DESIGN DOCUMENT

DSP/BIOS™ LINK

ENHANCED MULTIPROCESS SUPPORT

LNK 157 DES

Version 0.30

This page has been intentionally left blank.

IMPORTANT NOTICE

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

1 Introduction

1.1 Purpose & Scope

This document describes the design of enhanced multi-process support for DSP/BIOS™ LINK.

The document is targeted at the development team of DSP/BIOS™ LINK.

1.2 Terms & Abbreviations

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

1.3 References

1.	LNK 001 PRD	DSP/BIOS™ LINK Generic Product Requirement Document
2.	LNK 147 PRD	DSP/BIOS™ LINK Version-specific Product Requirement Document

1.4 Overview

DSP/BIOS™ LINK is runtime software, analysis tools, and an associated porting kit that simplifies the development of embedded applications in which a general-purpose microprocessor (GPP) controls and communicates with a TI DSP. DSP/BIOS™ LINK provides control and communication paths between GPP OS threads and DSP/BIOS™ tasks, along with analysis instrumentation and tools.

This module provides the design for enhanced multi-process support within *DSPLINK*. This allows multiple applications/processes to use *DSPLINK* independently, and without being aware of each other.

2 Requirements

SR110. (1.50) Enhanced multi-process support

DSP/BIOS Link must offer enhanced multi-process support. This must includes the following:

- Multiple unrelated processes must be able to come and go. There must be no hidden dependencies based on which process, for example, first used PROC_attach as there are in the current version. The first process to attach to the DSP must not be required to be the last to detach.

  * A reference count shall be used to ensure that the last PROC_detach () cleans up the system instead of processor ownership checks.

- Multiple independent applications must be able to gain access to, and utilize the resources provided by the DSP for the PROC module.

  * Multiple applications must be able to make calls to PROC_load () or PROC_start (). However, based on reference count, and if the applications are attempting to load the same DSP executable, a corresponding status code must be returned instead of re-loading the DSP. Reference count shall be checked for PROC_start () and PROC_stop () also.

3 Assumptions

None.

4 Constraints

1. Since PrOS only supports threads and not processes, PROC_attach () must not be called for each thread in PrOS applications.

5 High Level Design

Multiple applications may wish to use the services provided by *DSPLINK* to control and communicate with the DSP. In such scenarios, with existing versions of DSP/BIOS™ LINK, the applications must be integrated by a system integrator, to ensure that the first application initializes *DSPLINK*, and loads and starts the DSP before the second application can communicate with the DSP.

It is desirable to avoid such integration and allow independently developed applications to use *DSPLINK* without being aware of other applications doing the same.

In addition, if multiple different applications using *DSPLINK* are running on the target processor, a crash in one of these must not affect the execution of the other application.

5.1 Features

The following features shall be provided for enhanced multi-process support:

1. An application can be written to execute singly using *DSPLINK* to control and communicate with the DSP.
2. The same application can be used without any changes in the applications source code, to run simultaneously along with another application also using *DSPLINK*. The only consideration to be used while writing the application, is that the DSPLINK resources (e.g RingIO/MSGQ names) used by the applications must be unique for the system.
3. The applications shall use the same integrated DSP executable containing DSP-side content required for all the co-existing GPP-side applications.

5.2 Use case scenario

The following use-case scenario shall be supported:

Two applications contain source as follows:

```
PROC_setup (…) ;
PROC_attach (…) ;
POOL_open (poolId, poolParams) ;
PROC_load (…, dspExec, …) ;
PROC_start (…) ;
/* Application-specific code */
PROC_stop (…) ;
POOL_close (…) ;
PROC_detach (…) ;
PROC_destroy (…) ;
```

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

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

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

5.3 Design details

5.3.1 Changes to ownership concept:

1. The first process to attach shall not be designated as the owner of the DSP.
2. `PROC_setup ()/PROC_destroy ()`: `PMGR_IsSetup` flag shall be replaced by `PMGR_SetupRefCount` reference count. This reference count shall be incremented in `PROC_setup ()` and decremented in `PROC_destroy`. If `refCount` is 0, actual setup is done. When the reference count reaches 0, actual destroy is done.
3. `PROC_attach ()/PROC_detach ()`: `PMGR_PROC_Object` shall be modified to include an `attachRefCount` field. This reference count shall be incremented in

`PROC_attach ()` and decremented in `PROC_detach ()`. If reference count is 0, actual attach is done. When reference count reaches 0, actual detach is done, and DSP is powered down if power control is enabled in the configuration.

4. `PROC_load ()`: The DSP state shall be used to identify whether it has been already loaded. If already loaded, this call does not result in any actual load on the DSP.
5. `PROC_start ()`/`PROC_stop ()`: `PMGR_PROC_Object` shall be modified to include a `startRefCount` field. The reference count shall be incremented in `PROC_start ()` and decremented in `PROC_stop ()`. If reference count is 0, actual start is done. When reference count reaches 0, actual stop is done.

5.3.2 Multi-process cleanup on Linux:

To support cleanup in a multi-processing scenario, the PROC resources allocated by the process getting terminated shall be freed during cleanup. The system state shall not be polluted, and other applications can continue using *DSPLINK* to communicate with the DSP.

For every process, PROC, MSGQ and POOL setup and shutdown calls shall be tracked. It shall be tracked whether the following APIs have been called by the process:

- `PROC_setup`
- `PROC_attach`
- `POOL_open (for all the max. possible pools)`
- `PROC_start`
- `MSGQ_transportOpen`
- `MSGQ_transportClose`
- `PROC_stop`
- `POOL_close`
- `PROC_detach`
- `PROC_destroy`

With this tracking, it shall be checked in shutdown APIs, whether the corresponding startup call had been made for that process. This shall ensure that an errant process does not corrupt the reference count for each API, resulting in causing a crash in other processes.

There are two scenarios, for which cleanup is to be performed:

1. **Abnormal process termination:** When an application process ends abnormally, all threads within it are killed. This can happen for any of the following scenarios:
   - Application crash
   - Segmentation fault
   - Any other crash resulting in kernel still remaining usable
   - User kills the process with Ctrl C
   - User kills the process through kill command

   In all these scenarios, it must be possible to:
   - Continue execution of other applications using *DSPLINK*.
   - Perform basic shutdown calls for PROC, POOL and MSGQ to free these basic resources allocated by this process.

- Minimize memory or resource leaks.
- Allow stopping & restarting *DSPLINK* without having to reinsert the kernel module or reboot the hardware.

On Linux, this shall be done by registering a signal handler for process termination signals. This signal handler shall make all shutdown API calls.

- Signal handling shall be enabled by default
- Whether *DSPLINK* should handle signals for cleanup shall be dynamically configurable
- The signals to be handled shall be dynamically configurable

2. **Normal exit:** When an application exits, it performs the shutdown calls for PROC, POOL and MSGQ to free these basic resources allocated by it. If this is not done, the kernel resources and shared memory resources become unavailable to other applications, and are lost. *DSPLINK* shall allow applications to exit, making a minimum of exit API calls to free up all resources allocated by that process.

   On Linux, this shall be done by registering an atexit handler that makes all shutdown APIs calls.

6 Low Level Design

6.1 Constants & Enumerations

None.

6.2 Typedefs & Data Structures

6.2.1 PROC_CurStatus

This structure defines the current status of the PROC component for each process.

Definition

```
typedef struct PROC_CurStatus_tag {
    Bool  isSetup ;
    Bool  isAttached   [MAX_DSPS] ;
    Bool  isStarted    [MAX_DSPS] ;
#if defined (POOL_COMPONENT)
    Bool  poolIsOpened [MAX_POOLENTRIES] ;
#endif /* if defined (POOL_COMPONENT) */
#if defined (MSGQ_COMPONENT)
    Bool  mqtIsOpened  [MAX_DSPS] ;
#endif /* if defined (MSGQ_COMPONENT) */
} PROC_CurStatus ;
```

Fields

`isSetup`	Indicates whether PROC has been setup in this process.
`isAttached`	Indicates whether PROC has been attached in this process for the specified processor ID.
`isStarted`	Indicates whether PROC has been started in this process for the specified processor ID.
`poolIsOpened`	Indicates whether POOL has been opened in this process for the specified pool ID. Only defined if POOL component is enabled.
`mqtIsOpened`	Indicates whether MSGQ transport has been opened in this process for the specified processor ID. Only defined if MSGQ component is enabled.

Comments

This structure is used for tracking the PROC, MSGQ and CHNL startup and shutdown API calls made in each process. This is required to ensure that if a process has not called the startup API, it should not be allowed to call the corresponding shutdown API.

Constraints

None.

See Also

`PROC_Object`

6.2.2 PROC_Object

This structure defines the PROC object, which contains state information required by the PROC user-side component.

Definition

```
typedef struct PROC_Object_tag {
    SYNC_USR_CsObject * syncCsObj ;
    PROC_CurStatus      curStatus ;
} PROC_Object ;
```

Fields

`syncCsObj`	Mutex for protecting PROC operations in user-space.
`curStatus`	Current status for the components for each process.

Comments

This object is maintained in user space for each process and contains information for protecting and tracking user-space resources.

Constraints

None.

See Also

`PROC_CurStatus`

6.3 API Definition

6.3.1 PROC_setup

This function sets up the necessary data structures for the PROC component.

Syntax

```
DSP_STATUS PROC_setup (LINKCFG_Object * linkCfg) ;
```

Arguments

```
IN      LINKCFG_Object *          linkCfg
```

Pointer to the configuration information structure for DSP/BIOS™ LINK. If NULL, indicates that default configuration should be used.

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SALREADYSETUP`	The *DSPLINK* driver has already been setup by some other application/process.
`DSP_EALREADYSETUP`	The *DSPLINK* driver is already setup in this process.
`DSP_ECONFIG`	Error in specified dynamic configuration. Please check CFG_<PLATFORM>.c
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This function is the first *DSPLINK* API that applications must call before they can make calls to any other *DSPLINK* APIs. The only DSPLINK API that can be called before `PROC_setup` is `PROC_getState ()`.

This API initializes the *DSPLINK* driver. This API can be successfully called once by every process in the system. However, it is not a must for every application/process to make the call, if at least one process has initialized the *DSPLINK* driver before the other applications/processes.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

1. The calling applications must ensure that the contents of the dynamic configuration structure passed to this API are correct.
2. If called by multiple applications, the calling applications must pass the same dynamic configuration structure to this API. Otherwise it can result in indeterminate system behavior.

See Also

`PROC_destroy ()`

6.3.2 PROC_destroy

This function destroys the data structures for the PROC component, allocated earlier by a call to `PROC_setup ()`.

Syntax

```
DSP_STATUS PROC_destroy (Void) ;
```

Arguments

None.

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SDESTROYED`	The final client has finalized the driver.
`DSP_EACCESSDENIED`	The *DSPLINK* driver was not setup in this process.
`DSP_ESETUP`	The *DSPLINK* driver was not setup.
`DSP_EMEMORY`	Operation failed due to memory error.
`DSP_EFAIL`	General failure.

Comments

This function is the last *DSPLINK* API that applications must call after they have no further need to use *DSPLINK* services. The only DSPLINK API that can be called before `PROC_setup` is `PROC_getState ()`.

This API finalizes the *DSPLINK* driver. This API can be successfully called once by every process in the system. However, if the `PROC_setup ()` API was not called in the process, `PROC_destroy ()` must not be called.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

None.

See Also

`PROC_setup ()`

6.3.3 PROC_attach

This function attaches the client to the specified DSP and also initializes the DSP (if required).

Syntax

```
DSP_STATUS PROC_attach (ProcessorId   procId, PROC_Attrs *  attr) ;
```

Arguments

`IN`	`ProcessorId`	`procId`
	DSP identifier.	
`IN OPT`	`PROC_Attrs *`	`attr`
	Optional attributes for the processor on which attach is to be done.	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SALREADYSETUP`	Successful attach. Also, indicates that another client has already attached to DSP.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EACCESSDENIED`	Not allowed to access the DSP.
`DSP_EALREADYCONNECTED`	Another thread of the same process has already attached to the processor.
`DSP_EWRONGSTATE`	Incorrect state for completing the requested operation.
`DSP_EFAIL`	General failure.

Comments

When any client wishes to use a specific DSP, it first needs to attach to the DSP by calling this API specifying the required DSP ID.

Every process that needs to use *DSPLINK* with the specific DSP must make a call to this API.

This API carries out all initialization required to be able to use *DSPLINK* with the specified DSP ID from the calling process. This API can be successfully called once by every process in the system. If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

`PROC_setup ()` must be called by at least one client in the system before any process can call this API.

See Also

`PROC_detach ()`

6.3.4 PROC_detach

This function detaches the client from specified processor. If the caller is the owner of the processor, this function releases all the resources that this component uses and puts the DSP in an unusable state (from application perspective).

Syntax

```
DSP_STATUS PROC_detach (ProcessorId procId) ;
```

Arguments

`IN`	`ProcessorId`	`procId`
	DSP identifier.	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SDETACHED`	The final process has detached from the specific processor.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_ESETUP`	The *DSPLINK* driver was not setup.
`DSP_EACCESSDENIED`	Not allowed to access the DSP.
`DSP_EATTACHED`	Not attached to the target processor.
`DSP_EWRONGSTATE`	Incorrect state for completing the requested operation.
`DSP_EFAIL`	General failure.

Comments

This function is the last *DSPLINK* API that all applications/processes must call after they have no further need to use *DSPLINK* services for a specific processor ID. Once this API has been called, the process cannot perform any further activities specific to the DSP.

This API finalizes the *DSPLINK* driver for the specified processor ID in the calling process. This API can be successfully called once by every process in the system. However, if the `PROC_attach ()` API was not called in the process for the specific processor ID, `PROC_detach ()` must not be called.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

None.

See Also

`PROC_attach ()`

6.3.5 PROC_load

This function loads the specified DSP executable on the target DSP.

Syntax

```
DSP_STATUS PROC_load (ProcessorId  procId,
                      Char8 *      imagePath,
                      Uint32       argc,
                      Char8 **     argv) ;
```

Arguments

IN	`ProcessorId`	`procId`
	DSP identifier.	
IN	`Char8 *`	`imagePath`
	Full path to the image file to load on DSP.	
IN	`Uint32`	`argc`
	Number of argument to be passed to the base image upon start.	
IN	`Char8 **`	`argv`
	Arguments to be passed to DSP main application.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_SALREADYLOADED	The specified processor has already been loaded.
DSP_EINVALIDARG	Invalid argument.
DSP_EACCESSDENIED	Not allowed to access the DSP.
DSP_ESETUP	The *DSPLINK* driver has not been setup.
DSP_EATTACHED	This process has not attached to the specified processor.
DSP_EPENDING	H/W specific error. The request can't be serviced at this point of time.
DSP_EFILE	Invalid base image.
DSP_ESIZE	Size of the .args section is not sufficient to hold the passed arguments.
DSP_EWRONGSTATE	Incorrect state for completing the requested operation.
DSP_EFAIL	General failure.

Comments

Any client that wishes to use the DSP can call this API to load the DSP executable on it. However, only the first client to call this API actually loads the DSP. The subsequent calls are ignored.

Constraints

All applications using a specific DSP at the same time must ensure that they use the same base DSP executable.

See Also

`PROC_start ()`

6.3.6 PROC_start

This function starts execution of the loaded code on DSP from the starting point specified in the DSP executable loaded earlier by call to `PROC_load ()`.

Syntax

```
DSP_STATUS PROC_start (ProcessorId procId) ;
```

Arguments

`IN`	`ProcessorId`	`procId`
	DSP identifier.	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SALREADYSTARTED`	The specified processor has already been started.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EPENDING`	H/W specific error. The request can't be serviced at this point of time.
`DSP_EACCESSDENIED`	Not allowed to access the DSP.
`DSP_ESETUP`	The *DSPLINK* driver has not been setup.
`DSP_EATTACHED`	This process has not attached to the specified processor.
`DSP_EWRONGSTATE`	Incorrect state for completing the requested operation.
`DSP_ECONFIG`	The specified processor could not be started. Driver handshake failed due to DSP driver initialization/configuration failure.
`DSP_EFAIL`	General failure.

Comments

Any client that wishes to use the DSP can call this API to start the DSP executable on it. However, only the first client to call this API actually starts the DSP. The subsequent calls are ignored.

Constraints

All applications using a specific DSP at the same time must ensure that they use the same base DSP executable.

See Also

```
PROC_load ()
PROC_stop ()
```

6.3.7 PROC_stop

This function stops execution of the specified DSP. This API may place the DSP in reset.

Syntax

```
DSP_STATUS PROC_stop (ProcessorId procId) ;
```

Arguments

`IN`	`ProcessorId`	`procId`
	DSP identifier.	

Return Value

`DSP_SOK`	Operation successfully completed.
`DSP_SSTOPPED`	The final process has stopped the DSP execution.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EACCESSDENIED`	Not allowed to access the DSP.
`DSP_ESETUP`	The *DSPLINK* driver has not been setup.
`DSP_EATTACHED`	This process has not attached to the specified processor.
`DSP_EWRONGSTATE`	Incorrect state for completing the requested operation.
`DSP_ESTARTED`	The specified processor has not been started.
`DSP_EFAIL`	General failure.

Comments

Once a process has completed its processing requiring transfers with the DSP, it can call this API to stop the execution of the DSP with this API. However, it is not essential to call this API if it has been previously called by some other application/process.

Only the last client to call this API actually stops the DSP. The earlier calls are ignored.

If the `PROC_start ()` API was not called in the process, `PROC_stop ()` must not be called.

If this API is called more than once in a single process (even if called by different threads within the process), the subsequent calls return an error.

Constraints

None.

See Also

`PROC_load ()`