DSP/BIOS™ LINK

MPCS DESIGN

LNK 133 DES

Version 1.65.00.02

This page has been intentionally left blank.

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

1 Introduction

1.1 Purpose & Scope

This document describes the design and interface definition of the multi-processor critical section component.

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

1.2 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
MPCS	Multi-processor Critical Section
SMA	Shared Memory Allocator
☞	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.3 References

1.	LNK 084 PRD	DSP/BIOS™ LINK Product Requirement Document
2.	LNK 082 DES	POOL Design Document
3.	LNK 132 DES	PCI Driver Redesign
4.	LNK 207 DES	Configurable TSK and SWI approach

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 multi-processor critical section (MPCS) component.

This document gives an overview of the MPCS component on the GPP and DSP-sides of *DSPLINK*. The document also gives a detailed design of the MPCS component.

2 Requirements

Please refer to section 16.3 of LNK 084 PRD - DSP/BIOS™ LINK Product Requirement Document.

On devices with shared memory, applications may need to define their own data structures on shared memory. Such data structures can be used for communicating small pieces of information between the processors. However, applications need to ensure mutually exclusive access to such data structures to ensure consistency of data. To enable such scenarios, the product shall provide a new module that provides this functionality.

R110 This release shall support management of a shared memory specific lock on devices that support shared memory.

R111 The APIs shall enable applications to acquire and release the lock in an efficient manner.

R112 The APIs shall allow protected access to the data structures from processes running on remote processor as well as on the same processor.

In addition, the MPCS component must meet the following generic requirements:

1. The API exported by the MPCS component shall be common across different GPP operating systems.
2. Both the DSP as well as the GPP side shall expose same API.
3. Multiple threads can perform lock and unlock operations on the MPCS objects created in the system. However, ownership shall come into play while creation/deletion of the MPCS object. The processor which creates the MPCS object shall be the one which deletes it.

3 Assumptions

The MPCS design makes the following assumptions:

1. The hardware allows provision of a buffer pool, to which both the GPP and the DSP have access.

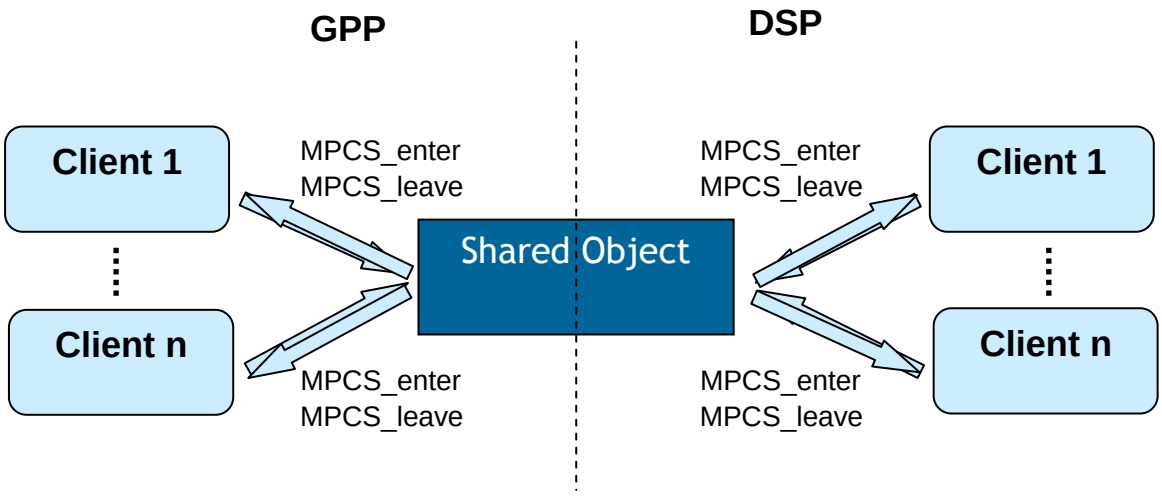
4 Constraints

- The MPCS object must be allocated and freed through POOL APIs provided by *DSPLINK*. Elements allocated through the POOL API can be accessed by multiple processors. Any other means for memory allocation (for example: standard OS calls) will fail as the elements cannot be accessed across processors.
- The user has to use unique identifier to identify individual MPCS objects across the system.

5 MPCS High Level Design

5.1 Introduction

In a multiprocessor system having shared access to a memory region, a multi-processor critical section between GPP and DSP can be implemented. This MPCS object can be used by applications to provide mutually exclusive access to a shared region between multiple processors, and multiple processes on each processor. In cases where a shared memory region does not exist, the module shall internally perform the synchronization required to provide the protection required by the MPCS component.



- Multi-processor critical section (MPCS) between GPP and DSP.
- Achieves mutually exclusive access to shared objects (data structures, buffers, peripherals etc.)

OS specific Level 1 protection to protect access to shared resource between tasks on ARM core.

DSP/BIOS specific Level 1 protection to protect access to shared resource between tasks on DSP core.

The MPCS component provides the following functionality to user applications:

- Creation of an MPCS object identified by a system-wide unique name.
- Deletion of an MPCS object identified by a system-wide unique name.
- Opening the MPCS object identified by a system-wide unique name, to get a handle to it. The handle returned by this API can be used for accessing the MPCS object in the process space of the caller.
- Entering the critical section specified by the MPCS object handle.
- Closing the MPCS object identified by its handle.

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

5.2 Current MPCS design Issues

MPCS enters the deadlock and priority inversion situation between the tasks at the GPP and DSP Level.

Please find below certain examples where a possibility of inversion/deadlock in earlier design is possible.

5.2.1 Global priority inversion:

1. DSP side in DSPLink API uses the default SWI mode
2. GPP-side can be preempted after taking MPCS lock
   - Process/thread that has taken the user/kernel MPCS lock can be preempted by a higher priority process/thread
   - DSP-side may now try to take the same MPCS lock
   - Results in scheduler disable till GPP-side releases the lock
   - If the GPP thread is preempted, this can be non-deterministic

Issue is seen because GPP-side is not using same level of protection as DSP-side

5.2.2 Deadlock situation:

- DSP side in DSPLink API can use either SWI or TSK mode
- The following pre-conditions exist.
  - A high priority task on GPP-side shares a resource with a low priority task on DSP side.
  - A low priority task on GPP-side shares a resource with a high priority task on DSP side.

5.2.2.1 MPCS Deadlock scenario:

Details on scenario causing deadlock

Scenario on ARM:

- Lower priority task 1 on ARM is using MSGQ. This task takes the MSGQ MPCS lock. After taking the lock this task is pre-empted by higher priority Task 2 on ARM
- Higher priority task 2 is performing RingIO operations. This RingIO MPCS lock is different from the MPCS lock.

Scenario on DSP at the same time:

- Task B on DSP side is using RingIO. This is a low priority task which is communicating with its high priority counterpart on Arm i.e. Task 2. This task takes the RingIO MPCS lock. At this point we have a situation in which

  a. Low priority Task 1 pre-empted in MSGQ MPCS on ARM.

  b. High priority Task 2 is using RingIO on ARM.

  c. Low priority Task B takes the RingIO MPCS.

- Task A is performing MSGQ operation. This is a high priority task which is communicating with its low priority counterpart on Arm i.e. Task 1.

- At this point we have a situation in which

  a. Low priority Task 1 pre-empted while in MSGQ MPCS on ARM

  b. High priority Task 2 using RingIO on ARM

  c. Low priority Task B takes the RingIO MPCS on DSP.

d. This is then pre-empted by high priority task a on DSP which desires MSGQ but gets stuck in MPCS section.

5.3 MPCS Bug details

5.3.1 MPCS Bug ID:

5.3.2 MPCS BUG details

MPCS enters the deadlock and Priority inversion situation between the tasks at the GPP and DSP Level. Message queue and Ring IO IPC components face deadlock in some cases as detailed in Deadlock situation:.

5.4 Design Goals

To overcome the issues mentioned in the above section we can implement on both the cores with the following designs on MPCS.

- Level 1: OS specific protection to protect access to shared resource between tasks on same core.

  a. Use semaphores for task mode, priority ceiling or Interrupt disable for SWI mode on ARM side.

  b. DSP/BIOS APIs to be used to DSP side.

- Level 2: Peterson's algorithm to protect access to shared resource between tasks on different core.

5.5 Code Flow

5.5.1 Introduction

The DSP/BIOS LINK code has two sections GPP and DSP side. GPP side deals with the ARM side implementations. DSP side is the DSP side implementations.
In GPP side implementations the API folder is the user side implementation. And rests of the folders are kernel side implementations. DSP side implementation is the complex integration of DSPLINK protocol implementation using DSP/BIOS APIs.

The GPP and DSP side implement both Task mode and SWI mode for MPCS protection.

5.6 MPCS Re-Design Solutions.

MPCS redesign can be done on two modes.

1. Task and SWI mode on ARM side.
2. Task and SWI mode on DSP side.

5.6.1 Level 1 Protection for Message Queues:

5.6.1.1 ARM side:

Task mode:

- **User and Kernel side:**
  a. Local lock of each MPCS object points to a single global lock used to protect the MPCS section. This lock is taken in MPCS_enter for every MPCS object.

SWI mode:

- **User and Kernel side:**
  a. Boost the priority of the task to a highest priority in MPCS_enter.
  b. Build option to add priority to be added with the MPCS object for the backward compatibility

5.6.1.2 DSP Side:

Task mode:

- Local lock of each MPCS object points to a single global lock used to protect the MPCS section. This lock is taken in MPCS_enter for every MPCS object.

SWI mode:
Scheduler is disabled.

Note: The Level two protection is implemented using Peterson algorithm.

5.7 MPCS -OS Dependent design implementation.

The ARM side has to be implemented with different APIs specific to the Operating system supported in the DSPLINK. The OS specific protection to protect access to shared resource between tasks on same core.

Given below are some of the APIs to be used on various platforms.

- Linux
  - User side protection - System 5 semaphores
  - Kernel side protection - Mutual exclusion lock - synchronization mechanism used to serialize the execution of threads using OS API mutex_lock_interruptible
- PrOS
  - User side protection - Critical section using API loc_mutex
  - Kernel side protection - Protection with all ISRs disabled - Protects critical regions of code from preemption by tasks, DPCs and all interrupts using OS API loc_cpu
- WinCE
  - User side protection – Protection using OS API WaitForSingleObject
  - Kernel side protection - Protection using OS API EnterCriticalSection
- Other OS
  - Task mode: The expectation is that the global lock used should support nesting.
  - Swi mode: The expectation is a design which matches closely to disabling the scheduler should be used.

6 Sequence Diagrams

None.

7 Low level design

The MPCS component has the same design on both the GPP and DSP sides. This section primarily refers to the GPP side design. However, the DSP-side design shall contain the same enumerations, structures, and API definitions, with minimal changes for different types on the GPP and DSP-sides.

7.1 Constants & Enumerations

7.1.1 MPCS_NUMENTRIES

Defines the maximum number of MPCS objects that can be created in the system.

Definition

#define MPCS_NUMENTRIES 256

Comments

This definition is present in the header file generated by the configuration script based on information provided in the configuration file for the platform: CFG_<platform.TXT. The value of the constant may vary depending on the value specified by the user in the configuration file.

Constraints

None.

See Also

MPCS_Region

7.1.2 MPCS_TABLE_SIZE

Defines the size of the MPCS region in the shared region containing information about all MPCS objects

Definition

#define MPCS_TABLE_SIZE sizeof (MPCS_Region)

Comments

This definition gives the size of the MPCS region, used by the *DSPLINK* static configuration to calculate the amount of memory that needs to be reserved for the MPCS component.

Constraints

None.

See Also

MPCS_Region

7.1.3 MPCS_INVALID_ID

Defines an invalid value for identifier(s) used by the MPCS component

Definition

#define MPCS_INVALID_ID (Uint32) -1

Comments

The invalid ID is used to indicate that an identifier value is not a valid value. For example, if used as the poolId for the MPCS object, it indicates that a pool was not used to allocate the object. This is useful for the global MPCS object used for protecting the MPCS region entries.

Constraints

None.

See Also

None.

7.1.4 MPCS_RESV_LOCKNAME

Defines the special reserved name prefix of the MPCS object(s) which are not stored in the entries table of the MPCS region.

Definition

#define MPCS_RESV_LOCKNAME "DSPLINK_MPCS_RESERVED"

Comments

This constant is provided to internal users of the MPCS component, where the memory provided for the MPCS object may not be allocated from a pool, and the object is not identified by name or stored in the entries table. This reserved prefix is used for names of such MPCS objects.

Constraints

None.

See Also

MPCS_RESV_LOCKNAMELEN

7.1.5 MPCS_RESV_LOCKNAMELEN

Defines the string length of the special reserved name prefix of the MPCS object(s) which are not stored in entries table of the MPCS region.

Definition

#define MPCS_RESV_LOCKNAMELEN 17

Comments

This constant is provided to internal users of the MPCS component, where the memory provided for the MPCS object may not be allocated from a pool, and the object is not identified by name or stored in the entries table. The reserved prefix of length defined by this constant is used for the names of such MPCS objects.

Constraints

None.

See Also

MPCS_RESV_LOCKNAME

7.2 Typedefs & Data Structures

7.2.1 MPCS_Attrs

This structure defines the attributes for creation of MPCS object.

Definition

```
typedef struct MPCS_Attrs_tag {
   Uint16  poolId ;
} MPCS_Attrs ;
```

Fields

poolId	ID of the pool used to allocate the MPCS object.

Comments

The attributes can contain the pool ID specified by the user. This will determine from which pool the MPCS_create () function will allocate memory for the MPCS.

Constraints

None.

See Also

None.

7.2.2 MPCS_Entry

This structure defines the global entry structure for an MPCS object. Every MPCS object in the system is identified through information present in the entry structure.

Definition

```
typedef struct MPCS_Entry_tag {
   Uint16      ownerProcId ;
   Uint16      poolId ;
   Pvoid       physAddress ;
   Char8       name [DSP_MAX_STRLEN] ;
   ADD_PADDING  (padding, MPCS_ENTRY_PADDING)
} MPCS_Entry ;
```

Fields

ownerProcId	ID of the processor that created the MPCS object.
poolId	ID of the pool used to allocate the MPCS object.
physAddress	Physical address of the MPCS object.
name	Unique system wide name used for identifying the MPCS object.
padding	Padding for alignment, depending on the platform.

Comments

The MPCS entry is created in a region accessible to all processors in the system. This is used to identify and get information about the MPCS objects created in the system.

Constraints

None.

See Also

MPCS_Region

7.2.3 MPCS_Ctrl

This structure defines the control structure required by the MPCS component. It contains information about all MPCS objects shared between the GPP and a specific DSP.

Definition

```
typedef struct MPCS_Ctrl_tag {
   Uint32      isInitialized ;
   Uint32      dspId ;
   Uint32      maxEntries ;
   Uint32      ipsId ;
   Uint32      ipsEventNo ;
   MPCS_Entry * dspAddrEntry ;
   ADD_PADDING  (padding, MPCS_CTRL_PADDING)
   MPCS_ShObj   lockObj ;
} MPCS_Ctrl ;
```

Fields

isInitialized	Indicates whether the MPCS region has been initialized.
dspId	ID of the DSP with which the MPCS region is shared.
maxEntries	Maximum number of MPCS instances supported by the MPCS.
ipsId	ID of the IPS to be used (if any). A value of -1 indicates that no IPS is required by the MPCS.
ipsEventNo	IPS Event number associated with MPCS (if any). A value of -1 indicates that no IPS is required by the MPCS.
dspAddrEntry	Pointer to array in DSP address space of MPCS objects that can be created.
padding	Padding for alignment, depending on the platform.
lockObj	MPCS lock object to provide mutually exclusive access to the MPCS region.

Comments

The MPCS control region is present within a region accessible to all processors in the system. This is used to identify and get information about the MPCS objects created in the system. The region is statically reserved through configuration.

Constraints

None.

See Also

MPCS_Entry

7.2.4 MPCS_ProcObj

This structure defines an object for a single processor used by the Multiprocessor Critical Section object.

Definition

```
typedef struct MPCS_ProcObj_tag {
   Uint32  localLock ;
   Uint16  flag ;
   Uint16  freeObject ;
#if defined (DDSP_PROFILE)
   Uint16  conflicts ;
   Uint16  numCalls  ;
#endif
   Uint32  priority;
} MPCS_ProcObj ;
```

Fields

localLock	Local lock to be used for protection on specific processor. The value stored also depends on the Operating System being used.
flag	Flags indicating whether the shared resource is being claimed by the processor.
freeObject	Contains information about whether the object was allocated internally, and needs to be freed at the time of MPCS delete.
conflicts	Number of conflicts that occurred in MPCS Enter. Defined only when profiling is enabled.
numCalls	Number of calls made to MPCS Enter. Defined only when profiling is enabled.
priority	Priority of the task MPCS task. Defined only when the SWI mode is enabled.

Comments

The MPCS object contains one MPCS_ProcObj object for each of the processors it provides mutually exclusive access to the shared objects from.

Constraints

None.

See Also

MPCS_Obj

7.2.5 MPCS_ShObj

This structure defines the shared Multiprocessor Critical Section object, which is used for protecting a specific critical section between multiple processors. The memory for this object is accessible to the two processors using the MPCS object.

Definition

```
typedef struct MPCS_ShObj_tag {
volatile MPCS_ProcObj    gppMpcsObj ;
        ADD_PADDING (gppPadding, MPCSOBJ_PROC_PADDING)
volatile MPCS_ProcObj    dspMpcsObj ;
        ADD_PADDING (dspPadding, MPCSOBJ_PROC_PADDING)

volatile Uint32          mpcsId ;
volatile Uint16          turn ;
        ADD_PADDING (padding, DSPLINK_16BIT_PADDING)
} MPCS_ShObj ;
```

Fields

gppMpcsObj	MPCS object for the GPP processor.
gppPadding	Padding for alignment, depending on the platform.
dspMpcsObj	MPCS object for the DSP processor.
dspPadding	Padding for alignment, depending on the platform.
mpcsId	MPCS Identifier for this object.
turn	Indicates the processor that owns the turn to enter the critical section.
padding	Padding for alignment, depending on the platform.

Comments

One MPCS object is used to provide mutually exclusive access to any shared region between the GPP and DSP.

Constraints

None.

See Also

MPCS_ProcObj

7.2.6 MPCS_Obj

This structure defines the Multiprocessor Critical Section object, which is used for protecting a specific critical section between multiple processors. This object is not shared between the processors, and the object instance is specific to the process creating the MPCS object.

Definition

```
typedef struct MPCS_Obj_tag {
    MPCS_ShObj *        mpcsObj ;
    SYNC_USR_CsObject * syncCsObj ;
} MPCS_Obj ;

typedef MPCS_Obj * MPCS_Handle ;
```

Fields

mpcsObj	Handle to the MPCS object in user space of the process.
syncCsObj	Handle to the user-side SYNC CS object.

Comments

Each process using the MPCS object opens the MPCS object and gets a handle to the object in its process space.

On the DSP-side, the MPCS_Obj is the same as the MPCS_ShObj.

Constraints

None.

See Also

MPCS_open ()

7.2.7 MPCS_MemInfo

This structure contains memory information for the MPCS component. It is internally used for mapping the MPCS memory into user space.

Definition

```
typedef struct MPCS_MemInfo_tag {
   ProcessorId procId ;
   Uint32      physAddr ;
   Uint32      kernAddr ;
   Uint32      userAddr ;
   Uint32      size ;
} MPCS_MemInfo ;
```

Fields

procId	ID of the processor with which the MPCS region is shared
physAddr	Physical address of the memory region for RingIO
kernAddr	Kernel address of the memory region for RingIO
userAddr	User address of the memory region for RingIO
size	Size of the memory region for RingIO

Comments

This structure is not required on the DSP-side.

Constraints

None.

See Also

MPCS_init ()

7.3 API Definition

7.3.1 _MPCS_init

This function initializes the MPCS component.

Syntax

DSP_STATUS _MPCS_init (ProcessorId procId) ;

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS region is to be shared.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function is called internally by *DSPLINK* and is not exposed to the user applications.

This function performs the following operations:

- Map the statically reserved MPCS region to the user space of the calling process.
- Open the MPCS object within the region object used for protection of the MPCS region.

Constraints

None.

See Also

_MPCS_exit ()

7.3.2 _MPCS_exit

This function finalizes the MPCS component.

Syntax

DSP_STATUS _MPCS_exit (ProcessorId procId) ;

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS region is shared.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.

Comments

This function is called internally by *DSPLINK* and is not exposed to the user applications.

This function performs the following operations:

- Close the MPCS object within the region object used for protection of the MPCS region.
- Unmap the statically reserved MPCS region from the user space of the calling process.

Constraints

None.

See Also

_MPCS_init ()

7.3.3 MPCS_create

This function creates an MPCS object between the calling processor and the processor whose ID is specified.

Syntax

```
DSP_STATUS MPCS_create (ProcessorId    procId,
                        Pstr           name,
                        MPCS_ShObj *   mpcsObj,
                        MPCS_Attrs *   attrs) ;
```

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS object is to be shared.

IN	Pstr	name

System-wide unique name for the MPCS object.

IN OPT	MPCS_ShObj *	mpcsObj

Pointer to the shared MPCS object. If memory for the MPCS object is provided by the user, the MPCS object handle is not NULL. Otherwise, if the memory is to be allocated by the MPCS component, the MPCS object handle can be specified as NULL.

IN	MPCS_attrs *	attrs

Attributes for creation of the MPCS object.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EALREADYEXISTS	The specified MPCS name already exists.
DSP_ERESOURCE	All MPCS entries are currently in use.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Acquire lock for the MPCS region.
- Check if the user specified name already exists in the global MPCS region object. If not, create an entry in the MPCS region.

- If user has not allocated memory for the MPCS object, allocate memory from the user specified pool.
- Initialize the MPCS object.

Constraints

The processor that creates the MPCS object must be the same as the processor that deletes the object.

A call to MPCS_create () must be followed by a call to MPCS_open () to get a handle to the MPCS object in the user space of the calling process.

See Also

MPCS_delete ()

7.3.3.1 OS specific Implementations

PROS:

GPP side:

- Initialize the localLock variable in the MPCS handle with the handle of the global mutex created in MPCS_init function.

DSP side:

- If a global lock does not exist, create it
- Assign the localLock variable in the MPCS handle to the global lock created.

7.3.4 MPCS_delete

This function deletes the specified MPCS object.

Syntax

```
DSP_STATUS MPCS_delete (ProcessorId    procId,
                        Pstr           name) ;
```

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS object is to be shared.

IN	Pstr	name

System-wide unique name for the MPCS object.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_ENOTFOUND	Specified MPCS object name does not exist.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Acquire lock for the MPCS region.
- Entering Task Mode.

  GPP Side:

  - None, Global Lock is deleted in MPCS_exit

  DSP side:

  - Delete the semaphore by calling DSP/BIOS APIs which was taken in MPCS_create Module.
- Check if the user specified name exists in the global MPCS region object.
- Finalize the MPCS object.
- If the MPCS component had allocated memory for the MPCS object, free the memory from the pool used for allocating the object.
- Remove the entry in the MPCS region.
- Release lock for the MPCS region.

Constraints

The processor that creates the MPCS object must be the same as the processor that deletes the object.

See Also

MPCS_create ()

7.3.4.1 OS specific Implementations

PROS:

- Enter Task Mode.

  GPP side:

  - None. The global lock is deleted In MPCS_exit

  DSP side:

  - None. The global mutex object is not deleted.

7.3.5 MPCS_open

This function opens an MPCS object specified by its name and gets a handle to the object.

Syntax

```
DSP_STATUS MPCS_open (ProcessorId    procId,
                      Pstr           name,
                      MPCS_Handle *  mpcsHandle) ;
```

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS object is to be shared.

IN	Pstr	name

System-wide unique name for the MPCS object.

OUT	MPCS_Handle *	mpcsHandle

Location to receive the MPCS object handle, which is valid in the process space of the calling process.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_ENOTFOUND	Specified MPCS object name does not exist.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Call the internal function _MPCS_open () with the user-specified MPCS name and NULL handle for address of shared MPCS object.

Constraints

Every process that needs to use the MPCS object must get a handle to the object by calling this API.

See Also

MPCS_close ()

7.3.6 _MPCS_open

This internal function opens an MPCS object specified by its name and gets a handle to the object. This function allows the user to open an MPCS object by a name with special reserved prefix MPCS_RESV_LOCKNAME indicating that the object is not registered within the MPCS entries table. For such objects, the user already has the pointer to the MPCS shared object in its process space. Every process that needs to use the MPCS object must get a handle to the object by calling this API.

Syntax

```
DSP_STATUS _MPCS_open (ProcessorId    procId,
                       Pstr           name,
                       MPCS_Handle *  mpcsHandle ,
                       MPCS_ShObj *   mpcsShObj) ;
```

Arguments

IN ProcessorId procId

Identifier of the processor with which the MPCS object is to be shared.

IN Pstr name

System-wide unique name for the MPCS object. Specifying the name with prefix as MPCS_RESV_LOCKNAME expects the user to pass the pointer to the MPCS shared object through the mpcsShObj parameter.

OUT MPCS_Handle * mpcsHandle

Location to receive the MPCS object handle, which is valid in the process space of the calling process.

IN OPT MPCS_ShObj * mpcsShObj

Pointer to the MPCS shared object in the caller's process space. This is an optional argument that is provided if the user already has the pointer to the MPCS shared object, and wishes to open the specific MPCS object. This parameter must be specified by the user if the name used is MPCS_RESV_LOCKNAME.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_ENOTFOUND	Specified MPCS object name does not exist.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.
DSP_SFREE	The last close for specified MPCS resulted in it getting closed.

Comments

This function performs the following operations:

- Acquire lock for the MPCS region.
- Check if the user has specified the MPCS object name with prefix MPCS_RESV_LOCKNAME. If yes, use the user-provided mpcsShObj as the address of the shared MPCS object. If not, check if the user specified name exists in the global MPCS region object, and translate the physical address of the shared object to user space.
- Allocate the MPCS object in user process space.
- Get the address of the MPCS object in the user-space of the process through pool address translation, and return this as the handle to MPCS object.
- For GPP-side: On the user-side, create a SYNC CS object for providing protection between multiple processes.
- Release lock for the MPCS region.

Constraints

Every process that needs to use the MPCS object must get a handle to the object by calling this API.

See Also

MPCS_close ()

7.3.7 MPCS_close

This function closes an MPCS object specified by its handle.

Syntax

```
DSP_STATUS MPCS_close (ProcessorId    procId,
                       MPCS_Handle    mpcsHandle) ;
```

Arguments

IN	ProcessorId	procId

Identifier of the processor with which the MPCS object is to be shared.

IN	MPCS_Handle	mpcsHandle

Handle to the MPCS object to be closed.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument
DSP_EMEMORY	Operation failed due to a memory error.
DSP_ENOTFOUND	Specified MPCS object not found.
DSP_EACCESSDENIED	The MPCS component has not been initialized.
DSP_EFAIL	General failure.
DSP_SFREE	The last close for specified MPCS resulted in it getting closed.

Comments

This function performs the following operations:

- Acquire lock for the MPCS region.
- For GPP-side: On the user-side, delete the SYNC CS object used for providing protection between multiple processes.
- Free the MPCS object allocated within the user process space.
- Release lock for the MPCS region.

Constraints

None.

See Also

MPCS_open ()

7.3.8 MPCS_enter

This function enters the critical section specified by the MPCS object.

Syntax

DSP_STATUS MPCS_enter (MPCS_Handle mpcsHandle) ;

Arguments

IN	MPCS_Handle	mpcsHandle

Handle to the MPCS object.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Enter task mode.

  GPP side

  - Wait on the global lock object to take exclusive access to MPCS.

  DSP side

  - Wait on the semaphore to take the exclusive access to MPCS by calling DSP/BIOS APIs.

- Enter SWI mode.

  GPP side

  - If user provides a priority to which priority of current task is to be boosted that is used. If default value is provided, the highest priority which the OS allows is used.

  DSP side

  - Disable the scehduler.

Enter the SWI mode.

- Indicate that the processor needs to use the resource by setting its flag to busy.
- Give away the turn to the other processor.
- Wait while the other process is using the resource and owns the turn.

Constraints

None.

See Also

MPCS_leave ()

7.3.8.1 OS specific Implementations

PROS:

- Enter task mode.

  GPP side

  - Wait on the semaphore to take exclusive access to MPCS by calling SYNC_WaitSEM.

  DSP side

  - Wait on the mutex to take the exclusive access to MPCS by calling DSP/BIOS API LCK_pend().

- Enter SWI mode.

  GPP side- Priority ceiling:

  Use chg_pri() API to change the priority of the current task, this API has tow paramters

  - Priority: Where the priority value is from the build options of dsplinkcfg.pl. the priority is updated in the mpcsHandle->gppMpcsObj.priority.
  - The task id option is given as TSK_SELF.

  DSP side:

  - Disable the Software interrupts by calling DSP/BIOS API SWI_disable().

7.3.9 MPCS_leave

This function leaves the critical section specified by the MPCS object.

Syntax

DSP_STATUS MPCS_leave (MPCS_Handle mpcsHandle) ;

Arguments

IN	MPCS_Handle	mpcsHandle

Handle to the MPCS object.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Indicate that the processor no longer needs to use the resource by resetting its flag to free.
- Enter Task mode.

  GPP side

  - Signal mutex to be available for other tasks by calling OS specific APIs.

  DSP side:

  - Signal mutex to be available for other tasks by calling DSP/BIOS APIs.

- Enter SWI mode.

  GPP side:

  - Restore back the priority of the calling task to its original value by calling chg_pri() API.

  DSP side:

  - To enable all the software interrupts by calling DSP/BIOS APIs.
- Release the user-side SYNC CS.

Constraints

None.

See Also

MPCS_enter ()

7.3.9.1 OS specific Implementations

PROS:

GPP side

- Signal mutex to be available for other tasks by calling SYNC_SignalSEM. The parameter to this function is mpcsHandle->dspMpcsObj.localLock. which was update in MPCS_enter Module.

DSP side:

- Signal mutex to be available for other tasks by calling LCK_post().

- Enter SWI mode.

  GPP side:

  - Priority Ceiling: Restore back the priority of the calling task to its original value by calling chg_pri() API.

  DSP side:

  - To enable all the software interrupts, call SWI_enable().

8 Internal Discussions

8.1 IDM module

The ID Manager (IDM) module provides the functionality for acquiring and releasing IDs for different objects identified based on a unique key provided by the caller. The range of supported IDs for an IDM object is specified by the user while creating the IDM object. Each ID within an IDM object is also associated with an idKey of string type. If the idKey specified while acquiring an ID is already present, the ID associated with this string idKey is returned. Otherwise a new ID is acquired and associated with this idKey. While releasing the ID, reference count is used to free the ID only when all references to it are closed.

The IDM component is present on the kernel-side for OSes such as Linux and is part of the GEN module.

8.1.1 Constants & Enumerations

8.1.1.1 MAX_IDM_OBJECTS

This constant denotes the maximum number of objects supported by the IDM component.

Definition

```
#define MAX_IDM_OBJECTS      32
```

Comments

None.

Constraints

None.

See Also

IDM_State

8.1.1.2 IDM_INVALID_KEY

This constant denotes an invalid key used for identifying the IDM object.

Definition

```
#define IDM_INVALID_KEY      (Uint32) -1
```

Comments

None.

Constraints

None.

See Also

IDM_create ()

8.1.2 Typedefs & Data Structures

8.1.2.1 IDM_Attrs

This structure defines the attributes for creation of the IDM object.

Definition

```
typedef struct IDM_Attrs_tag {
   Uint16  baseId ;
   Uint16  maxIds ;
} IDM_Attrs ;
```

Fields

baseId	Base ID supported by this IDM object.
maxIds	Maximum number of IDs supported by this IDM objects.

Comments

The IDM attributes are specified by the user while creating the IDM object. They define the characteristics of the object including the range of IDs supported.

Constraints

None.

See Also

IDM_create ()

8.1.2.2 IDM_Id

This structure defines the IDM ID identified by a unique ID key per IDM object.

Definition

```
typedef struct IDM_Id_tag {
   Char8    idKey [DSP_MAX_STRLEN] ;
   Uint16   refCount ;
} IDM_Id ;
```

Fields

idKey	ID key associated with the ID to be returned.
refCount	Reference count indicating the number of clients that have acquired this ID.

Comments

The IDM object contains all information required to identify and perform ID acquire and release functionality.

Constraints

None.

See Also

IDM_create ()

8.1.2.3 IDM_Object

This structure defines the IDM object identified by a unique key.

Definition

```
typedef struct IDM_Object_tag {
   Uint32    key ;
   Uint16    baseId ;
   Uint16    maxIds ;
   IDM_Id *  idArray ;
} IDM_Object ;
```

Fields

key	Unique key identifying the IDM object.
baseId	Base ID supported by this IDM object.
maxIds	Maximum number of IDs supported by this IDM objects.
idArray	Dynamically allocated array of ID objects indicating acquired and released IDs. The value of the ID is its index in the array added to the base ID.

Comments

The IDM object contains all information required to identify and perform ID acquire and release functionality.

Constraints

None.

See Also

IDM_create ()

8.1.2.4 IDM_State

This structure defines the state object for the IDM component.

Definition

```
typedef struct IDM_State_tag {
   Bool       isInitialized ;
   IDM_Object  idmObjs [MAX_IDM_OBJECTS] ;
} IDM_State ;
```

Fields

isInitialized	Indicates whether the IDM component is initialized.
idmObjs	Array of IDM objects.

Comments

The IDM state object contains the global information required for the IDM component.

Constraints

None.

See Also

IDM_init ()

8.1.3 API Definition

8.1.3.1 IDM_init

This function initializes the IDM component.

Syntax

DSP_STATUS IDM_init (Void) ;

Arguments

None.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

This function initializes the global IDM state object.

Constraints

None.

See Also

IDM_exit ()

8.1.3.2 IDM_exit

This function finalizes the IDM component.

Syntax

DSP_STATUS IDM_exit (Void) ;

Arguments

None.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

This function finalizes the global IDM state object.

Constraints

None.

See Also

IDM_init ()

8.1.3.3 IDM_create

This function creates an IDM object identified based on a unique key specified by the user.

Syntax

DSP_STATUS IDM_create (Uint32 key, IDM_Attrs * attrs) ;

Arguments

IN	Uint32	key
	Unique key used to identify the IDM object created.	
IN	IDM_Attrs *	attrs
	Attributes for creation of the IDM object.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_ERESOURCE	All IDM objects have been used.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Get a free slot in the IDM objects array in the state object. The free slot is identified based on key value of IDM_INVALID_KEY.
- Initialize the object based on attributes provided by the user and set its key to the user-provided key.
- Allocate the ID array in the object of size maxIds and initialize the idKey and refCount for all IDs to indicate that they are free to be acquired.

Constraints

None.

See Also

IDM_delete ()

8.1.3.4 IDM_delete

This function deletes an IDM object identified based on a unique key specified by the user.

Syntax

DSP_STATUS IDM_delete (Uint32 key) ;

Arguments

IN	Uint32	key

Unique key used to identify the IDM object being deleted.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	Object corresponding to specified key not found.
DSP_EMEMORY	Operation failed due to memory error.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Identify the object to be deleted in the IDM objects array in the state object based on the specified key.
- Free the ID array in the object.
- Reset all other fields in the object and reset the key to IDM_INVALID_KEY to indicate the freed object.

Constraints

None.

See Also

IDM_create ()

8.1.3.5 IDM_acquireId

This function acquires a free ID for the specified IDM object.

Syntax

DSP_STATUS IDM_acquireId (Uint32 key, Pstr idKey, Uint32 * id) ;

Arguments

IN	Uint32	key
	Unique key used to identify the IDM object.	
IN	Pstr	idKey
	String key to associate with the ID to be returned. If the specified idKey already exists for the IDM object, the id for this idKey is returned and a reference count incremented. If the idKey does not exist, a new id is acquired and returned for this idKey.	
OUT	Uint32 *	id
	Location to receive the ID being acquired.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_SEXISTS	The specified idKey already exists and its ID is returned.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	Object corresponding to specified key not found.
DSP_ERESOURCE	All IDs for this object have been consumed.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Identify the object in the IDM objects array in the state object based on the specified key.
- Check if the specified idKey already exists. If found, increase the reference count for the ID and return its value (index in the array + baseId for the object).
- If the idKey does not exist, find a free ID within the ID array based on the idKey. Initialize the idKey to specified value, set its reference count to 1 and return the value of the ID.
- If no free ID is found (all IDs have been used up), return error.

Constraints

None.

See Also

IDM_releaseId ()

8.1.3.6 IDM_releaseId

This function releases the specified ID for the specified IDM object.

Syntax

DSP_STATUS IDM_releaseId (Uint32 key, Uint32 id) ;

Arguments

IN	Uint32	key
	Unique key used to identify the IDM object.	
IN	Uint32	id
	ID to be released.	

Return Value

DSP_SOK	Operation successfully completed.
DSP_SFREE	The last release for specified ID resulted in it getting freed.
DSP_EINVALIDARG	Invalid argument.
DSP_ENOTFOUND	Object corresponding to specified key not found.
DSP_EFAIL	General failure.

Comments

This function performs the following operations:

- Identify the object in the IDM objects array in the state object based on the specified key.
- Calculate the index of the ID within the array (Specified ID – baseId for the object).
- Decrement the reference count of the ID and check if it needs to be released.
- If the reference count reaches zero, release the ID in the array by resetting its idKey.

Constraints

None.

See Also

IDM_releaseId ()

8.2 SYNC_USR module

The SYNC_USR module provides the functionality for user-side protection between multiple processes. It gives a standard set of APIs across multiple OSes such as Linux and PrOS. On Linux, it uses the System V semaphores, and on PrOS, it uses the OSAL implementation using mutex.

8.2.1 Constants & Enumerations

None.

8.2.2 Typedefs & Data Structures

8.2.2.1 SYNC_USR_CsObject

This structure defines the user-side critical section object. The definition of the structure is OS-specific.

Definition

typedef struct SYNC_USR_CsObject_tag SYNC_USR_CsObject ;

Comments

This object contains all information used for identifying and operating on the user-side critical section object.

For Linux, the definition is:

```
struct SYNC_USR_CsObject_tag {
   int    osSemId ;
   Uint32 semId ;
   Uint32 refCount ;
} ;
```

For PrOS, the OSAL SYNC CS object is directly used. The definition is:

typedef SyncCsObject SYNC_USR_CsObject_tag ;

Constraints

None.

See Also

_SYNC_USR_createCS ()

8.2.2.2 SYNC_USR_State

This structure defines the state object for the SYNC_USR component.

Definition

```
typedef struct IDM_State_tag {
   Bool            isInitialized ;
   SYNC_USR_CsObject csObjs [MAX_SYNC_CS] ;
} IDM_State ;
```

Fields

isInitialized	Indicates whether the SYNC_USR component is initialized.

idmObjs	Array of pointers to SYNC_USR_CsObject objects.

Comments

The SYNC_USR state object contains the global information required for the SYNC_USR component.

Constraints

None.

See Also

_SYNC_USR_init ()

8.2.3 API Definition

8.2.3.1 _SYNC_USR_init

This function initializes the SYNC_USR component.

Syntax

DSP_STATUS _SYNC_USR_init () ;

Arguments

None.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system.

For PrOS, this function performs the following operations:

- Create an IDM object for the CS objects based on a key used for the SYNC CS component.
- Initialize the SYNC_USR state object.

For Linux, this function performs the following operations:

- Initialize the SYNC_USR state object.
- Get a unique key value using the ftok call. The key is based on a filename (/dev/dsplink) and a string indicating usage for Critical Section objects.
- Create a semaphore set based on the unique key using the semget API.
- Create an IDM object for the CS objects based on a key used for the SYNC CS component.

Constraints

None.

See Also

_SYNC_USR_exit ()

8.2.3.2 _SYNC_USR_exit

This function finalizes the SYNC_USR component.

Syntax

DSP_STATUS _SYNC_USR_exit () ;

Arguments

None.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system.

For PrOS, this function performs the following operations:

- Finalize the SYNC_USR state object.
- Delete the IDM object created for the CS objects based on the key used for the SYNC CS component.

For Linux, this function performs the following operations:

- Delete the IDM object created for the CS objects based on the key used for the SYNC CS component.
- Get a unique key value using the ftok call. The key is based on a filename (/dev/dsplink) and a string indicating usage for Critical Section objects.
- Get the semaphore ID based on the unique key using the semget API.
- Delete the semaphore set corresponding to the semaphore ID using the semctl API with IPC_RMID command.

Constraints

None.

See Also

_SYNC_USR_init ()

8.2.3.3 _SYNC_USR_createCS

This function creates the Critical section object.

Syntax

DSP_STATUS
_SYNC_USR_createCS (Pstr idKey, SYNC_USR_CsObject ** csObj) ;

Arguments

IN	Pstr	idKey

String key to identify the CS being created. If a CS with the specified key has been created, a handle to the same CS is returned to provide protection between multiple processes. If a CS corresponding to specified key does not exist, a new object is created and returned to the user.

OUT	SYNC_USR_CsObject **	csObj

Location to receive the pointer to created critical section object.

Return Value

DSP_SOK	Operation successfully completed.
DSP_SEXISTS	Semaphore corresponding to the specified idKey already exists, and is returned.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EPOINTER	Invalid pointer passed.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system. For PrOS, this function performs the following operations:

- Check if a new object is being created, or a handle to existing object is to be returned by acquiring the ID corresponding to the specified idKey.
- If the ID already exists (indicated by return code DSP_SEXISTS), return the handle to existing CS object at the acquired ID.
- Otherwise create a new semaphore by making a call to the OSAL SYNC CS API and set its handle within the state object for the acquired ID.

For Linux, this function performs the following operations:

- Get the ID corresponding to the specified idKey by making a call to the IDM component to acquire the ID. Return code of DSP_SEXISTS indicates that the semaphore is being created for the first time.
- Get a unique key value using the ftok call. The key is based on a filename (/dev/dsplink) and a string indicating usage for Critical Section objects.
- Get the semaphore ID based on the unique key using the semget API.

- If the semaphore is being created for the first time, initialize the semaphore for the unique ID to 1 as initially available.
- Check if the semaphore has been already created in this process by checking for a set handle in the csObjs array in the state object. If yes, return the same handle after incrementing the reference count for the semaphore.
- If the semaphore has not been created in this process, allocate memory for a new SYNC_USR_CsObject. Initialize the semaphore fields.

Constraints

None.

See Also

_SYNC_USR_deleteCS ()

8.2.3.4 _SYNC_USR_deleteCS

This function creates the Critical section object.

Syntax

DSP_STATUS _SYNC_USR_deleteCS (SYNC_USR_CsObject ** csObj) ;

Arguments

IN OUT SYNC_USR_CsObject ** csObj

Address of the location containing the pointer to the critical section object. The pointer may be reset on successful return from the function, based on whether the object was deleted.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system. For PrOS, this function performs the following operations:

- Search for the specified csObj in array of existing CS objects to get its ID.
- Release the ID for the CS object from kernel-side ID manager based on the key used for the SYNC CS component and this ID.
- If the status of IDM_release () indicates that the last reference to the semaphore is being closed (return status of DSP_SFREE), make a call to the OSAL SYNC CS API to delete the CS object. Free the memory for the SYNC_USR_CsObject and reset the user pointer. Reset the handle for the semaphore object within the state object.

For Linux, this function performs the following operations:

- Release the ID for the CS object from kernel-side ID manager based on the key used for the SYNC CS component and ID stored within the SYNC_USR_CsObject.
- Decrement the reference count for the semaphore. If the reference count reaches zero, free the memory for the SYNC_USR_CsObject and reset the user pointer. Reset the handle for the semaphore object within the state object.

Constraints

None.

See Also

_SYNC_USR_createCS ()

8.2.3.5 _SYNC_USR_enterCS

This function enters the critical section that is passed as an argument to it. After successful return of this function no other process can enter until this process exits the CS.

Syntax

DSP_STATUS _SYNC_USR_enterCS (SYNC_USR_CsObject * csObj) ;

Arguments

IN	SYNC_USR_CsObject *	csObj

Pointer to the critical section object to be entered.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system. For PrOS, this function performs the following operations:

- Make a call to the OSAL SYNC CS API to enter the CS.

For Linux, this function performs the following operations:

- Enter the critical section using the semop API.

Constraints

None.

See Also

_SYNC_USR_leaveCS ()

8.2.3.6 _SYNC_USR_leaveCS

This function makes the critical section available for other processes to enter.

Syntax

DSP_STATUS _SYNC_USR_leaveCS (SYNC_USR_CsObject * csObj) ;

Arguments

IN	SYNC_USR_CsObject *	csObj

Pointer to the critical section object to be released.

Return Value

DSP_SOK	Operation successfully completed.
DSP_EINVALIDARG	Invalid argument.
DSP_EFAIL	General failure.

Comments

The implementation of this function varies depending on the operating system. For PrOS, this function performs the following operations:

- Make a call to the OSAL SYNC CS API to leave the CS.

For Linux, this function performs the following operations:

- Leave the critical section using the semop API.

Constraints

None.

See Also

_SYNC_USR_leaveCS ()

8.3 MPCS API flow

It is assumed that the usage of the api's is in correct sequence i.e. MPCS_enter is called after MPCS_open etc. Checks have been placed to ensure pre-conditions for most API's are true at the time of calling the API.

Consider the following case:

Invoking MPCS_enter() twice in a sequence shows different behaviors on GPP/DSP.

In order to verify that the same process does not call MPCS_enter in a sequence without calling MPCS_leave, the following changes need to be made on the GPP side. The SYNC module needs to be updated to capture book keeping information about the process and thread id (in case of Linux) and task id(in case of PROS). This book keeping information has to be updated every time a process calls MPCS_enter and MPCS_leave.

It is an overkill to do this for a boundary condition. It is assumed that the user of the API does not call MPCS_enter twice in a sequence without calling MPCS_leave.