor must be a valid handle returned from a previous call to **DSPProcessor_Attach**.

See Also

DSPProcessor_Attach

2.1.7.4. DSPProcessor_EnumNodes

```
DBAPI DSPProcessor_EnumNodes( DSP_HPROCESSOR hProcessor,
                              IN DSP_HNODE * aNodeTab,
                              IN UINT        uNodeTabSize,
                              OUT UINT *     puNumNodes,
                              OUT UINT *     puAllocated
                            )
```

Description

Enumerate the nodes currently allocated for a processor.

Parameters

hProcessor	Handle of the processor for which nodes are to be enumerated, as returned from a successful call to DSPProcessor_Attach.
aNodeTab	The first location of an array allocated for node handles. DSPProcessor_EnumNodes will place handles to the nodes currently allocated on the processor into this table.
uNodeTabSize	The number of (DSP_HNODE) handles that can be held to the memory the client has allocated for aNodeTab.
puNumNodes	Location where DSPProcessor_EnumNodes will return the number of valid node handles written to aNodeTab.
puAllocated	Location where DSPProcessor_EnumNodes will return the number of nodes that are allocated on the processor.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EPOINTER	Parameter aNodeTab, puNumNodes, or puAllocated is invalid.
DSP_ESIZE	The amount of memory allocated for aNodeTab is insufficient. That is, the number of nodes actually allocated on the processor is greater than the value specified for uNodeTabSize.
DSP_EFAIL	A failure occurred during enumeration.

Comments

DSPProcessor_EnumNodes is used to take a 'snapshot' of the nodes currently allocated on a processor. After calling **DSPProcessor_EnumNodes** to get node handles, a client can get detailed information about a node by passing its handle to **DSPNode_GetAttr.**

DSPProcessor_EnumNodes will return an array of handles of all allocated nodes in a block of memory allocated by the API client. If the client has not allocated enough memory the error code DSP_ESIZE will be returned. If this happens, the client can look at the value returned at *puAllocated* to determine the amount of memory needed to hold all node handles. The client is responsible for freeing all memory blocks passed to **DSPProcessor_EnumNodes**.

The **DSPProcessor_EnumNodes** function returns handles to the nodes currently allocated on a processor. Alternatively, **DSPManager_EnumNodeInfo** is used to get static information about nodes, as configured in the DCD.

DSPProcessor_EnumNodes will not cause queries to the processor, and the function will not block execution.

See Also

DSPManager_EnumNodeInfo
DSPNode_GetAttr

2.1.7.5. DSPProcessor_FlushMemory

```
DBAPI DSPProcessor_FlushMemory( DSP_HPROCESSOR hProcessor,
                                PVOID          pMpuAddr,
                                ULONG          ulSize,
                                ULONG          ulFlags
                              )
```

Description

Flushes a buffer from the MPU data cache.

Parameters

hProcessor	Handle of the processor for which nodes are to be enumerated, as returned from a successful call to DSPProcessor_Attach.
pMpuAddr	The start address of the buffer to be flushed.
ulSize	The size of the buffer to be flushed.
ulFlags	Currently Reserved.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EFAIL	A failure occurred during flush.

Comments

DSPProcessor_FlushMemory will flush the GPP data cache range starting from pMpuAddr to pMpuAddr+ ulSize.

See Also

DSPProcessor_Attach

2.1.7.6. DSPProcessor_GetResourceInfo

```
DBAPI DSPProcessor_GetResourceInfo( DSP_HPROCESSOR         hProcessor,
                                    UINT                   uResourceMask,
                                    OUT DSP_RESOURCEINFO * pResourceInfo,
                                    UINT                   uResourceInfoSize
                                  )
```

Description

Get information about a processor's resources.

Parameters

hProcessor	Handle of the processor for which resource info is to be reported, as returned from a successful call to DSPProcessor_Attach.
uResourceMask	Mask of type of resource to be reported
pResourceInfo	Pointer to the DSP_RESOURCEINFO structure, which specifies the type of resource information needed, and in which the processor resource information will be returned.
uResourceInfoSize	Size of the DSP_RESOURCEINFO structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EVALUE	uResourceType in the DSP_RESOURCEINFO structure is invalid.
DSP_EPOINTER	Parameter pResourceInfo is invalid.
DSP_EWRONGSTATE	The processor is not in the PROC_RUNNING state.
DSP_ETIMEOUT	A timeout occurred before the processor responded to the query.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EFAIL	Unable to get resource information.
DSP_ESIZE	The size of the specified DSP_RESOURCEINFO structure is too small to hold all resource information, (i.e., uResourceInfoSize is too small).

Comments

DSPProcessor_GetResourceInfo is used to retrieve information about the resources currently available on a processor. The type of resource to retrieve information about is specified in the field *uResourceType* in the DSP_RESOURCEINFO structure specified by *pResourceInfo*.

DSPProcessor_GetResourceInfo will block until the processor responds to the resource query.

DSPProcessor_GetResourceInfo is used to get the current resource load of a processor, whereas **DSPManager_EnumProcessorInfo** is used to get a static measure of basic processor capabilities before any resources are consumed.

Note

The DSP_RESOURCEINFO structure pointed to by *pResourceInfo* must be allocated before **DSPProcessor_GetResourceInfo** is called.

DSPProcessor_GetResourceInfo should only be called for a processor that is in the PROC_RUNNING state.

See Also

DSPManager_EnumProcessorInfo
DSP_RESOURCEINFO

2.1.7.7. DSPProcessor_GetState

```
DBAPI DSPProcessor_GetState (DSP_HPROCESSOR           hProcessor,
                             OUT DSP_PROCESSORSTATE * pProcState,
                             UINT                     uStateInfoSize
                            )
```

Description

Report the state of the specified processor

Parameters

hProcessor	Handle of the processor for which status is to be reported, as returned from a successful call to DSPProcessor_Attach.
pProcStatus	Pointer to the DSP_PROCESSORSTATE structure in which the processor status will be returned.
uStateInfoSize	Size of the DSP_PROCESSORSTATE structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EPOINTER	Parameter pBuf is not valid.
DSP_EFAIL	Unable to retrieve trace buffer information.
DSP_ESIZE	The size of the specified DSP_PROCESSORSTATE structure is too small to hold all state information, (i.e., uStateInfoSize is too small).

Comments

The `DSP_PROCESSORSTATE` structure pointed to by *`pProcStatus`* must be allocated before **`DSPProcessor_GetState`** is called.

See Also

DSPProcessor_Attach
DSP_PROCESSORSTATE

2.1.7.8. DSPProcessor_InvalidateMemory

```
DBAPI DSPProcessor_InvalidateMemory( DSP_HPROCESSOR hProcessor,
                                     PVOID          pMpuAddr,
                                     ULONG          ulSize,
                                   )
```

Description

Invalidates a buffer from the MPU data cache.

Parameters

hProcessor	Handle of the processor for which nodes are to be enumerated, as returned from a successful call to DSPProcessor_Attach.
pMpuAddr	The start address of the buffer to be Invalidated.
ulSize	The size of the buffer to be invalidated.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EFAIL	A failure occurred during Invalidate.

Comments

DSPProcessor_InvalidateMemory will invalidate the GPP data cache range starting from pMpuAddr to pMpuAddr+ ulSize.

See Also

DSPProcessor_FlushMemory
DSPProcessor_Attach

2.1.7.9. DSPProcessor_Load

```
DBAPI DSPProcessor_Load ( DSP_HPROCESSOR   hProcessor,
                          IN CONST INT     iArgc,
                          IN CONST CHAR ** aArgv,
                          IN CONST CHAR ** aEnvp
                        )
```

Description

Reset a processor and load a new base program image. This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
iArgc	The number of arguments (strings) in aArgv.
aArgv	An array of arguments (ANSI strings).
aEnvp	An array of environment settings (ANSI strings).

Return Value

DSP_SOK	Success.
DSP_EHANDLE	The parameter hProcessor is an invalid handle.
DSP_EFILE	The processor executable was not found.
DSP_ECORRUPTFILE	Unable to parse the processor executable.
DSP_EPOINTER	Parameter aArgv is invalid.
DSP_EINVALIDARG	Parameter iArgc is invalid.
DSP_EATTACHED	Abort because a GPP client is attached to the processor.
DSP_EACCESSDENIED	Client does not have the required access rights to reset and load the processor.
DSP_EFAIL	Unable to load processor.

Comments

This function, used in conjunction with **DSPProcessor_Start**, allows a new base program image to be loaded to a processor and started, overriding the program image loaded at GPP OS driver initialization time.

iArgc, *aArgv*, and *aEnvp* allow passing of arguments and environment to the executable.

This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

Note

DSPProcessor_Load should only be called when no GPP clients are attached to the specified processor; any clients attached to the processor must be detached before **DSPProcessor_Load** is called.

aArgv[0] must be the name of a valid executable file.

See Also

DSPProcessor_Start

2.1.7.10. DSPProcessor_Map

```
DBAPI DSPProcessor_Map( DSP_HPROCESSOR hProcessor,
                        PVOID          pMpuAddr,
                        ULONG          ulSize,
                        PVOID          pReqAddr,
                        PVOID *        ppMapAddr,
                        ULONG          ulMapAttr
                      )
```

Description

Maps an MPU buffer to the DSP virtual address space.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
pMpuAddr	Starting address of the MPU memory region to map.
ulSize	Size in bytes of the MPU memory region to map.
pReqAddr	Requested DSP start address; Offset-adjusted actual mapped address is returned into ppMapAddr below.
ulMapAttr	Optional endianness attributes and virtual to physical flag. This structure is ignored on OMAP1510 and OMAP1610 platform.
ppMapAddr	Pointer to DSP side mapped BYTE address.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EMEMORY	MPU side memory allocation error.
DSP_ENOTFOUND	Cannot find a reserved region starting with this address.
DSP_EFAIL	A failure occurred during mapping.

Note

DSPProcessor_Map will map a user-allocated buffer to a DSP virtual address space. The virtual address space must have been correctly reserved via **DSPProcessor_ReserveMemory**.

Pass NULL for *pMapAttr* to use default attributes, which assume that the MPU address is virtual and the buffer contains 16-bit little-endian data.

See Also

DSPProcessor_Attach
DSPProcessor_ReserveMemory

2.1.7.11. DSPProcessor_RegisterNotify

```
DBAPI DSPProcessor_RegisterNotify( DSP_HPROCESSOR    hProcessor,
                                   UINT              uEventMask,
                                   UINT              uNotifyType,
                                   DSP_HNOTIFICATION hNotification
                                 )
```

Description

Register to be notified of specific processor events.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
uEventMask	Mask of types of events to be notified about:
DSP_PROCESSORSTATECHANGE	The processor's state has changed.
DSP_PROCESSORRESTART	A critical error has occurred, and the target processor is being restarted.
DSP_PROCESSORATTACH	A GPP client has attached to the target processor.
DSP_PROCESSORDETACH	A GPP client has detached from the target processor.
DSP_SYSERROR	SYS_error was called on the target processor.
DSP_MMUFAULT	A DSP MMU fault occurred.
uNotifyType	Type of notification to be sent:
DSP_SIGNALEVENT	Signal the event specified by hNotification.
hNotification	Handle of a DSP_NOTIFICATION object.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor or hNotification is invalid.
DSP_EVALUE	Parameter uEventMask is invalid.
DSP_ENOTIMPL	The notification type specified in uNotifyType is not supported.
DSP_EFAIL	Unable to register for notification.

Comments

`DSPProcessor_RegisterNotify` allows API clients to register for notification of certain types of events that occur on a processor. For example, an application can register to be notified on processor state changes.

The types of events to be notified about are defined by *`uEventMask`*. The type of notification to be sent is defined by *`uNotifyType`*.

hNotification is a handle to a `DSP_NOTIFICATION` object that defines the notification object name or handle. The type of named object or handle specified in the notification structure will depend upon the notification type specified by *`uNotifyType`*.

A client can de-register for notifications by calling **`DSPProcessor_RegisterNotify`**, specifying the same notification object handle that was used in a previous **`DSPProcessor_RegisterNotify`** call, with *`uEventMask`* now set to zero. De-registering in this manner will not flush any pending notifications, so the client must be able to handle notifications that might already be pending.

Note

Named events are not supported

See Also

DSPNode_RegisterNotify
DSPStream_RegisterNotify

2.1.7.12. DSPProcessor_ReserveMemory

```
DBAPI DSPProcessor_ReserveMemory( DSP_HPROCESSOR hProcessor,
                                  ULONG          ulSize,
                                  PVOID *        ppRsvAddr
                                )
```

Description

Reserve a vitually contiguous region of DSP address space.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
ulSize	Size of the address space to reserve
ppRsvAddr	Pointer to the DSP side reserved BYTE address

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EINVALIDARG	Parameter ulSize is not 4KB page-aligned.
DSP_EMEMORY	Cannot reserve memory space of this size.
DSP_EFAIL	A failure occurred during reserve.

Comments

`DSPProcessor_ReserveMemory` is responsible for reserving *`ulSize`* bytes of the DSP virtual memory space. A pointer to the address of the reserved space is returned in *`ppRsvAddr`*.

Note

Parameter *ulSize* must be aligned along a 4KB page boundary before it is given to **`DSPProcessor_ReserveMemory`**. This is to ensure that the DSP virtual address space is mapped according to pages.

See Also

DSPProcessor_Attach

2.1.7.13. DSPProcessor_Start

DBAPI DSPProcessor_Start(DSP_HPROCESSOR *hProcessor*)

Description

Start a processor running, executing a base program image loaded by **DSPProcessor_Load**. This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	The parameter hProcessor is an invalid handle.
DSP_EWRONGSTATE	The processor is not in the PROC_LOADED state.
DSP_EFAIL	Unable to start processor.

Comments

This function, used in conjunction with **DSPProcessor_Load**, allows a new base program image to be loaded to a target processor and started, overriding the program image loaded at GPP OS driver initialization time.

This is an OEM-only function, and not part of the DSP/BIOS Bridge application developer's API.

Note

DSPProcessor_Start should only be called following a **DSPProcessor_Load** call to load a base program image to the processor.

See Also

DSPProcessor_Load

2.1.7.14. DSPProcessor_UnMap

DBAPI DSPProcessor_UnMap(DSP_HPROCESSOR *hProcessor*,
PVOID *pMapAddr*
)

Description

Remove an MPU buffer mapping from the DSP virtual address space.

Parameters

hProcessor	Handle of the processor, as returned from a successful call to DSPProcessor_Attach.
pMapAddr	Pointer to the starting address of the mapped DSP virtual address space.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_ENOTFOUND	Cannot find a mapped region starting with this address.
DSP_EFAIL	A failure occurred during unmap.

Comments

`DSPProcessor_UnMap` must be given a DSP starting address of a mapped DSP virtual address space that resulted from a successful call to **`DSPProcessor_Map`**. The function will remove the mapping between the MPU buffer address and the DSP virtual address space. The DSP virtual address space, while not mapped, is still reserved and is available to be mapped to another MPU address.

See Also

DSPProcessor_Attach

DSPProcessor_Map
DSPProcessor_ReserveMemory

2.1.7.15. DSPProcessor_UnReserveMemory

DBAPI DSPProcessor_UnReserveMemory(DSP_HPROCESSOR *hProcessor*,
PVOID *pRsvAddr*
)

Description

Frees a previously reserved region of the DSP virtual address space.

Parameters

hProcessor	Handle of the DSP for which nodes are to be enumerated, as returned from a successful call to DSPProcessor_Attach.
pRsvAddr	Pointer to a DSP side reserved BYTE address.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_ENOTFOUND	Cannot find a reserved region starting with this address.
DSP_EFAIL	A general failure occurred.

Comments

`DSPProcessor_UnReserveMemory` will unreserve the DSP virtual memory space beginning at the given address, allowing the region to be reserved again at a later date.

See Also

DSPProcessor_Attach
DSPProcessor_Map
DSPProcessor_ReserveMemory

2.1.8. DSPNode Interface

Include File

```
#include <DSPNode.h>
```

Table 4 Node interface functions

API	Description
DSPNode_Allocate	Reserve (GPP-side) resources for a node; return a handle for the node
DSPNode_AllocMsgBuf	Allocate a buffer whose descriptor will be passed to a DSP node within a (DSP_MSG) message
DSPNode_ChangePriority	Change a node's runtime priority
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 connection parameter to DSP side
DSPNode_Create	Create a node on a target processor, and execute the node's create-phase function
DSPNode_Delete	Run the node's delete-phase function, and release all target processor side and GPP side resources for the node
DSPNode_FreeMsgBuf	Free a buffer allocated with DSPNode_AllocMsgBuf
DSPNode_GetAttr	Report the attributes of a node, including its execution state
DSPNode_GetMessage	Retrieve a message from a node (if available)
DSPNode_Pause	Temporarily suspend execution of a node
DSPNode_PutMessage	Send a message to a node
DSPNode_RegisterNotify	Register with API to be notified of specific events for this node
DSPNode_Run	Start a node's execute phase; or resume execution of a paused node
DSPNode_Terminate	Signal a node to tell it to exit its execute phase

The **`DSPNode_`** functions support direct communication with, and control of individual nodes on a DSP processor. Each **`DSPNode_`** function call requires a node handle, which identifies the individual node on a specific DSP processor. This node handle is returned by a successful invocation of the **`DSPNode_Allocate`** function.

2.1.8.1. DSPNode_Allocate

```
DBAPI DSPNode_Allocate( DSP_HPROCESSOR                    hProcessor,
                        IN CONST DSP_UUID *               pNodeID,
                        IN CONST OPTIONAL DSP_CBDATA *    pArgs,
                        IN CONST OPTIONAL DSP_NODEATTRIN * pAttrIn,
                        OUT DSP_HNODE *                   phNode
                      )
```

Description

Allocate data structures for controlling and communicating with a node on a specific target processor.

Parameters

hProcessor	Handle of the processor for which the node is to be allocated, as returned from a successful call to DSPProcessor_Attach.
pNodeID	Pointer to a DSP_UUID for the node.
pArgs	Pointer to a DSP_CBDATA structure containing any (optional) arguments to be passed to the node. This parameter can be NULL if there are no arguments for the node.
pAttrIn	Pointer to the DSP_NODEATTRIN structure that contains the attributes to be applied to the node. If the value of this parameter is NULL, a default set of attributes will be assigned to the node.
phNode	Location where the node handle is to be returned. A value of NULL for this argument is an error.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hProcessor is invalid.
DSP_EPOINTER	One of the input pointer parameters (pArgs, pAttrIn, phNode, or pNodeID) is invalid.
DSP_EMEMORY	Insufficient memory was available on the GPP system to allocate a new node.
DSP_EWRONGSTATE	The specified processor is in the wrong state, (this usually means that the DSP subsystem is not running).
DSP_ERANGE	The iPriority field specified in the DSP_NODEATTRIN structure is out of range.
DSP_EUUID	The specified DSP_UUID is not registered in the DSP/BIOS Bridge Configuration Database.
DSP_EFAIL	A failure occurred, unable to allocate the node.

Comments

To use other **DSPNode_** functions, you must begin the allocation of a node by calling **DSPNode_Allocate**.

Any (optional) arguments you wish to provide to the node are specified by the *cData* member array of the DSP_CBDATA structure pointed at by *pArgs*. This array is of arbitrary (DSP processor dependent) length; the length is specified by the *cbData* member variable of the DSP_CBDATA structure.

The fields of the DSP_CBDATA structure are passed as arguments to the DSP node's create function as follows: *cbData* is passed as **int** *argLength*, *cData* is passed as **char *** *argData*.

As part of node allocation, **DSPNode_Allocate** will check the *bCacheOnGPP* flag in the DCD entry for the node. If *bCacheOnGPP* is TRUE, **DSPNode_Allocate** will attempt to cache the node's executable code in GPP memory for quick access during the create phase.

See Also

DSP_CBDATA
DSP_NODEATTRIN
DSPProcessor_Attach
DSPNode_Connect
DSPNode_Delete
DSPNode_Terminate

2.1.8.2. DSPNode_AllocMsgBuf

```
DBAPI DSPNode_AllocMsgBuf( DSP_HNODE                        hNode,
                           UINT                             uSize,
                           IN OPTIONAL DSP_BUFFERATTR *     pAttr,
                           OUT BYTE **                      ppBuffer
                         )
```

Description

Allocate and prepare a buffer whose descriptor will be passed to a target processor node within a (DSP_MSG) message.

Parameters

hNode	Handle of the node, as returned from a successful call to DSPNode_Allocate.
uSize	Size (in GPP bytes) of the buffer to be allocated.
pAttr	Pointer to a DSP_BUFFERATTR structure. A NULL pointer indicates default attributes.
ppBuffer	Address to receive the address of the allocated buffer.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hNode handle.
DSP_EMEMORY	Insufficient memory.
DSP_EPOINTER	ppBuffer is not a valid address.
DSP_EALIGNMENT	The alignment specified via pAttr is not supported.
DSP_EBADSEGID	The segment identifier specified via pAttr does not exist.
DSP_EFAIL	A failure occurred, unable to allocate buffer.
DSP_ESIZE	Invalid size specified.

Comments

DSPNode_AllocMsgBuf allocates and prepares a data buffer that will be passed to a node on the DSP via **DSPNode_PutMessage**. A buffer returned by **DSPNode_AllocMsgBuf** can be written to by the GPP application, and then the buffer's address and size can be sent to the node via **DSPNode_PutMessage**.

DSPNode_AllocMsgBuf implicitly does any preparation needed of the buffer for it to be passed to the node as a message. For example, the buffer may be page locked, and allocated in a GPP/DSP shared memory region.

Buffers allocated with **DSPNode_AllocMsgBuf** must be freed by the client application, when the DSP node is no longer accessing them, via calls to **DSPNode_FreeMsgBuf**.

Buffers allocated with **DSPNode_AllocMsgBuf** must **not** be passed to the DSP with the **DSPStream_Issue** API, as this will not perform the necessary GPP to DSP address conversion on these shared buffers.

See Also

DSPNode_FreeMsgBuf

2.1.8.3. DSPNode_ChangePriority

DBAPI DSPNode_ChangePriority(DSP_HNODE *hNode*,
INT *iPriority*
)

Description

Change a node's runtime priority. For nodes running on the DSP, priorities are changed within the DSP RTOS.

Parameters

hNode	Handle of the node whose priority is to be changed, as returned by a successful call to DSPNode_Allocate.
iPriority	New runtime priority level. Valid values range from DSP_NODE_MIN_PRIORITY to DSP_NODE_MAX_PRIORITY, as reported by the DSP_PROCESSORINFO structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_ERANGE	The parameter iPriority is out of range.
DSP_ENODETYPE	Operation is invalid for this type of node.
DSP_ETIMEOUT	A timeout occurred before the DSP responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EWRONGSTATE	The node is not in the Allocated, Paused, or Running state.
DSP_EFAIL	Unable to change node's runtime priority level.

Comments

A node's runtime priority level within the DSP RTOS is statically defined in the node configuration database. **DSPNode_ChangePriority** is provided to allow an application to change a node's priority level at runtime as needed. For example, an application might need to change its operating mode, and give more priority to a particular signal processing function. Other signal processing nodes should remain running (i.e., they should not be paused), but at a lower priority level. **DSPNode_ChangePriority** can be used to decrease the priority of less critical nodes, or increase the priority of the more critical nodes.

For a node in the Running state, **DSPNode_ChangePriority** will block until the DSP responds to the request. For nodes in the Allocated or Paused state, no blocking will occur; when the node enters the Running state it will execute at the priority specified by the most recent **DSPNode_ChangePriority** call.

Note that **DSPNode_ChangePriority** changes the execution priority of a single node; the API client may need to call this function for a group of interconnected nodes to achieve the desired effect.

Use caution when using **DSPNode_ChangePriority** on nodes which utilize the DSKT2 scratch feature. Often, nodes which use this feature specify their priority when associating themselves with a scratch group. This technique easily satisfies the requirement that nodes in the same scratch group can't pre-empt each other. Remember that if you use this technique, and change a node's priority with **DSPNode_ChangePriority**, you may no longer be meeting that "no pre-emption" requirement.

This API is not supported for IVA1 nodes.

Note

DSPNode_ChangePriority should only be called for a node that is currently in the Allocated, Paused, or Running state.

DSPNode_ChangePriority is used to change the priority of task and XDAIS socket nodes. Calling **DSPNode_ChangePriority** for a device node will result in a return value of DSP_ENODETYPE.

See Also

DSPNode_GetAttr

2.1.8.4. DSPNode_Connect

```
DBAPI DSPNode_Connect( DSP_HNODE                    hNode,
                       UINT                         uStream,
                       DSP_HNODE                    hOtherNode,
                       UINT                         uOtherStream,
                       IN OPTIONAL DSP_STRMATTR *   pAttr
                     )
```

Description

Make a stream connection, either between two nodes on a DSP, or between a node on a DSP and the GPP.

Parameters

hNode	Handle of the first node to connect to the second "other" node. If this first node is on the DSP, this handle must be the node handle returned from a successful call to DSPNode_Allocate. Alternatively, if this is the GPP side of a GPP ⇒ DSP connection, then DSP_HGPPNODE should be specified as the node handle.
uStream	If the first node (hNode) is on the DSP, uStream is the output stream index on the node that is to be connected to the "other" node's input stream. This stream index value must be in the range of 0 to one less than the number of output streams. If hNode is specified as DSP_HGPPNODE then uStream is ignored.
hOtherNode	Handle of the "other" node. If this second node is on the DSP, this handle must be the node handle returned from a successful call to DSPNode_Allocate. Alternatively, if this is the GPP side of a DSP ⇒ GPP connection, then DSP_HGPPNODE should be specified as the node handle.
uOtherStream	If the second node (hOtherNode) is on the DSP, uOtherStream is the input stream index on the second node that is to be connected to the first node's output stream. This stream index value must be in the range of 0 to one less than the number of input streams. If hOtherNode is specified as DSP_HGPPNODE then uOtherStream is ignored.
pAttr	A pointer to a DSP_STRMATTR structure that defines attributes for the stream connecting the two nodes. If the value of this parameter is NULL, a default set of attributes will be used for the stream.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	A handle parameter (hNode or hOtherNode) is invalid.
DSP_EALREADYCONNECTED	One of the specified connections has already been made.
DSP_EMEMORY	GPP memory allocation failure.
DSP_EWRONGSTATE	A specified node (hNode or hOtherNode) is not in the NODE_ALLOCATED state.
DSP_EVALUE	A stream index parameter (uStream or uOtherStream) is invalid.
DSP_ENOMORECONNECTIONS	No more connections are allowed.
DSP_ESTRMMODE	The stream mode contained in the DSP_STRMATTR structure is invalid.
DSP_EFAIL	Unable to make connection.

Comments

DSPNode_Connect connects an output stream of one node to the input stream of another node. Since a node can have multiple input and output streams, **DSPNode_Connect** can be called multiple times for a node.

The *uStream* and *uOtherStream* indices are NOT stream handles (used by the **DSPStream_** functions), but numeric stream indices for each node. For example, *uStream* and *uOtherStream* might both be 0, indicating *hNode*'s first output stream is to be connected to *hOtherNode*'s first input stream.

DSPNode_Connect is used to define a stream connection: between two nodes on the DSP (between a task node and a device node, or between two task nodes); or between a node on the DSP, and the GPP. A (software) pipe device will be automatically inserted by **DSPNode_Connect** when a connection between two task nodes is requested. For a connection between two nodes on a DSP, there is no interaction with the GPP for that particular stream.

A task node can have any mix of connections for its input and output streams. For example, a task node might receive input on stream 0 from the GPP host, and input on stream 1 from a DSP peripheral device (such as a codec), and input on stream 2 from a pipe device to another DSP task. Similarly, the task node might produce output on stream 0 that goes to a DSP peripheral device, and output on stream 1 that goes to a GPP application.

This API is not supported for IVA1nodes.

Note

When connecting two nodes on a DSP, **DSPNode_Connect** should only be called to connect two nodes that are both in the NODE_ALLOCATED state on the same DSP. **DSPNode_Connect** returns an error if one of the specified nodes is already created on the DSP (by a call to **DSPNode_Create**).

Other than destroying the nodes (via **DSPNode_Delete**), there is no way to disconnect two nodes that are currently running on the DSP.

Once the **DSPNode_Connect** call has returned successfully, the resources reserved for the connection remain allocated until one of the nodes is deleted, or your GPP program terminates.

DSP_EFAIL will be returned if any of the following connections are attempted:

- device node to device node
- device node to GPP

- GPP to GPP
- two nodes on different DSPs

Please use **DSPNode_ConnectEx** if required to pass connection parameter to DSP side device node.

See Also

DSPNode_Allocate
DSPNode_ConnectEx

2.1.8.5. DSPNode_ConnectEx

```
DBAPI DSPNode_Connect( DSP_HNODE                    hNode,
                       UINT                         uStream,
                       DSP_HNODE                    hOtherNode,
                       UINT                         uOtherStream,
                       IN OPTIONAL DSP_STRMATTR *   pAttr,
                       IN OPTIONAL DSP_CBDATA *     pConnParam,
                     )
```

Description

Make a stream connection, either between two nodes on a DSP, or between a node on a DSP and the GPP. This API is similar to DSPNode_Connect except it additionally provides a new connection parameter (string) to pass to DSP side while conneting device and task nodes. All other functionality remains same as DSPNode_Connect.

Addtioanl Parameter to DSPNode_Connect

pConnParam	A pointer to a DSP_CBDATA structure that defines connection parameter for device nodes to pass to DSP side. If the value of this parameter is NULL, then this API behaves like DSPNode_Connect. The DSP_CBDATA strcture will have length of the string and the null terminated string. This can be extended in future to pass binary data.

See Also

DSPNode_Allocate
DSPNode_Connect

2.1.8.6. DSPNode_Create

DBAPI DSPNode_Create(DSP_HNODE *hNode*)

Description

Create a node in a pre-run (i.e., inactive) state on its target processor; for task and XDAIS socket nodes, execute (on the target processor) the node's create-phase function.

Parameters

hNode	Handle of the node to create on the target processor, as returned from a successful call to DSPNode_Allocate.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	The parameter hNode is an invalid handle.
DSP_ETASK	Unable to create the node's task or process on the target processor.
DSP_EMEMORY	Memory allocation failure on the target processor.
DSP_ERESOURCE	A requested resource is not available.
DSP_EMULINST	Multiple instances are not allowed.
DSP_ENOTFOUND	A specified entity was not found.
DSP_EOUTOFIO	An I/O resource is not available.
DSP_ESTREAM	Stream creation failure on the target processor.
DSP_ETIMEOUT	A timeout occurred before the target processor responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EUSER1-16	A node-specific failure occurred on the target processor.
DSP_EWRONGSTATE	The operation is invalid for the current node state.
DSP_EOVERLAYMEMORY	Cannot overlay node's create phase code or data because memory is in use by another node.
DSP_ENOTCONNECTED	Cannot create node because an input (output) stream index passed to DSPNode_Connect is larger than one less than the number of input (output) stream connections made. For example, an output stream index of 1 was passed to DSPNode_Connect, but the output stream index of 0 was never passed to DSPNode_Connect.
DSP_EFAIL	A failure occurred, unable to create node.

Comments

DSPNode_Create is used to take a node that has been allocated on the GPP (via **DSPNode_Allocate**), and possibly connected to other nodes (via **DSPNode_Connect**), and create it on the appropriate target processor.

For task and XDAIS socket nodes, calling **DSPNode_Create** will cause a command to be sent to the RM Server on the DSP, telling it to create the node. After establishing the node's environment, the RM Server will invoke the node's create-phase function. If the create-phase function is successful, **DSPNode_Create**. will return DSP_SOK. If a problem occurs, either a pre-defined error code (e.g., DSP_ESTREAM), or a node-specific error will be reported (e.g., DSP_EUSER2). If an error occurs such that the node's create-phase function cannot be initiated, an appropriate error code will be returned by the DSP to indicate the cause of the failure. For example, if the node's task in the DSP RTOS cannot be created, then DSP_ETASK will be reported from the DSP, indicating that the create-phase function was not initiated.

Calling **DSPNode_Create** for a device node will not result in a command being sent to the RM Server, because a device driver is actually initialized when the client task calls **STRM_create** to open the device. For a static system where a device driver's code is already resident on the target processor, there is no real need to call **DSPNode_Create** for device nodes. However, for dynamic systems that allow loading of a device's code to the DSP at runtime, calling **DSPNode_Create** for device nodes is essential, as this is the 'trigger' to load the device code to the DSP. An additional requirement for dynamic systems is that **DSPNode_Create** is called for all device nodes ***before*** other node types that use the device. For example, if a GPP application wants to create two device nodes (a source and a sink device) and a single task node on a dynamic system, **DSPNode_Create** should be called for the two device nodes first. This will allow the device code to be dynamically loaded to the DSP before the task node's create-phase function is run (which will want to open the devices).

When **DSPNode_Create** indicates a failure during DSP code execution (e.g., DSP_ETASK, DSP_EMEMORY, DSP_ESTREAM, etc.), your program should call **DSPNode_Delete** for the node, to cleanup any partial allocations that may have succeeded in the previous (unsuccessful) call to **DSPNode_Create**.

The **DSPNode_Create** function will block until either the DSP responds to the creation request, or a timeout occurs.

Note

DSPNode_Create must only be called on a node that has already been allocated, and had its streams connected to their runtime counterparts.

On platforms that dynamically load device driver code at runtime, **DSPNode_Create** must be called to create a device node *before* **DSPNode_Create** is called to create the device's clients on the DSP.

For IVA1 nodes, the GPP side device driver is informed of the node state change. There will not be any messages sent to IVA1 when **DSPNode_Create** is called.

See Also

DSPNode_Allocate
DSPNode_Connect

2.1.8.7. DSPNode_Delete

DBAPI DSPNode_Delete(DSP_HNODE *hNode*)

Description

For task and XDAIS socket nodes only, first run the node's delete-phase function; then, for all node types, free all target processor side and GPP side resources for the node.

Parameters

hNode	Handle of the node to be deleted, as returned from a successful call to DSPNode_Allocate.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_EDELETE	A deletion failure occurred.
DSP_EFREE	A target processor memory free operation failed.
DSP_EIOFREE	A target processor I/O free operation failed.
DSP_ETIMEOUT	A timeout occurred before the target processor responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EUSER1-16	A node-specific failure occurred on the target processor.
DSP_EOVERLAYMEMORY	Can't overlay node's delete phase code or data because memory is in use by another node.
DSP_EFAIL	A failure occurred, unable to delete node.

Comments

DSPNode_Delete is used to delete a node, and free all its GPP and target processor resources. **DSPNode_Delete** is typically called to delete a node that has exited its execute-phase, either by design, or because **DSPNode_Terminate** was called by the GPP-client to shutdown the node.

When **DSPNode_Delete** is called for a task or XDAIS socket node that has been created on the DSP, a command will be sent to the RM Server on the DSP to: run the node's delete-phase function, (which should free all resources allocated by the node's create-phase function); and then clean up all (RM Server-created) default DSP-side resources for the node. When the RM Server responds back to the GPP, **DSPNode_Delete** will then free all GPP-side resources for the node. **DSPNode_Delete** will block until the node's delete-phase function has run, and all DSP and GPP-side resources for the node have been freed.

DSPNode_Delete can be called for a node that is in the Allocated state. In this situation, all GPP-side resources for the node will be freed, and no interaction with the target processor will occur.

After calling **DSPNode_Delete**, the node handle is invalid and cannot be used with any other functions. Therefore, DSPNode_Delete must only be called once for a given node handle.

To shut down a running task or XDAIS socket node, **DSPNode_Terminate** should be called before **DSPNode_Delete**, to allow the node a chance to gracefully exit its execute phase.

See Also

DSPNode_Terminate

2.1.8.8. DSPNode_FreeMsgBuf

```
DBAPI DSPNode_FreeMsgBuf( DSP_HNODE                    hNode,
                          IN BYTE *                    pBuffer,
                          IN OPTIONAL DSP_BUFFERATTR * pAttr
                        )
```

Description

Free a message buffer previously allocated by **DSPNode_AllocMsgBuf**.

Parameters

hNode	Handle of the node, as returned from a successful call to DSPNode_Allocate.
pBuffer	Address of the buffer, as returned from a previous call to DSPNode_AllocMsgBuf.
pAttr	Pointer to a DSP_BUFFERATTR structure. A NULL pointer indicates default attributes.

Return Value

DSP_SOK	Success.
DSP_EPOINTER	pBuffer is invalid.
DSP_EBADSEGID	The segment identifier specified via pAttr does not exist.
DSP_EFAIL	Failure to free the data buffer.

Comments

DSPNode_FreeMsgBuf unprepares and frees a data buffer that was previously allocated and prepared by **DSPNode_AllocMsgBuf**.

If, when the buffer was allocated with **DSPNode_AllocMsgBuf**, a non-NULL *pAttr* structure was specified, the same attributes must be passed to **DSPNode_FreeMsgBuf** as the buffer is freed. If *pAttr* was NULL when the buffer was allocated, it can be specified as NULL for **DSPNode_FreeMsgBuf**.

Note

DSPNode_FreeMsgBuf should only be called to free a buffer that was successfully allocated via **DSPNode_AllocMsgBuf**.

See Also

DSPNode_AllocMsgBuf

2.1.8.9. DSPNode_GetAttr

```
DBAPI DSPNode_GetAttr( DSP_HNODE              hNode,
                       OUT DSP_NODEATTR *     pAttr,
                       UINT                   uAttrSize
                     )
```

Description

Copy the current attributes of the specified node into a designated DSP_NODEATTR structure.

Parameters

hNode	Handle of the node whose attributes are to be reported, as returned from a successful call to DSPNode_Allocate.
pAttr	Pointer to an empty DSP_NODEATTR structure.
uAttrSize	Size of the DSP_NODEATTR structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_EPOINTER	Parameter pAttr is invalid.
DSP_EFAIL	Unable to retrieve node attributes.
DSP_ESIZE	The size of the specified DSP_NODEATTR structure is too small to hold all node information.

Comments

DSPNode_GetAttr is used to get information about a node that is currently allocated in the system. Alternatively, **DSPManager_EnumNodeInfo** can be used to get static information about a node (from the node configuration database), without allocating the node first.

See Also

DSPNode_ChangePriority
DSPManager_EnumNodeInfo
DSP_NODEATTR

2.1.8.10. DSPNode_GetMessage

```
DBAPI DSPNode_GetMessage( DSP_HNODE      hNode,
                          OUT DSP_MSG *  pMessage,
                          UINT           uTimeout
                        )
```

Description

Retrieve a message from a node.

Parameters

hNode	Handle of node to retrieve the message from, as returned by a successful call to DSPNode_Allocate.
pMessage	Pointer to location into which the message is to be copied.
uTimeout	Timeout value in milliseconds.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_EPOINTER	Parameter pMessage is invalid.
DSP_ETRANSLATE	The shared memory buffer contained in the message could not be mapped to the GPP client process' virtual space.
DSP_ENODETYPE	Messages cannot be retrieved from this type of node.
DSP_ETIMEOUT	A timeout occurred, and there is no message available from the node.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EFAIL	An error occurred while trying to retrieve a message.

Comments

DSPNode_GetMessage retrieves a message from the specified node on a target processor. If a message is not available, the function will block until one becomes available, or until the specified timeout value is reached.

The return value of **DSPNode_GetMessage** indicates whether or not a message was received. If DSP_SOK is returned a message was received, otherwise there was an error or a timeout.

Note

DSPNode_GetMessage must only be called for Message, Task, or XDAIS Socket nodes.

See Also

DSPNode_Allocate
DSPNode_AllocMsgBuf
DSPNode_FreeMsgBuf
DSPNode_PutMessage

2.1.8.11. DSPNode_Pause

DBAPI DSPNode_Pause(DSP_HNODE *hNode*)

Description

Temporarily suspend execution of a task node that is currently running on a target processor.

Parameters

hNode	Handle of the node to pause on the target processor, as returned from a successful call to DSPNode_Allocate.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	The parameter hNode is an invalid handle.
DSP_ENODETYPE	The specified node cannot be paused.
DSP_ETIMEOUT	A timeout occurred before the target processor responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EWRONGSTATE	The operation is invalid for the current node state.
DSP_EFAIL	Failure to transition node to Paused state.

Comments

DSPNode_Pause is used to temporarily suspend execution of a task or XDAIS socket node on a target processor, putting it in the **Paused** state; node execution is resumed by calling **DSPNode_Run**. **DSPNode_Pause** does not actually save any of the node's context, but rather simply lowers the priority of the DSP thread so it no longer runs. This function is provided as an alternative to **DSPNode_Delete**, allowing a GPP application the ability to temporarily reduce processor loading, or to temporarily suspend target processor tasks for application-specific purposes.

The **DSPNode_Pause** function will block until the target processor responds to the pause request.

Note

DSPNode_Pause should only be called on a node that is currently in the **Running** state.

Calling **DSPNode_Run** to resume a node's execution will not cause the node's create-phase function to be run again. The node will continue execution from the state it was in when **DSPNode_Pause** was called.

When **DSPNode_Pause** is called to suspend a node, the node's streams are paused also. The streams are not flushed, nor are the stream buffers freed. When node execution is resumed data streaming is resumed also.

See Also

DSPNode_Delete
DSPNode_Run

2.1.8.12. DSPNode_PutMessage

```
DBAPI DSPNode_PutMessage( DSP_HNODE            hNode,
                          IN CONST DSP_MSG *   pMessage,
                          UINT                 uTimeout
                        )
```

Description

Send a message to a task or XDAIS socket node.

Parameters

hNode	Handle of node to receive the message, as returned by a successful call to DSPNode_Allocate.
pMessage	Pointer to the message to be sent to the node.
uTimeout	Timeout value in milliseconds.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_EPOINTER	Parameter pMessage is invalid.
DSP_ETRANSLATE	The shared memory buffer contained in the message could not be mapped to the GPP client process' virtual space.
DSP_ENODETYPE	A message cannot be sent to this type of node.
DSP_EWRONGSTATE	The node is in an invalid state for receiving messages.
DSP_ETIMEOUT	A timeout occurred.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EFAIL	A failure occurred, unable to send the message.

Comments

DSPNode_PutMessage attempts to queue a message to the node. The function blocks until the message is queued, or until a node timeout occurs.

The return value of **DSPNode_PutMessage** is DSP_SOK if the message was successfully queued. Note that the message is copied out of the message structure referenced by **pMessage**, so the structure can be re-used immediately after returning from **DSPNode_PutMessage**.

See Also

DSPNode_Allocate
DSPNode_AllocMsgBuf
DSPNode_FreeMsgBuf
DSPNode_GetMessage

2.1.8.13. DSPNode_RegisterNotify

```
DBAPI DSPNode_RegisterNotify( DSP_HNODE         hNode,
                              UINT              uEventMask,
                              UINT              uNotifyType,
                              DSP_HNOTIFICATION hNotification
                            )
```

Description

Register to be notified of specific events for this node.

Parameters

hNode	Handle of the node, as returned from a successful call to DSPNode_Allocate.
uEventMask	Maksk of types of events to be notified about:
DSP_NODESTATECHANGE	The node's state has changed.
DSP_NODEMESSAGEREADY	A message from the node has arrived on the GPP.
uNotifyType	Type of notification to be sent:
DSP_SIGNALEVENT	Signal the event specified by hNotification.
hNotification	Handle of a DSP_NOTIFICATION object.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode or hNotification is invalid.
DSP_EVALUE	Parameter uEventMaskis invalid.
DSP_ENOTIMPL	The notification type specified in uNotifyType is invalid, or not supported.
DSP_EFAIL	Unable to register for notification.

Comments

DSPNode_RegisterNotify allows API clients to register for notification of certain types of events that occur for a specific node. For example, an application can register to be notified when a message has arrived from the node.

Note that **DSPNode_RegisterNotify** is used for registering for a notification for a **specific** node, whereas **DSPProcessor_RegisterNotify** can be used to register for events from **all** nodes allocated on a target processor.

The types of events to be notified about are defined by *uEventMask*. The type of notification to be sent is defined by *uNotifyType*, currently, the only notification type available is DSP_SIGNALEVENT.

hNotification is a handle to a DSP_NOTIFICATION object that defines the notification object name or handle to be signaled.

For example, if a GPP application wants to be notified when the node's state changes, it can call **DSPNode_RegisterNotify** with: *hNode* set to the node's handle, *uEventMask* set to DSP_NODESTATECHANGE; *uNotifyType* set to DSP_SIGNALEVENT; and *hNotification* pointing to a notification object that contains the event to be signaled.

A client can de-register for notifications by calling **DSPNode_RegisterNotify**, specifying the same notification object handle that was used in a previous **DSPNode_RegisterNotify** call, with *uEventMask* now set to zero. De-registering in this manner will not flush any pending notifications, so the client must be able to handle notifications that might already be pending.

Note

Named events are not supported.

See Also

DSPProcessor_RegisterNotify
DSPStream_RegisterNotify
DSP_NOTIFICATION

2.1.8.14. DSPNode_Run

DBAPI DSPNode_Run(DSP_HNODE *hNode*)

Description

Start a node running, or resume execution of a previously paused node.

Parameters

hNode	Handle of the node to start or resume, as returned from a call to DSPNode_Allocate.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_ENODETYPE	DSPNode_Run cannot be called for this type of node.
DSP_ETIMEOUT	A timeout occurred before the target processor responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EWRONGSTATE	The node is not in the Created or Paused state.
DSP_EOVERLAYMEMORY	Can't overlay node's execute phase code or data because memory is in use by another node.
DSP_EFAIL	Unable to start or resume execution.

Comments

The return value of **DSPNode_Run** indicates whether or not the request to start or resume the node on the target processor was successful.

DSPNode_Run should only be called for task or XDAIS socket nodes.

See Also

DSPNode_GetAttr
DSPNode_Pause

2.1.8.15. DSPNode_Terminate

DBAPI DSPNode_Terminate(DSP_HNODE *hNode*,
DSP_STATUS * *pStatus*
)

Description

Signal a task or XDAIS socket node running on a target processor that it should exit its execute-phase function.

Parameters

hNode	Handle of the node that is to terminate, as returned from a successful call to DSPNode_Allocate.
pStatus	Location where the target processor node's exit code is to be returned. A value of NULL for this argument is an error.

Return Value

DSP_SOK	Success.
DSP_ENODETYPE	The type of node specified cannot be terminated.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_ETIMEOUT	A timeout occurred before the target processor responded.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EWRONGSTATE	The operation is invalid for the current node state.
DSP_EPOINTER	Input pointer pStatus is invalid.
DSP_EFAIL	Unable to terminate the node.

Comments

This function will cause a task or XDAIS socket node to be terminated on the target processor on which it is running. On a successful return from **DSPNode_Terminate**, the status value returned by the node's execute-phase function will be returned at the location pointed to by *pStatus*, and the state of the node will be NODE_DONE.

After calling **DSPNode_Terminate**, your program must additionally call **DSPNode_Delete** in order to de-allocate the resources used by the node.

The **DSPNode_Terminate** function will block until either the node responds to the termination request, or a timeout occurs.

Note

DSPNode_Terminate should only be called on a node that is currently in the NODE_RUNNING state.

See Also

DSPNode_Allocate
DSPNode_Create
DSPNode_Delete
DSPNode_Run

2.1.9. DSPStream Interface

Include File

```
#include <DSPStream.h>
```

Table 5 Stream interface functions

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

DSPStream Interface functions are not supported for the IVA processor.

The **`DSPStream_`** functions support direct communication with individual nodes on a DSP processor. Each **`DSPStream_`** function call requires a stream handle, which identifies the individual stream. This stream handle is returned by a successful invocation of the **`DSPStream_Open`** function.

2.1.9.1. DSPStream_AllocateBuffers

```
DBAPI DSPStream_AllocateBuffers( DSP_HSTREAM hStream,
                                 UINT        uSize,
                                 OUT BYTE ** apBuffer,
                                 UINT        uNumBufs
                               )
```

Description

Allocate data buffers for use with a specific stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
uSize	Size (in GPP bytes) of the buffer(s) to be allocated.
apBuffer	Array to receive the addresses of the allocated buffers.
uNumBufs	The number of buffers to be allocated.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hStream handle.
DSP_EMEMORY	Insufficient memory.
DSP_EPOINTER	apBuffer is not a valid address.
DSP_EFAIL	A failure occurred, unable to allocate buffer.

Comments

DSPStream_AllocateBuffers allocates and prepares data buffers for use with a specific stream. Buffers returned by **DSPStream_AllocateBuffers** are issued to a stream using **DSPStream_Issue**, and reclaimed from a stream using **DSPStream_Reclaim**.

The function parameter *uSize* defines the size of the buffer(s) to be allocated; the buffer memory segment and the alignment for the allocation(s) were specified when the stream was openned (see **DSPStream_Open**).

DSPStream_AllocateBuffers prepares buffers for use with the specific stream. For example, the buffer may be page locked, and allocated in a memory region that is appropriate for the underlying “link” driver to the DSP.

Buffers allocated with **DSPStream_AllocateBuffers** must be freed by the client application via calls to **DSPStream_FreeBuffers**.

See Also

DSPStream_Issue
DSPStream_Reclaim
DSPStream_FreeBuffers
DSPStream_Open

2.1.9.2. DSPStream_Close

DBAPI DSPStream_Close(DSP_HSTREAM *hStream*)

Description

Close a stream and free the underlying stream object.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hStream handle.
DSP_EPENDING	All data buffers issued to the stream have not been reclaimed from the stream yet.
DSP_EFAIL	Failure to close the stream.

Comments

`DSPStream_Close` is called to shutdown a stream and free the GPP-side resources allocated for the stream.

Note

`DSPStream_Close` should only be called when all stream data buffers previously submitted to the stream (via **`DSPStream_Issue`**), have been retrieved from the stream (using **`DSPStream_Reclaim`**).

See Also

DSPStream_GetInfo
DSPStream_Issue
DSPStream_Open
DSPStream_Reclaim

2.1.9.3. DSPStream_FreeBuffers

DBAPI DSPStream_FreeBuffers(DSP_HSTREAM *hStream*,
IN BYTE ** *apBuffer*,
UINT *uNumBufs*
)

Description

Free previously allocated stream data buffers.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
apBuffer	Array of buffers, as returned from a previous call to DSPStream_AllocateBuffers.
uNumBufs	The number of buffers to be freed.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hStream handle.
DSP_EPOINTER	apBuffer is invalid.
DSP_EFAIL	Failure to free the data buffer(s).

Comments

`DSPStream_FreeBuffers` unprepares and releases stream data buffers that were previously allocated and prepared for the stream via a call to **`DSPStream_AllocateBuffers`**. *`apBuffer`* points to the first element of an array of pointers to buffers that are to be freed. The number of buffers to be freed is specified by *`uNumBufs`*. Note that the buffer pointer array that *`apBuffer`* points to is **not** freed by **`DSPStream_FreeBuffers`**; this array must be freed by the DSP/BIOS Bridge API client that allocated it.

Note

`DSPStream_FreeBuffers` should only be called to free a buffer that was successfully allocated via **`DSPStream_AllocateBuffers`**.

See Also

DSPStream_AllocateBuffers

2.1.9.4. DSPStream_GetInfo

```
DBAPI DSPStream_GetInfo( DSP_HSTREAM              hStream,
                         OUT DSP_STREAMINFO *     pStreamInfo,
                         UINT                     uStreamInfoSize
                       )
```

Description

Get information about a stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
pStreamInfo	Pointer to the DSP_STREAMINFO structure in which the stream information will be returned.
uStreamInfoSize	Size of the DSP_STREAMINFO structure.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hStream is invalid.
DSP_EPOINTER	Parameter pStreamInfo is not valid.
DSP_EFAIL	Unable to retrieve stream info.
DSP_ESIZE	The size of the specified DSP_STREAMINFO structure is too small to hold all stream information.

Comments

DSPStream_GetInfo allows an API client to get information about a stream, including the number of bytes transferred on the stream since the stream was last reset, and a handle to the stream's underlying GPP OS synchronization object. **DSPStream_GetInfo** returns information in a DSP_STREAMINFO structure.

DSP_STREAMINFO includes the number of bytes transferred since the stream was most recently reset. The API resets the stored value when **DSPStream_Idle** is called, so the number returned is either the total count since the stream was created, or the count since the stream was last reset.

DSPStream_Select allows a client to block until one or more streams are ready for I/O, but the function is limited to blocking on stream object handles. Some applications may want to block execution until a stream is ready, *or* some other type of synchronization object is signaled. **DSPStream_GetInfo** allows a client to get a handle to the underlying GPP OS synchronization object for a specific stream. The client can then wait on this object in combination with other types of synchronization objects, (i.e., not just other stream objects). For example, an application might want to wait for data on multiple streams, or the occurrence of other events in the system, such as the posting of a mutex by another application. Calling **DSPStream_GetInfo** allows the application to get the synchronization object handle for a stream, and include this handle in a call to **DSPManager_WaitForEvents**. See the Note section of **DSPStream_Select** for further clarification on how to use this feature, if supported.

Note

The DSP_STREAMINFO structure pointed to by pStreamInfo must be allocated before **`DSPStream_GetInfo`** is called.

See Also

DSPStream_Open
DSPStream_Idle
DSPStream_Select
DSP_STREAMINFO

2.1.9.5. DSPStream_Idle

DBAPI DSPStream_Idle(DSP_HSTREAM *hStream*,
BOOL *bFlush*
)

Description

Idle a stream, and (optionally) flush output data buffers.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
bFlush	Flag indicating if output stream data should be discarded.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hStream is invalid.
DSP_ETIMEOUT	A timeout occurred before the stream could be idled.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EFAIL	Unable to idle stream.

Comments

If the stream is an input stream, **DSPStream_Idle** resets the stream, and causes any currently buffered input data to be discarded.

If the stream is an output stream, operation depends on the value of the *bFlush* flag:

- If *bFlush* is FALSE, **DSPStream_Idle** causes any currently buffered data to be transferred through the stream. **DSPStream_Idle** will suspend program execution for the time specified as this GPP stream's *uTimeout* attribute (specified in **DSPStream_Open**).
- If *bFlush* is TRUE, **DSPStream_Idle** causes any currently buffered data to be discarded, without blocking.

After **DSPStream_Idle** is called, data buffers (that were enqueued by one or more calls to **DSPStream_Issue**) may be immediately reclaimed from the stream using **DSPStream_Reclaim**.

DSPStream_Idle returns DSP_SOK if the stream is successfully idled.

Note

DSPStream_Idle has the side effect of setting the number of bytes transferred for the stream back to zero. That is, after a call to **DSPStream_Idle**, the count retrieved by a **DSPStream_GetInfo** call is zero.

See Also

DSPStream_Issue
DSPStream_Reclaim
DSPStream_GetInfo

2.1.9.6. DSPStream_Issue

DBAPI DSPStream_Issue(DSP_HSTREAM *hStream*,
IN BYTE * *pBuffer*,
ULONG *dwDataSize*,
ULONG *dwBufSize*,
IN DWORD *dwArg*
)

Description

Send a buffer of data to a stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
pBuffer	Pointer to buffer of data to be sent to the stream.
dwDataSize	Number of actual data bytes in the buffer.
dwBufSize	The actual (allocated) size of the buffer.
dwArg	A user defined buffer context.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hStream is invalid.
DSP_EPOINTER	Parameter pBuffer is invalid.
DSP_ESTREAMFULL	The stream has been issued the maximum number of buffers allowed in the stream at once; buffers must be reclaimed from the stream before any more can be issued.
DSP_ETRANSLATE	A shared memory buffer contained in the stream could not be mapped to the GPP client process' virtual space.
DSP_EFAIL	A failure occurred, unable to issue buffer.

Comments

`DSPStream_Issue` is used to send a data buffer to a stream. The data transfer is accomplished by adding a buffer to the queue of buffers for the stream. **`DSPStream_Issue`** does not block; it returns without indicating the success of the transfer process, only the success of enqueuing the buffer (that is, the data transfer process operates asynchronously to your program).

The interpretation of *`dwDataSize`*, the logical size of a buffer, is direction-dependent. For an output stream, the logical size of the buffer indicates the number of valid bytes of data it contains. For an input stream, the logical length of a buffer indicates the number of bytes being requested by the client. In either case, the logical size of the buffer must be less than or equal to the physical size of the buffer.

Issuing a zero-length buffer (i.e., *`dwDataSize`* = 0) to an output stream will signal an end-of-stream to the recipient, and will cause the stream to transition to the `NODE_DONE` state. Likewise, receiving a zero-length buffer from an input stream is a signal of an end-of-stream condition.

The argument *`dwArg`* is not interpreted by **`DSPStream_Issue`** or **`DSPStream_Reclaim`**, but is offered as a service to the stream client, allowing for association of additional information with a particular buffer of data.

DSPStream_Issue is used in conjunction with **DSPStream_Reclaim**. The **DSPStream_Issue** call sends a buffer to a stream, and **DSPStream_Reclaim** retrieves a buffer from a stream. In normal operation each **DSPStream_Issue** call is followed by a **DSPStream_Reclaim** call. Short bursts of multiple **DSPStream_Issue** calls can be made without an intervening **DSPStream_Reclaim** call, but over the life of the stream **DSPStream_Issue** and **DSPStream_Reclaim** must be called the same number of times.

Failure of **DSPStream_Issue** indicates that the stream was not able to accept the buffer being issued or that there was an underlying device error.

Note

Buffers submitted to a stream should be allocated and prepared via the **DSPStream_AllocateBuffers** function. If the API client receives buffers (e.g., from another application), these buffers can also be issued to the stream, as long as the have been properly prepared for the stream, using **DSPStream_PrepareBuffer**.

A **DSPStream_Reclaim** call should not be made without at least one outstanding **DSPStream_Issue** call. Calling **DSPStream_Reclaim** with no outstanding **DSPStream_Issue** calls has undefined results.

See Also

DSPStream_AllocateBuffers
DSPStream_Reclaim
DSPStream_Idle

2.1.9.7. DSPStream_Open

```
DBAPI DSPStream_Open( DSP_HNODE                        hNode,
                      UINT                             uDirection,
                      UINT                             uIndex,
                      IN OPTIONAL DSP_STREAMATTRIN *   pAttrIn,
                      OUT DSP_HSTREAM *                phStream
                    )
```

Description

Retrieve a stream handle for sending/receiving data buffers to/from a node on a DSP.

Parameters

hNode	Handle of the node, as returned by a successful call to DSPNode_Allocate.
uDirection	Stream direction: DSP_TONODE for an output stream from the GPP to the node; or DSP_FROMNODE for an input stream from the node to the GPP.
uIndex	Stream index. This value must be in the range from 0 to less than the number of input/output streams for a node.
pAttrIn	Pointer to the DSP_STREAMATTRIN structure that contains the attributes to be applied to the stream. If the value of this parameter is NULL, a default set of attributes will be assigned to the stream.
phStream	Location where the stream handle is to be returned. A value of NULL for this argument is an error.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hNode is invalid.
DSP_EDIRECTION	Parameter uDirection is invalid.
DSP_EVALUE	Either parameter uIndex is invalid, or one of the attributes specified by pAttrIn is invalid.
DSP_EPOINTER	Parameter phStream is invalid.
DSP_ENODETYPE	A stream cannot be opened to this type of node.
DSP_ESTRMMODE	The stream mode contained in the DSP_STRMATTR structure is invalid.
DSP_EFAIL	Unable to open stream.
DSP_EMEMORY	There was insufficient GPP memory to allocate a new DSPStream object.
DSP_EINVALIDARG	One or more of the fields in the DSP_STREAMATTRIN structure are invalid.

Comments

`DSPStream_Open` allocates and returns a handle for a stream object, which represents a logical communication channel between the GPP and a node on a DSP. The parameters for **`DSPStream_Open`** are the handle of the node, the direction of data flow, and a stream index identifying a specific stream.

The stream handle can be subsequently used with **`DSPStream_Issue`**, **`DSPStream_Reclaim`**, **`DSPStream_Select`**, and other **`DSPStream_`** functions.

Note

DSPStream_Open must only be called *after* all of the corresponding node's connections have been specified via **DSPNode_Connect** calls. In other words, a node should be allocated, and its streams connected, before your GPP application attempts to open a stream to the node.

See Also

DSP_STREAMATTRIN
DSPStream_Close
DSPStream_Issue
DSPStream_Reclaim
DSPStream_Select

2.1.9.8. DSPStream_PrepareBuffer

```
DBAPI DSPStream_PrepareBuffer( DSP_HSTREAM hStream,
                               UINT        uSize,
                               BYTE *      pBuffer
                             )
```

Description

Prepare a buffer that was not allocated by **DSPStream_AllocateBuffers** for use with a stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
uSize	Size (in GPP bytes) of the allocated buffer.
pBuffer	Address of the buffer.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hStream handle.
DSP_EPOINTER	pBuffer is not a valid address.
DSP_EFAIL	A failure occurred, unable to prepare buffer.

Comments

DSPStream_PrepareBuffer prepares a data buffer that was not allocated with **DSPStream_AllocateBuffers**, for use with a specific stream. For example, a buffer may need to be page locked before it is submitted to a stream. **DSPStream_PrepareBuffer** will perform the appropriate preparation needed for the given platform.

In general, **DSPStream_AllocateBuffers** should be used to allocate and prepare all stream buffers. However, some applications may be passed pre-allocated buffers, which need to be fed to a stream. Before these pre-allocated buffers can be passed to the stream, they must be prepared with **DSPStream_PrepareBuffer**.

See Also

DSPStream_AllocateBuffers
DSPStream_UnprepareBuffer

2.1.9.9. DSPStream_Reclaim

```
DBAPI DSPStream_Reclaim( DSP_HSTREAM hStream,
                         OUT BYTE **    pBufPtr,
                         OUT ULONG *    pDataSize,
                         OUT ULONG *    pBufSize,
                         OUT DWORD *    pdwArg
                       )
```

Descriptions

Request a buffer back from a stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
ppBufPtr	Location where the pointer to the reclaimed data buffer should be written.
pDataSize	Location where the number of actual data bytes in the buffer should be written.
pBufSize	Location where the actual (allocated) size of the buffer should be written. This is typically used when a GPP ↔ DSP stream is opened for STRMMODE_ZEROCOPY, where reclaimed buffers can be of different sizes.
pdwArg	Location where the user argument that travels with the buffer should be written.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hStream is invalid.
DSP_EPOINTER	Parameter pBufPtr, pDataSize, or pdwArg is invalid.
DSP_ETIMEOUT	A timeout occurred before a buffer could be retrieved.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_ETRANSLATE	A shared memory buffer contained in the stream could not be mapped to the GPP client process' virtual space.
DSP_EFAIL	A failure occurred, unable to reclaim a buffer.

Comments

DSPStream_Reclaim is used to request a data buffer back from a stream. On success, it returns a pointer to the buffer, the number of valid bytes in the buffer, the size of the buffer, and a user argument (*pdwArg* points to the same value that was passed into the stream with this buffer, via a **DSPStream_Issue** call).

For platforms that incorporate a data copy when transferring buffers between the GPP and DSP, **DSPStream_Reclaim** only returns buffers that were passed into the stream using **DSPStream_Issue**, and it returns the buffers in the same order that they were issued to the stream. That is, the ordering of buffers issued and reclaimed from the stream follows the pattern of a circular queue.

For platforms that support a "zero-copy" buffer exchange mechanism, (where only the buffer descriptors are copied, not the buffer data contents), **DSPStream_Reclaim** may return a pointer to a buffer that was actually allocated by the DSP. In this case the GPP will receive buffers it didn't issue to the stream, but the ordering of buffers received by the GPP will correspond to the order the buffers were issued to the stream on the DSP side. The contents of *pBufSize* will contain the actual buffer size when originally allocated by either the DSP or GPP.

If the stream is an output stream, then **DSPStream_Reclaim** will return an empty buffer, and the data size will be zero, since the buffer is empty. If the stream is an input stream, **DSPStream_Reclaim** will return a non-empty buffer, and the data size will be the number of valid bytes of data in the buffer. In either mode **DSPStream_Reclaim** will block until a buffer can be returned to the caller, or until the timeout expires.

A timeout or failure of **DSPStream_Reclaim** indicates that no buffer was returned to the client. The stream timeout is specified in the node configuration database, and can be overridden by a DSP_STREAMATTRIN structure when **DSPStream_Open** is called. If **DSPStream_Reclaim** fails, the client should not attempt to de-reference *pBufPtr*, since it is not guaranteed to contain a valid buffer pointer.

DSPStream_Reclaim is used in conjunction with **DSPStream_Issue**. The **DSPStream_Issue** call sends a buffer to a stream, and **DSPStream_Reclaim** retrieves a buffer from a stream. In normal operation each **DSPStream_Issue** call is followed by a **DSPStream_Reclaim** call. Short bursts of multiple **DSPStream_Issue** calls can be made without an intervening **DSPStream_Reclaim** call, but over the life of the stream **DSPStream_Issue** and **DSPStream_Reclaim** must be called the same number of times.

Note

A **DSPStream_Reclaim** call should not be made without at least one outstanding **DSPStream_Issue** call. Calling **DSPStream_Reclaim** with no outstanding **DSPStream_Issue** calls has undefined results.

All buffers issued to a stream must be reclaimed before closing the stream.

See Also

DSPStream_AllocateBuffers
DSPStream_Issue

2.1.9.10. DSPStream_RegisterNotify

```
DBAPI DSPStream_RegisterNotify( DSP_HSTREAM        hStream,
                                UINT               uEventMask,
                                UINT               uNotifyType,
                                DSP_HNOTIFICATION  hNotification
                              )
```

Descriptions

Register to be notified of specific events for this stream.

Parameters

hStream	Handle of the node, as returned from a successful call to DSPStream_Open.
uEventMask	Mask of types of events to be notified about:
DSP_STREAMDONE	The stream has entered the DONE state.
DSP_STREAMIOCOMPLETION	A stream I/O operation has completed.
uNotifyType	Type of notification to be sent:
DSP_SIGNALEVENT	Signal the event specified by hNotification.
hNotification	Handle of a DSP_NOTIFICATION object.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Parameter hStream or hNotification is invalid.
DSP_EVALUE	Parameter uEventMask is invalid.
DSP_ENOTIMPL	The notification type specified in uNotifyType is not supported.
DSP_EFAIL	Unable to register for notification.

Comments

`DSPStream_RegisterNotify` allows API clients to register for notification of certain types of events that occur for a specific stream. For example, an application can register to be notified when an I/O operation completes on a stream.

Note that **`DSPStream_RegisterNotify`** is used for registering for a notification for a **specific** stream, whereas **`DSPProcessor_RegisterNotify`** can be used to register for events from **all** streams allocated on a DSP processor.

The types of events to be notified about are defined by *`uEventMask`*.

The type of notification to be sent is defined by *`uNotifyType`*. Currently only `DSP_SIGNALEVENT` is supported.

`hNotification` is a handle to a `DSP_NOTIFICATION` object that defines the notification object name or handle.

For example, if a GPP application wants to be notified when a stream I/O operation completes, it can create an event, and call **`DSPStream_RegisterNotify`** with: *`hStream`* set to the stream handle, *`uEventMask`* set to `DSP_STREAMIOCOMPLETION`; *`uNotifyType`* set to `DSP_SIGNALEVENT`; and *`hNotification`* pointing to a `DSP_NOTIFICATION` structure that contains the event to be signaled.

A client can de-register for notifications by calling **DSPStream_RegisterNotify**, specifying the same notification object handle that was used in a previous **DSPStream_RegisterNotify** call, with *uEventMask* now set to zero. De-registering in this manner will not flush any pending notifications, so the client must be able to handle notifications that might already be pending.

Note

Named events are not supported in Linux.

See Also

DSP_NOTIFICATION
DSPProcessor_RegisterNotify
DSPManager_WaitForEvents
DSPNode_RegisterNotify

2.1.9.11. DSPStream_Select

```
DBAPI DSPStream_Select( IN DSP_HSTREAM * aStreamTab,
                        UINT             nStreams,
                        OUT UINT *       pMask,
                        UINT             uTimeout
                      )
```

Description

Select a ready stream.

Parameters

aStreamTab	Array of stream handles (each handle acquired via a call to DSPStream_Open).
nStreams	Number of stream handles in the aStreamTab array.
pMask	Pointer to the location to receive the mask of ready streams.
uTimeout	Timeout value in milliseconds.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Handle pointed by a StreamTab is not valid.
DSP_EMEMORY	Unable to allocate synchronization object array.
DSP_ERANGE	Parameter nStreams is out of range.
DSP_EPOINTER	Parameter pMask is invalid.
DSP_ETIMEOUT	A timeout occurred before a stream became ready.
DSP_ERESTART	A critical error has occurred and the processor is being restarted.
DSP_EFAIL	An error occurred, failed to select a stream.

Comments

DSPStream_Select waits until one or more of the streams in the *aStreamTab* array is ready for I/O. That is, a subsequent I/O operation on the ready stream will not block.

nStreams indicates the number of stream handles in the *aStreamTab* array. The maximum value allowed for *nStreams* corresponds to the GPP word size, i.e., the number of streams that can be represented by the word pointed to by *pMask*. The *uTimeout* parameter indicates the number of milliseconds to wait before a stream becomes ready. If *uTimeout* is zero, **DSPStream_Select** will return immediately, indicating which (if any) streams are ready. If *uTimeout* is DSP_FOREVER, **DSPStream_Select** will block until one of the streams is ready.

On success (DSP_SOK), the word pointed to by *pMask* indicates which streams are ready for I/O. A **1** in bit position **j** indicates the stream *aStreamTab[j]* is ready.

Note

aStreamTab must contain handles returned from prior calls to **DSPStream_Open**.

See Also

DSPStream_Open

2.1.9.12. DSPStream_UnprepareBuffer

DBAPI DSPStream_UnprepareBuffer(DSP_HSTREAM *hStream*,
UINT *uSize*,
BYTE * *pBuffer*
)

Description

Unprepare a buffer that had been previously prepared for a stream by **DSPStream_PrepareBuffer**, and will no longer be used with the stream.

Parameters

hStream	Handle of the stream, as returned from a successful call to DSPStream_Open.
uSize	Size (in GPP bytes) of the buffer.
pBuffer	Address of the buffer.

Return Value

DSP_SOK	Success.
DSP_EHANDLE	Invalid hStream handle.
DSP_EPOINTER	pBuffer is not a valid address.
DSP_EFAIL	A failure occurred, unable to unprepare buffer or uSize is invalid (i.e = 0).

Comments

DSPStream_UnprepareBuffer is used to unprepare a buffer that was previously prepared for a stream via **DSPStream_PrepareBuffer**. For example, if **DSPStream_PrepareBuffer** page-locks a buffer prior to its use in a stream, **DSPStream_UnprepareBuffer** will unlock the buffer.

Note

DSPStream_UnprepareBuffer should only be called on a buffer that has been prepared for a stream via **DSPStream_PrepareBuffer**.

See Also

DSPStream_PrepareBuffer

2.1.10. Macros

2.1.10.1. DSP_SUCCEEDED

DSP_SUCCEEDED(DSP_STATUS *status*)

Description

The DSP_SUCCEEDED macro simplifies checking of GPP-side API function return codes for success. The macro returns TRUE if the DSP_STATUS code indicates success (i.e., the left-most bit is zero), otherwise the macro returns FALSE.

Example

```
lStatus = DSPProcessor_Attach(1, phProcessor);
if (DSP_SUCCEEDED(lStatus)) {
    ‘ attach successful, start node allocation …’
}
```

2.1.10.2. DSP_FAILED

DSP_FAILED(DSP_STATUS status)

Description

The DSP_FAILED macro simplifies checking of GPP-side API function return codes for failure. The macro returns TRUE if the DSP_STATUS code indicates failure (i.e., the left-most bit is one), otherwise the macro returns FALSE.

Example

```
lStatus = DSPNode_Delete(hNode);
if (DSP_FAILED(lStatus)) {
    ‘ report error to user’
}
```

2.1.11. GPP-side API Return Codes

Include File

```
#include <errbase.h>
```

Table 6 GPP-side API return codes

Return Code	Name	Usage
0x00008000	**DSP_SOK**	Success.
0x00008001	**DSP_SALREADYATTACHED**	Success; by the way, GPP is already attached to this target processor.
0x00008002	**DSP_SENUMCOMPLETE**	Success; this is the last object available for enumeration.
0x80008000	**DSP_EACCESSDENIED**	The caller does not have access privileges to call this function.
0x80008001	**DSP_EALREADYCONNECTED**	The specified connection already exists.
0x80008002	**DSP_EATTACHED**	The GPP must be fully detached from the target processor before this function is called.
0x80008003	**DSP_ECHANGEDURINGENUM**	During enumeration, a change in the number or properties of the objects has occurred.
0x80008004	**DSP_ECORRUPTFILE**	An error occurred while parsing the target processor executable file.
0x80008005	**DSP_EDELETE**	A failure occurred during a delete operation.
0x80008006	**DSP_EDIRECTION**	The specified direction is invalid.
0x80008007	**DSP_ESTREAMFULL**	A stream has been issued the maximum number of buffers allowed in the stream at once; buffers must be reclaimed from the stream before any more can be issued.
0x80008008	**DSP_EFAIL**	A general failure occurred.
0x80008009	**DSP_EFILE**	The specified executable file could not be found.
0x8000800A	**DSP_EHANDLE**	The specified handle is invalid.
0x8000800B	**DSP_EINVALIDARG**	An invalid argument was specified.
0x8000800C	**DSP_EMEMORY**	A memory allocation failure occurred.
0x8000800D	**DSP_ENODETYPE**	The requested operation is invalid for this node type.
0x8000800E	**DSP_ENOERRTEXT**	No error text was found for the specified error code.
0x8000800F	**DSP_ENOMORECONNECTIONS**	No more connections can be made for the node.
0x80008010	**DSP_ENOTIMPL**	The indicated operation is not supported.

Return Code	Name	Usage
0x80008011	**DSP_EPENDING**	I/O is currently pending, e.g., an attempt is made to close a stream before all buffers have been reclaimed.
0x80008012	**DSP_EPOINTER**	An invalid pointer was specified.
0x80008013	**DSP_ERANGE**	A parameter is specified outside its valid range.
0x80008014	**DSP_ESIZE**	An invalid size parameter was specified.
0x80008015	**DSP_ESTREAM**	A stream creation failure occurred on the DSP.
0x80008016	**DSP_ETASK**	A task creation failure occurred on the target processor.
0x80008017	**DSP_ETIMEOUT**	A timeout occurred before the requested operation could complete.
0x80008018	**DSP_ETRUNCATED**	A data truncation occurred.
0x8000801A	**DSP_EVALUE**	A parameter is invalid.
0x8000801B	**DSP_EWRONGSTATE**	The state of the specified object is incorrect for the requested operation.
0x8000801C	**DSP_ESYMBOL**	A symbol was not found in the DBOF file.
0x8000801D	**DSP_EUUID**	The specified UUID was not found.
0x8000801E	**DSP_EDCDREADSECT**	Unable to read contents of a DCD data section; typically caused by improperly configured nodes.
0x8000801F	**DSP_EDCDPARSESECT**	Unable to decode DCD data section content.
0x80008020	**DSP_EDCDGETSECT**	Unable to get pointer to DCD data section; typically caused by improperly configured UUIDs.
0x80008021	**DSP_EDCDLOADBASE**	Unable to load file containing DCD data section.
0x80008022	**DSP_EDCDNOAUTOREGISTER**	Unable to get pointer to DCD auto-register section.
0x80008028	**DSP_ERESOURCE**	A requested resource is not available.
0x80008029	**DSP_ERESTART**	A critical error has occurred, and the processor is being restarted.
0x8000802A	**DSP_EFREE**	A target processor memory free operation failed.
0x8000802B	**DSP_EIOFREE**	A target processor I/O free operation failed.
0x8000802C	**DSP_EMULINST**	Multiple instances are not allowed.
0x8000802D	**DSP_ENOTFOUND**	A specified entity was not found.
0x8000802E	**DSP_EOUTOFIO**	A target processor I/O resource is not available.
0x8000802F	**DSP_ETRANSLATE**	A shared memory buffer contained in a message or stream could not be mapped to the GPP client process's virtual space.

Return Code	Name	Usage
0x80008031	**DSP_EFWRITE**	File or section load write function failed to write to the target processor.
0x80008032	**DSP_ENOSECT**	Unable to find a named section in the DSP executable.
0x80008033	**DSP_EFOPEN**	Unable to open file.
0x80008034	**DSP_EFREAD**	Unable to read file.
0x80008037	**DSP_EOVERLAYMEMORY**	Unable to overlay node code or data because the memory is currently in use by another node.
0x80008038	**DSP_EBADSEGID**	A non-existent memory segment identifier was specified.
0x80008039	**DSP_EALIGNMENT**	The specified alignment value is not supported.
0x8000803A	**DSP_ESTRMMODE**	Not a valid stream mode.
0x8000803B	**DSP_ENOTCONNECTED**	Insufficient number of stream connections made before calling DSPNode_Create.
0x8000803C	**DSP_ENOTSHAREDMEM**	Specified memory segment identifier is not a valid shared memory segment.
0x8000803D	**DSP_EDYNLOAD**	There was an error when loading a dynamically loadable node.
0x80008040 – 0x8000804F	**DSP_EUSER1-16**	A node-specific error has occurred.

2.1.12. Kernel Level APIs

DSP/BIOS Bridge provides kernel level API for kernel modules to access DSP/BIOS Bridge in kernel mode. The kernel level API is exactly the same as the user level API, with two exceptions: **DSPManager_Open** and **DSPManager_Close** need not be called in kernel mode. Otherwise, all data structures, error codes, and function signatures remain the same as the user level API.

3. DSP-Side Bridge APIs

Please refer to omapsw_dspbridge_referenceguide.chm.

Appendix A: DSP Bridge API Data Structures

1. DSP_BUFFERATTR

The DSP_BUFFERATTR structure describes the attributes of a GPP-side data buffer used for messaging.

```
typedef struct {
    DWORD       cbStruct;
    UINT        uSegment;
    UINT        uAlignment;
} DSP_BUFFERATTR, * DSP_HBUFFERATTR;
```

Fields

cbStruct	The size in bytes of the DSP_BUFFERATTR structure.
uSegment	The memory segment from which the buffer is to be allocated. Currently, must be set to 1. (Default: 1).
uAlignment	Address alignment requirement for the buffer. Must be 0, 1, 2, or 4. (Default: 0).

Comments

DSP_BUFFERATTR is used when calling **DSPNode_AllocMsgBuf** and **DSPNode_FreeMsgBuf** to define buffer allocation requirements. All fields must be filled in before this structure is passed to DSPNode_AllocMsgBuf.

A pre-defined structure contains the default attributes used for a data buffer. A program can declare a structure equal to , and then modify the specific fields that differ from the defaults.

See Also

DSPNode_AllocMsgBuf

2. DSP_CBDATA

The DSP_CBDATA structure is used to pass variable amounts of data.

```
typedef struct {
    ULONG   cbData;
    BYTE    cData[1];
} DSP_CBDATA, *DSP_HCBDATA;
```

Fields

cbData	Specifies the length of the actual *cData* array in DSP_CBDATA.
cData[]	BYTE (character) array of arbitrary length.

Comments

The cData array of the DSP_CBDATA structure is initially declared to be of length 1. This array definition only serves as a placeholder for an actual array that your application program must define. Do not use this structure directly to define allocations in memory. Instead, use it as a cast over already allocated space. The following example shows how a DSP_CBDATA structure might be used in allocating a node.

```
# define ARGSIZE 64

typedef struct {
    ULONG   cbData;
    BYTE    cData[ARGSIZE];
} MY_NODEDATA;

/* Allocate task create args on stack. */
MY_NODEDATA argsBuf;

/* Fill node create argument structure: data and length. */
strncpy(argsBuf->cData, "1000", ARGSIZE);
argsBuf->cbData = strlen(argsBuf->cData) + 1;

/* Pass DSP_CBDATA (args and length) for the node's create phase */
lStatus = DSPNode_Allocate(hProcessor, pNodeID, (DSP_CBDATA
*)&argsBuf, NULL, &hNode);
```

Note

Because its data array is only one byte, this structure cannot be employed to define a usable information block. DSP_CBDATA can only be used as a cast, as shown above.

See Also

DSPNode_Allocate

3. DSP_ERRORINFO

The DSP_ERRORINFO structure describes the last exception condition signaled from the DSP to the GPP.

```
typedef struct {
    DWORD      dwErrMask;
    DWORD    dwVal1;
    DWORD    dwVal2;
    DWORD    dwVal3;
} DSP_ERRORINFO;
```

Fields

dwErrMask	The value of the event mask associated with the error information. Can be either DSP_SYSERROR or DSP_MMUFAULT.
dwVal1	Error information, determined by the value of *dwErrMask*:
DSP_SYSERROR	*dwVal1* indicates the DSP/BIOS, DSP/BIOS Bridge, or user-defined error value indicated in the DSP/BIOS **SYS_error** call.
DSP_MMUFAULT	*dwVal1* contains the high order bits of the MMU fault address register.
dwVal2	Error information, determined by the value of *dwErrMask*:
DSP_SYSERROR	Not used.
DSP_MMUFAULT	*dwVal2* contains the low order bits of the MMU fault address register.
dwVal3	Error information, determined by the value of *dwErrMask*:
DSP_SYSERROR	Not used.
DSP_MMUFAULT	*dwVal3* contains the status bits of the MMU fault status register.

Comments

When **DSPProcessor_GetState** is called, it returns processor information in a DSP_PROCESSORSTATE structure, which includes a DSP_ERRORINFO structure, for reporting the last exception condition that was signaled from the DSP to the GPP.

See Also

DSP_PROCESSORSTATE

4. DSP_MSG

The DSP_MSG structure is used for message passing between the GPP and DSP or IVA.

```
typedef struct {
    DWORD dwCmd;
    DWORD dwArg1;
    DWORD dwArg2;
} DSP_MSG, *DSP_HMSG;
```

Fields

dwCmd	An application specific message identifier. Valid values are from DSP_RMSUSERCODESTART to DSP_RMSUSERCODEEND, inclusive. Or'ing DSP_RMSBUFDESC into *dwCmd* will force DSP/BIOS Bridge to interpret the *dwArg1* and *dwArg2* fields as the address and size, respectively, of a shared memory buffer passed between the GPP and DSP.
dwArg1	The first message argument. If DSP_RMSBUFDESC is or'ed into *dwCmd*, *dwArg1* must be the address of a shared memory buffer allocated using **DSPNode_AllocMsgBuf** (if the buffer is allocated from the GPP side) or using **NODE_allocMsgBuf** (if the buffer is allocated from the DSP side).
dwArg2	The second message argument. If DSP_RMSBUFDESC is or'ed into *dwCmd*, *dwArg2* must be the size of the shared memory buffer whose address is specified in the *dwArg1* field.

Comments

DSP_MSG defines the structure of messages that are passed between the GPP and the DSP. *dwCmd* identifies the type of message, and determines the meaning of *dwArg1* and *dwArg2*. DSP node developers can define their own command codes for application-specific purposes.

Simple fixed length messages comprising a command id and two arguments can be encapsulated entirely within a DSP_MSG structure.

If a developer needs to send a larger, variable sized message, the *dwCmd* field can overloaded by or'ing the command identifier with DSP_RMSBUFDESC, and the *dwArg1* field can be set to point to a block of DSP/BIOS Bridge allocated shared memory that contains the rest of the message. In this case, DSP/BIOS Bridge will automatically perform the necessary translation between GPP virtual to DSP physical addresses during node messaging, allowing each side to access the same shared memory buffer.

Note also that *dwCmd* is set to RMS_EXIT when the Resource Manager is sending a shutdown command to a node. In this case *dwArg1* and *dwArg2* are unused.

See Also

DSPNode_AllocMsgBuf
NODE_allocMsgBuf

5. DSP_NDBPROPS

The DSP_NDBPROPS structure reports the attributes of a node, as stored in the DSP/BIOS Bridge Configuration Database (DCD).

```
typedef struct {
    DWORD               cbStruct;
    DSP_UUID            uiNodeID;
    CHARACTER           acName[DSP_MAXNAMELEN];
    DSP_NODETYPE        uNodeType;
    UINT                bCacheOnGPP;
    DSP_RESOURCEREQMTS  dspResourceReqmts;
    INT                 iPriority;
    UINT                uStackSize;
    UINT                uSysStackSize;
    UINT                uStackSeg;
    UINT                uMessageDepth;
    UINT                uNumInputStreams;
    UINT                uNumOutputStreams;
    UINT                uTimeout;
} DSP_NDBPROPS, *DSP_HNDBPROPS
```

Fields

cbStruct	The size in bytes of the DSP_NDBPROPS structure.
uiNodeID	The DSP_UUID for the node.
acName[32]	A human-readable (but not necessarily unique) name for the node. The name must be NULL-terminated, and can be up to DSP_MAXNAMELEN characters long.
uNodeType	The node type:
NODE_TASK	A task node.
NODE_DAISSOCKET	A TMS320 DSP Algorithm Standard (hereafter referred to as XDAIS) socket node.
NODE_DEVICE	A device node. (For device nodes, the fields in the DSP_NDBPROPS structure beyond dspResourceReqmts are undefined.)
bCacheOnGPP	A Boolean flag indicating if the node's DSP executable code should be cached on the GPP during node allocation (see **DSPNode_Allocate**).
dspResourceReqmts	A structure containing the resource requirements for the node, as specified in the DCD.
iPriority	The node's (DSP RTOS) runtime priority.
uStackSize	The node's stack size (in DSP MAUs).
uSysStackSize	Size of the node's system stack (in DSP MAUs). The value of this field is meaningful only for the C55x processor.
uStackSeg	The memory segment to place the node's stack. [Applicable when the DSP RTOS allows mapping of memory regions into distinct segments, e.g., mapping disjoint sections of different types of physical memory into different memory segments.]

uMessageDepth	The maximum number of simultaneous, outstanding messages between the node, another node, and the GPP. In other words, the number of message transport frames that the RM Server needs to allocate on the DSP to support the node's messaging needs.
uNumInputStreams	The number of input streams for the node.
uNumOutputStreams	The number of output streams for the node.
uTimeout	The timeout (in milliseconds) for blocking **DSPNode**_ calls for this node.

Comments

DSP_NDBPROPS is returned as an element of the DSP_NODEINFO and DSP_NODEATTR structures.

See Also

DSPProcessor_EnumNodes
DSPNode_Connect
DSP_NODEINFO

6. DSP_NODEATTR

The DSP_NODEATTR structure describes the attributes of a node, as reported by a DSPNode_GetAttr call. This is a superset of the attributes that can be set by a GPP program when the node is allocated. For that set, see the description of the DSP_NODEATTRIN structure.

```
typedef struct {
    DWORD       cbStruct;
    DSP_NODEATTRIN inNodeAttrIn;
    ULONG          uInputs;
    ULONG          uOutputs;
    DSP_NODEINFO   iNodeInfo;
} DSP_NODEATTR, * DSP_HNODEATTR;
```

Fields

cbStruct	The size in bytes of the DSP_NODEATTR structure.
inNodeAttrIn	A DSP_NODEATTRIN structure, which contains the node parameters that can be specified when the node is allocated.
uInputs	The number of input streams on this node from the DSP to the GPP.
uOutputs	The number of output streams on this node to the DSP from the GPP.
iNodeInfo	A DSP_NODEINFO structure, which includes static node information (from the configuration database), as well as dynamic information, such as stream connections, and execution state.

Comments

This structure is for reporting node attributes to an API client, and must not be passed to DSPNode_Allocate.

See Also

DSPNode_GetAttr
DSP_NODEATTRIN

7. DSP_NODEATTRIN

The DSP_NODEATTRIN structure describes the attributes of a node that may be set by an API client when the node is allocated with DSPNode_Allocate.

```
typedef struct {
    DWORD     cbStruct;
    INT       iPriority;
    UINT      uTimeout;
#ifdef OMAP_2430
    UINT      uProfileID;
    UINT      uHeapSize;  /* Reserved, For Bridge internal use */
    UINT      pGPPVirtAddr;  /* Reserved, For Bridge internal use */
#endif
} DSP_NODEATTRIN, * DSP_HNODEATTRIN;
```

Fields

cbStruct	The size in bytes of the DSP_NODEATTRIN structure.
iPriority	The new runtime priority level for the node. Valid values range from DSP_NODE_MIN_PRIORITY to DSP_NODE_MAX_PRIORITY, as also reported by the DSP_PROCESSORINFO structure. Default: The value specified for the node's "Runtime Priority" property in the DSP/BIOS Bridge Configuration Database. This field is not applicable to IVA nodes.
uTimeout	The implicit 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**). Valid values are from 0 to DSP_FOREVER. Default: The value specified for the node's "Timeout value of blocking calls" property in the DSP/BIOS Bridge Configuration Database.
uProfileID	Node heap profile. Each profile specifies the amount of Node heap memory requirement. The Node attributes read from the node DLL contains this information. The node private heap feature is supported only on 2430/3430 processors.
uHeapSize	Reserved. Used by DSP bridge library for internal use. Any user input values are ignored and overwritten.
pGPPVirtAddr	Reserved. Used by DSP bridge library for internal use. Any user input values are ignored and overwritten.

Comments

If the pAttrIn parameter in the DSPNode_Allocate call is NULL, then the default attributes for the node (as defined in the DCD), will be used when the node is allocated. Otherwise, all fields must be filled in before this structure is passed to DSPNode_Allocate.

structure equal to DSP_NODEATTRIN_DEFAULTS, and then modify the specific fields that differ from the defaults.

See Also

DSPNode_Allocate

8. DSP_NODEINFO

The DSP_NODEINFO structure is used to retrieve information about a node, including its DCD properties, its stream connections, and its execution state.

```
typedef struct {
    DWORD           cbStruct;
    DSP_NDBPROPS    nbNodeDatabaseProps;
    UINT            uExecutionPriority;
    DSP_NODESTATE   nsExecutionState;
    DSP_HNODE       hDeviceOwner;
    UINT            uNumberStreams;
    DSP_STREAMCONNECT   scStreamConnection[16];
    UINT        uNodeEnv;
} DSP_NODEINFO, * DSP_HNODEINFO;
```

Fields

cbStruct — The size in bytes of the DSP_NODEINFO structure.

nbNodeDatabaseProps — A DSP_NDBPROPS structure, which contains the node properties as defined in the DCD.

uExecutionPriority — The node's execution priority within the DSP RTOS. Unless the execution priority was changed via a **DSPNode_ChangePriority** call, this priority will be the same as that specified in the DCD. This field is not applicable to IVA1 nodes.

nsExecutionState — The node's execution state:

- NODE_ALLOCATED — The node has been allocated on the GPP (via **DSPNode_Allocate**), but has not been created on the DSP yet.
- NODE_CREATED — The node has been allocated on the GPP, and created on the DSP (via **DSPNode_Create**), but is in a pre-run state.
- NODE_RUNNING — The node has been launched into its execute phase, via a call to **DSPNode_Run**.
- NODE_PAUSED — The node has been temporarily suspended by an API call to **DSPNode_Pause**.
- NODE_DONE — The node has exited its execute phase, either because it finished its processing, or because **DSPNode_Terminate** was called.

hDeviceOwner — For device nodes, this is the handle of the task node that 'owns' the device. Note that this field is only relevant if one task is connected to the device; if more than one task is connected, this field indicates the last task node that was connected to the device.

uNumberStreams — The number of stream connections defined for the node. This field is not applicable to IVA1 nodes.

scStreamConnection[16] — An array of DSP_STREAMCONNECT structures, describing the node's stream connections (made via **DSPNode_Connect** calls). *uNumberStreams* indicates the number of valid DSP_STREAMCONNECT structures in the array.

See Also

DSP_NDBPROPS
DSP_STREAMCONNECT
DSPNode_Allocate
DSPNode_ChangePriority
DSPNode_Create
DSPNode_Run
DSPNode_Pause
DSPNode_Terminate

9. DSP_NOTIFICATION

The DSP_NOTIFICATION structure is used to define an OS-specific notification object.

```
typedef struct {
    PSTRING     psName;
    HANDLE      handle;
} DSP_NOTIFICATION, *DSP_HNOTIFICATION;
```

Fields

psName	The name of the event (N/A for Linux)
handle	The handle of the notification object.

Comments

A DSP/BIOS Bridge API client specifies a handle to a DSP_NOTIFICATION object when it registers for notification with **DSPProcessor_RegisterNotify**, **DSPNode_RegisterNotify**, or **DSPStream_RegisterNotify**.

See Also

DSPManager_WaitForEvents
DSPProcessor_RegisterNotify
DSPNode_RegisterNotify
DSPStream_RegisterNotify

10. DSP_PROCESSORATTRIN

The DSP_PROCESSORATTRIN structure describes the attributes of a processor that may be set by an API client when the processor is attached to with DSPProcessor_Attach.

```
typedef struct {
    DWORD     cbStruct;
    UINT      uTimeout;
} DSP_PROCESSORATTRIN, * DSP_HPROCESSORATTRIN;
```

Fields

cbStruct	The size in bytes of the **DSP_PROCESSORATTRIN** structure.
uTimeout	The timeout (in milliseconds) for blocking **DSPProcessor_** calls for this node. Currently not used. Default: 10,000 milliseconds.

Comments

If the *pAttrIn* parameter in the **DSPProcessor_Attach** call is NULL, then the default attributes for the processor will be used. Otherwise, all fields must be filled in before this structure is passed to **DSPProcessor_Attach**.

structure equal to **DSP_PROCESSORATTRIN_DEFAULTS**, and then modify the specific fields that differ from the defaults.

Note

If a GPP client is already attached to a DSP when **DSPProcessor_Attach** is called on the same DSP again, then the attributes specified in DSP_PROCESSORATTRIN will be ignored. That is, DSP_PROCESSORATTRIN is only useful on the first attach to a processor.

See Also

DSPProcessor_Attach

11. DSP_PROCESSORINFO

The DSP_PROCESSORINFO structure describes basic capabilities of a target processor, such as the speed of the processor, the amount of RAM visible to the processor, etc.

```
typedef struct {
    DWORD          cbStruct;
    DSP_PROCFAMILY uProcessorFamily;
    DSP_PROCTYPE   uProcessorType;
    UINT           uClockRate;
    ULONG       ulInternalMemSize;
    ULONG       ulExternalMemSize;
    UINT           uProcessorID;
    DSP_RTOSTYPE   tyRunningRTOS;
    INT      nNodeMinPriority;
    INT      nNodeMaxPriority;
} DSP_PROCESSORINFO, * DSP_HPROCESSORINFO;
```

Fields

cbStruct	The size in bytes of the DSP_PROCESSORINFO structure.
uProcessorFamily	The processor family (e.g., C5400, C5500, C6000, etc.).
uProcessorType	The processor type within its family (e.g., 5402, 5409, etc.).
uClockRate	The processor's clock rate (e.g., 80 MHz, 100 MHz, etc.).
ulInternalMemSize	The total amount of on-chip RAM (e.g., 16K, 32K, etc.).
ulExternalMemSize	The total amount of visible off-chip RAM (e.g., 64K, 128K, etc.).
uProcessorID	An identifier to distinguish a particular processor from other processors with the same characteristics.
tyRunningRTOS	An identifier indicating the type of DSP RTOS currently running on the processor (e.g., DSP/BIOS-II, OSE, etc.).
nNodeMinPriority	The minimum runtime priority level allowed for a task or XDAIS socket node.
nNodeMaxPriority	The maximum runtime priority level allowed for a task or XDAIS socket node.

Comments

When **DSPManager_ENUMProcessorInfo** is called, it returns processor information in a DSP_PROCESSORINFO structure.

See Also

DSPManager_EnumProcessorInfo

12. DSP_PROCESSORSTATE

The DSP_PROCESSORSTATE structure describes the state of a DSP processor.

```
typedef struct {
    DWORD       cbStruct;
    DSP_PROCSTATE  iState;
    DSP_ERRORINFO  errInfo;
} DSP_PROCESSORSTATE, * DSP_HPROCESSORSTATE;
```

Fields

cbStruct	The size in bytes of the DSP_PROCESSORSTATE structure.
iState	The execution state of the processor:
PROC_STOPPED	The processor is not running.
PROC_LOADED	An executable program is loaded but the processor is not running yet.
PROC_RUNNING	An executable program is loaded and running.
errInfo	A DSP_ERRORINFO structure, containing information about the last exception condition signaled from the DSP to the GPP.

Comments

When DSPProcessor_GetState is called, it returns processor information in a DSP_PROCESSORSTATE structure.

See Also

DSPProcessor_GetState

13. DSP_RESOURCEINFO

The DSP_RESOURCEINFO structure is used to retrieve information about a processor's resources. When **DSPProcessor_GetResourceInfo** is called, one of the parameters is a pointer to a DSP_RESOURCEINFO structure. In this structure *uResourceType* will define the specific resource to be queried; **DSPProcessor_GetResourceInfo** will return the resource information in the *ulResource* or *memStat* field within the result union of this same DSP_RESOURCEINFO structure, depending on the type of resource being queried.

```
typedef struct {
    DWORD               cbStruct;
    DSP_RESOURCEMASK    uResourceMask;
    union {
    ULONG               ulResource;
    DSP_MEMSTAT         memStat;
    } result;
} DSP_RESOURCEINFO, *DSP_HRESOURCEINFO;
```

Fields

cbStruct

The size in bytes of the DSP_RESOURCEINFO structure.

uResourceMask

Mask for processor resources.

ulResource

A numeric value for the resource. The meaning of *ulResource* is dependent upon *uResourceMask*.

Comments

When **DSPProcessor_GetResourceInfo** is called the Resource Manager will block while it queries the RM server running on the DSP for information about the specific resource. Once the DSP responds, **DSPProcessor_GetResourceInfo** reports the DSP's response in a DSP_RESOURCEINFO structure.

See Also

DSPProcessor_GetResourceInfo

14. DSP_RESOURCEREQMTS

The DSP_RESOURCEREQMTS structure is used to store information about a node's resource requirements, as specified in the node configuration database.

```
typedef struct {
    DWORD cbStruct;
    UINT     uStaticDataSize;
    UINT     uGlobalDataSize;
    UINT     uProgramMemSize;
    UINT     uWCExecutionTime;
    UINT     uWCPeriod;
    UINT     uWCDeadline;
    UINT     uAvgExectionTime;
    UINT     uMinimumPeriod;
} DSP_RESOURCEREQMTS, * DSP_HRESOURCEREQMTS;
```

Fields

cbStruct	The size in bytes of the DSP_RESOURCEREQMTS structure.
uStaticDataSize	The amount of static data memory required by the node (in DSP MAUs for DSP nodes; in bytes for IVA1 nodes).
uGlobalDataSize	The amount of global data memory required by the node (in DSP MAUs for DSP nodes; in bytes for IVA1 nodes).
uProgramMemSize	The maximum amount of program memory required by the node (in DSP MAUs for DSP nodes; in bytes for IVA1 nodes), at any given time.
uWCExecutionTime	The worst-case execution time for the node (in μsec).
uWCPeriod	The worst-case period for the node (in μsec).
uWCDeadline	The worst-case deadline for the node (in μsec).
uAvgExecutionTime	The average execution time for the node (in μsec).
uMinimumPeriod	The minimum period that achieves the average execution time (in μsec).

Comments

DSP_RESOURCEREQMTS is returned as part of a DSP_NDBPROPS structure.

See Also

DSP_NDBPROPS

15. DSP_STRMATTR

The DSP_STRMATTR structure defines DSP-side stream creation attributes, stored by a **DSPNode_Connect** call, which are later passed to the DSP node's create phase during **DSPNode_Create**.

```
typedef struct {
    UINT     uSegid;
    UINT     uBufsize;
    UINT     uNumBufs;
    UINT     uAlignment;
    UINT     uTimeout;
    DSP_STRMMODE   lMode;
    UINT     uDMAChnlId;
    UINT     uDMAPriority;
} DSP_STRMATTR;
```

Fields

uSegid — The DSP memory segment to be used for buffer allocations. For zero copy streaming (when *lMode* is set to STRMMODE_ZEROCOPY), *uSegid* must be set to DSP_SHMSEG0. This specifies a memory segment (SHMSEG0) shared between the GPP and DSP. For other streaming modes, *uSegid* can be the identifier of any valid data memory segment configured in the DSP/BIOS configuration tool. Default: 0.

uBufsize — The default stream buffer size, measured in bytes (GPP MAUs). Default: 32 bytes.

uNumBufs — The number of buffers that can be outstanding, (i.e., issued to the stream, but not reclaimed yet), at any point in time. Default: 2.

uAlignment — The memory alignment for stream buffers. Must be 0, or a power of 2. Default: 0.

uTimeout — The timeout (in milliseconds) for DSP-side **STRM_reclaim** calls. Default: 10,000 milliseconds.

lMode — The operating mode for a GPP to DSP stream connection. *lMode* defines the underlying communication transport mechanism used to transfer the stream buffer data between GPP and DSP. For a non GPP to DSP stream connection, *lMode* should be specified as STRMMODE_PROCCOPY. Default: STRMMODE_PROCCOPY.

- STRMMODE_PROCCOPY — The GPP and DSP processors perform the stream buffer data copy.
- STRMMODE_ZEROCOPY — Stream buffers are not copied; the shared memory buffer pointers are swapped between processors.

uDMAChnlId — Not currently used.

uDMAPriority — Not currently used.

Comments

If the *pAttr* parameter in the **DSPNode_Connect** call is NULL, then default attributes will be used for the stream. Otherwise, all fields must be filled in before this structure is passed to **DSPNode_Connect**.

The information in the DSP_STRMATTR structure is passed from the GPP to the DSP during **DSPNode_Create** into the DSP node's create function (through the create phase's *inDef* parameter for a DSP node's input stream, or via the *outDef* parameter for a DSP node's output stream. The node's create function would then use this information to create a DSP/BIOS stream object (using **STRM_create**) and stream buffers (using **STRM_allocateBuffer**).

The mapping from GPP side DSP_STRMATTR fields to the corresponding DSP side create phase RMS_StrmDef fields is shown below:

Table 7 Stream Definition Structure Mappings

RMS_StrmDef field	DSP_STRMATTR equivalent	Usage on DSP side
bufsize	uBufsize / word size 'C55x: word size = 2 'C64x word size = 1	bufsize argument to **STRM_create**.
nbufs	nNumBufs	nbufs field for **STRM_create** attrs argument
segid	uSegid	segid field for **STRM_create** attrs argument. Used in **STRM_allocateBuffer** calls.
align	uAlignment	align field for **STRM_create** attrs argument. Used in **STRM_allocateBuffer** calls.
timeout	uTimeout	timeout field for **STRM_create** attrs argument. Used in **STRM_reclaim** calls.
name[]	For a GPP to DSP connection, "/host<id>", a string comprised by "/host" plus an <id> constructed from the value of lMode. For a DSP node to DSP node connection, "/dbpipe<id>", where <id> is a free pipe number managed by the Resource Manager. For a DSP node to DSP device connection, this is the name of the DSP device driver, as specified in the device node's "Name" property.	The device name (first) argument to **STRM_create**.

Note

This structure is not applicable to IVA1 nodes.

The 'ZEROCOPY' and 'DMACOPY' stream modes are NOT supported on 2430/3430 platforms.

See Also

DSPNode_Connect

16. DSP_STREAMATTRIN

The DSP_STREAMATTRIN structure describes the attributes of a GPP-side stream (to or from the DSP), that may be set by an API client when the stream is opened with **DSPStream_Open**.

```
typedef struct {
    DWORD       cbStruct;
    UINT     uTimeout;
    UINT     uSegment;
    UINT     uAlignment;
    UINT     uNumBufs;
    DSP_STRMMODE   lMode;
    UINT     uDMAChnlId;
    UINT     uDMAPriority;
 } DSP_STREAMATTRIN, * DSP_HSTREAMATTRIN;
```

Fields

cbStruct — The size in bytes of the DSP_STREAMATTRIN structure.

uTimeout — The timeout (in milliseconds) for blocking **DSPStream_Reclaim** calls for this stream. Default: 10,000 milliseconds.

uSegment — The memory segment to be used when allocating buffers for the stream. For zero copy streaming (when *lMode* is set to STRMMODE_ZEROCOPY) *uSegment* **must** be set to DSP_SHMSEG0. This specifies a memory segment (SHMSEG0_GPP) shared between the GPP and DSP. For processor copy mode (*lMode* set to STRMMODE_PROCCOPY), *uSegment* must be set to 0. Default: 0.

uAlignment — The address alignment to be used for buffer allocations. Must be 0, or a power of 2. [Currently not used]. Default: 0.

uNumBufs — The number of buffers the stream should be prepared for. *uNumBufs* is the maximum number of buffers that can be outstanding (i.e., issued but not reclaimed), at any point in time. Default: 2.

lMode — The operating mode for a GPP to DSP stream connection. *lMode* defines the underlying communication transport mechanism used to transfer the stream buffer data between GPP and DSP. For a non GPP to DSP stream connection, *lMode* should be specified as STRMMODE_PROCCOPY. Default: STRMMODE_PROCCOPY.

- STRMMODE_PROCCOPY — The GPP and DSP processors perform the stream buffer data copy.
- STRMMODE_ZEROCOPY — Stream buffers are not copied; the shared memory buffer pointers are swapped between processors.

uDMAChnlId — Currently not used.

uDMAPriority — Currently not used.

Comments

If the *pAttrIn* parameter in the **DSPStream_Open** call is NULL, then the default attributes for the stream will be used when the stream is opened. Otherwise, all fields must be filled in before this structure is passed to **DSPStream_Open**.

This structure is not applicable to IVA1 nodes.

The ‘ZEROCOPY’ and ‘DMACOPY’ stream modes are NOT supported on 2430/3430 platforms.

structure equal to **DSP_STREAMATTRIN_DEFAULTS**, and then modify the specific fields that differ from the defaults.

See Also

DSPStream_Open

17. DSP_STREAMCONNECT

The DSP_STREAMCONNECT structure describes a stream connection between two nodes, or between a node and the GPP.

```
typedef struct {
    DWORD       cbStruct;
    DSP_CONNECTTYPE   lType;
    UINT     uThisNodeStreamIndex
    DSP_HNODE      hConnectedNode;
    DSP_UUID    uiConnectedNodeID;
    UINT     uConnectedNodeStreamIndex
 } DSP_STREAMCONNECT, * DSP_HSTREAMCONNECT;
```

Fields

cbStruct	The size in bytes of the DSP_STREAMCONNECT structure.
lType	The type of the stream connection relative to this node:
CONNECTTYPE_NODEOUTPUT	An output stream to another node (from this node to the connected node).
CONNECTTYPE_GPPOUTPUT	An output stream to the GPP (from this node to the GPP).
CONNECTTYPE_NODEINPUT	An input stream from another node (from the connected node to this node).
CONNECTTYPE_GPPINPUT	An input stream from the GPP (from the GPP to this node).
uThisNodeStreamIndex	The stream index for this node.
hConnectedNode	If *lType* is CONNECTTYPE_NODEOUTPUT or CONNECTTYPE_NODEINPUT, *hConnectedNode* is the handle of the connected node, as returned by a successful call to **DSPNode_Allocate**; otherwise *hConnectedNode* is the corresponding GPP stream handle.
uiConnectedNodeID	A DSP_UUID identifying the connected node.
uConnectedNodeStreamIndex	The stream index on the connected node, or the GPP.

Comments

DSP_STREAMCONNECT structures are returned as part of the DSP_NODEINFO structure.

This structure is not applicable to IVA1 nodes.

See Also

DSPNode_Allocate
DSP_NODEINFO

18. DSP_STREAMINFO

The DSP_STREAMINFO structure is used to retrieve information about a stream, including the number of bytes transferred on the stream, and a handle to the stream's underlying synchronization object.

```
typedef struct {
    DWORD           cbStruct;
    UINT            uNumberBufsAllowed;
    UINT            uNumberBufsInStream;
    ULONG           ulNumberBytes;
    HANDLE          hSyncObjectHandle;
    DSP_STREAMSTATE   ssStreamState;
} DSP_STREAMINFO, * DSP_HSTREAMINFO;
```

Fields

cbStruct	The size in bytes of the DSP_STREAMINFO structure.
uNumberBufsAllowed	The maximum number of data buffers that can be issued to the stream at any given time.
uNumberBufsInStream	The number of data buffers currently issued to the stream.
ulNumberBytes	The number of bytes transferred on the stream, since the stream was most recently reset (via **DSPStream_Idle**).
hSyncObjectHandle	The handle to the stream's underlying GPP OS synchronization object.
ssStreamState	The stream's execution state:
STREAM_IDLE	The stream is open, but no I/O is currently pending.
STREAM_PENDING	A buffer has been queued to the stream, and the I/O operation is in progress.
STREAM_READY	An I/O operation has completed, and a buffer is ready to be reclaimed from the stream.
STREAM_DONE	An I/O operation has completed with zero bytes transferred, (i.e., end-of-stream has been detected).

Comments

When **DSPStream_GetInfo** is called, it returns stream information in a DSP_STREAMINFO structure.

This structure is not applicable to IVA nodes.

See Also

DSPStream_GetInfo
DSPStream_Idle

19. DSP_UUID

A DSP_UUID is a Universally Unique Identifier (UUID). DSP/BIOS Bridge uses DSP_UUIDs to uniquely identify individual nodes.

```
typedef struct _DSP_UUID {
    ULONG   ulData1;
    USHORT  usData2;
    USHORT  usData3;
    BYTE    ucData4;
    BYTE    ucData5;
    CHAR    ucData6[6];
} DSP_UUID, *DSP_HUUID;
```

Fields

ulData1	The low field of the timestamp for when the UUID was created.
usData2	The middle field of the timestamp.
usData3	The high field of the timestamp, combined with the UUID version field.
ucData4	The high field of a clock sequence, combined with the UUID variant field.
ucData5	The low field of the clock sequence.
ucData6[6]	The IEEE 802 node address of the host machine that the created the UUID.

Comments

UUIDs are typically generated using a utility program that fetches a time stamp and clock sequence code, and accesses the network address of the machine the utility is being run on, and creates the unique identifier.

Microsoft Corporation GUIDs are one of the defined variants of UUIDs.

See Also

More details about UUIDs can be found at the following URL:
http://www.opengroup.org/onlinepubs/009629399/apdxa.htm.