DSP/BIOS™ LINK

Pool

LNK 082 DES

Version 1.30

This page has been intentionally left blank.

IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

This page has been intentionally left blank.

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

1.1 Purpose & Scope

This document describes the design of shared memory allocator in DSP/BIOS™ LINK. This component shall be used to allocate buffers / messages to be transferred across processors (GPP and DSP).

This document describes the design of shared memory allocator for the GPP and DSP.

The document is targeted at the developers of DSP/BIOS™ LINK. Customers can also use it to get a better understanding of the component.

1.2 Text Conventions

⚑	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

1.3 Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
Client	Refers to a process/ thread/ task in an operating system that uses DSP/BIOS™ LINK API. It is used to ensure that description is free from the specifics of 'unit of execution' for a particular OS.

1.4 References

None.

1.5 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.

The POOL will be used for allocating buffers.

This document provides a high-level description of the POOL design.

2 Requirements

The requirements of the POOL are:

1. It shall provide uniform API's for allocating buffers.

3 Assumptions

None.

4 Constraints

None.

5 High Level Design

5.1 POOL

POOL Module shall manage different buffer pools, each can have different buffer allocation techniques. It shall allow different buffer pools to be managed simultaneously. It shall provide uniform APIs for pool operations. Internal operations specific to a pool are opaque for the client side application.

It shall be scaleable, so that in case of OSes like Linux, it will map the buffers from user address space to kernel address space and vice-versa. For other OSes, it shall simple skip the mapping/unmapping logic.

In case of OSes like Linux, it shall provide an interface, using which buffers can be allocated/deallocated directly in kernel context, i.e. code executing in kernel context can directly alloc/free buffers.

It shall provide Initialize and Finalize functions, which shall initialize/finalize all buffer pools by calling Initialize and Finalize function of the buffer pools (internal to POOL). So that all buffer pools are initialized/finalized at the time of initializing/finalizing of DSPLINK. Each pool shall implement its own initialize/finalize logic.

After initialization, calling the Open function shall create the buffer pools for the given pool ID. For OSes like Linux, Open function shall return information specific to mapping/unmapping buffers between user and kernel address space.

Closing a specific buffer pool shall destroy the buffers inside the buffer pool and make the buffer pool unusable.

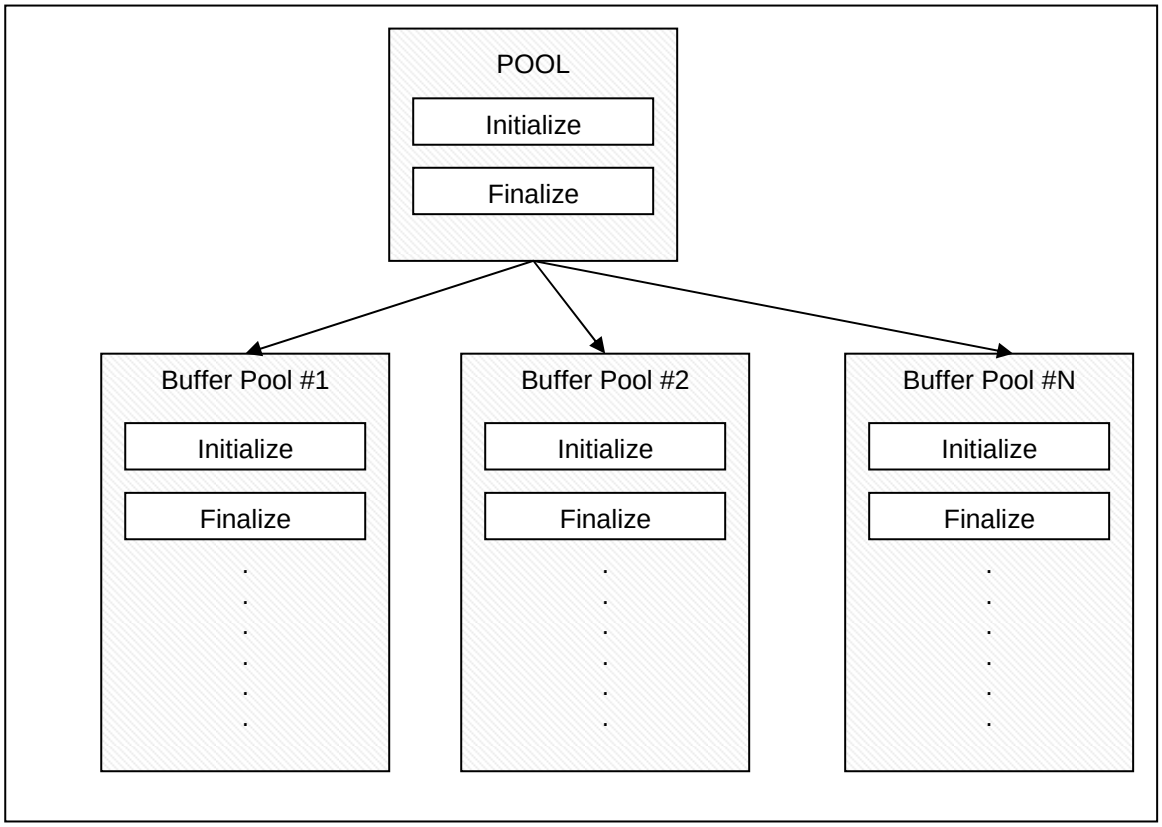


Figure 1. POOL Architecture.

The above figure depicts the POOL architecture.

5.2 LDRV POOL

This interface provides an access to the buffer pools. This reduces the overhead of calling the POOL APIs for buffer pool operations. For OSes like Linux, kernel context code cans directly allocate/deallocate buffers by using this interface. These interface will be define at kernel level only.

All calls to POOL APIs are actually translated to this interface. So additional mapping/unmapping logic is required between POOL API alloc/free and POOL Interface alloc/free. These logics should be scaleable, i.e. it shall be removed for OSes, which does not have user/kernel separation.

All buffer pool shall expose its own interface, which shall be plugged into an array of POOL interface type. POOL count shall be maintained to reflect the number of buffer pools present at any given time. Each buffer pool is provided unique POOL ID, which shall be used for translating POOL API calls to POOL Interface calls. All information required for buffer pool creation is also stored in the pool interface array.

POOL is the basic backbone of ZCPY (zero copy) mechanism of transferring information in DSPLINK. POOL can be implemented using shared memory on devices like Davinci. For devices like DM642 where the only selected 4MB of DSP memory can be accessed, which may not fit the requirement for big sized POOL (Also the 4MB slot has very slow read and write operations). In this case, all control information related to POOL are accessed through 4MB slot, but buffers (can be very large) are kept in local physical contiguous memory on both GPP and DSP. These local copies are replica of peer's copy. These copies are kept in sync with the help of DMA engine. The implementation of pool can be via shared memory (SMA POOL) or synchronized using DMA(SYNC POOL).

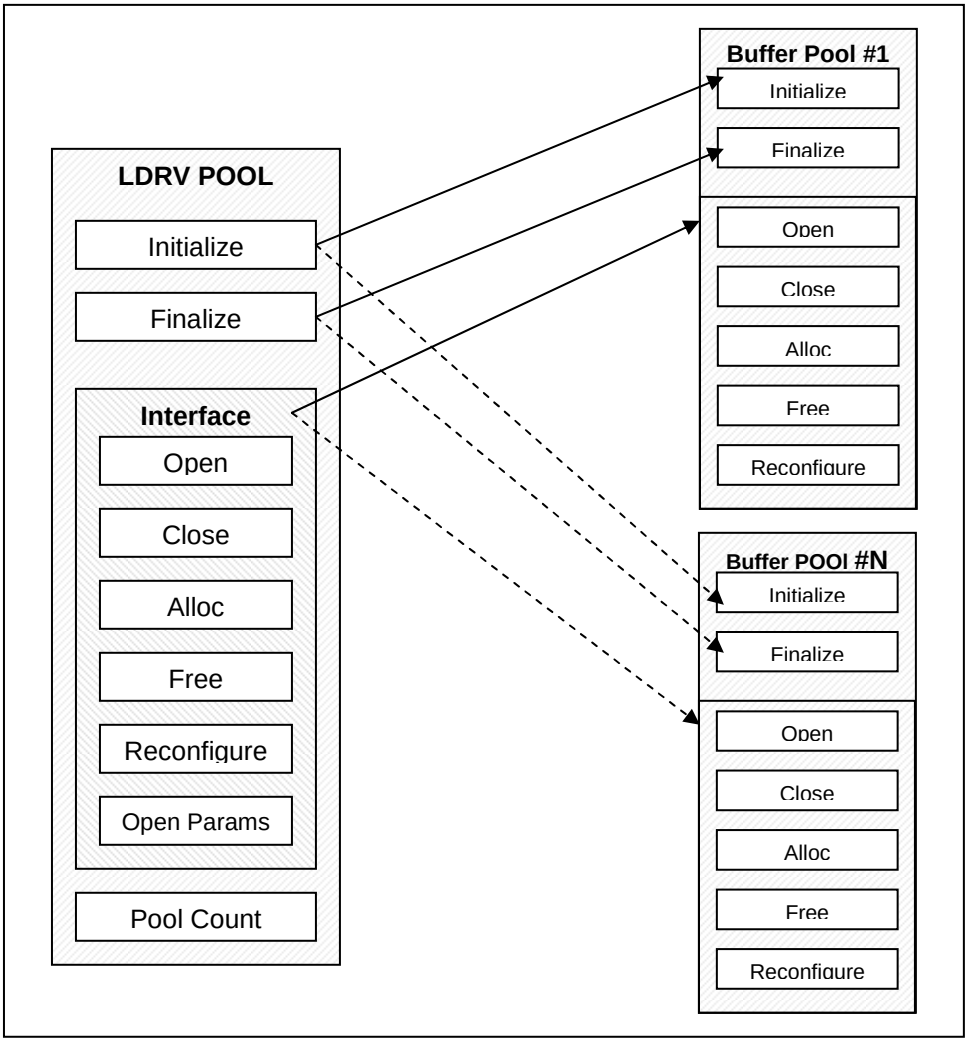


Figure 2. POOL Interface Architecture.

6 POOL API

6.1 Typedefs & Data Structures

6.1.1 Pool_ AddrInfo

This structure defines the buffer information structure for the Pool. This structure defines the configuration attributes required for mapping/unmapping the buffers.

Definition

```
typedef struct POOL_AddrInfo_tag {
    Uint32  addr [MAX_ADDR_TYPES] ;
    Uint32  size ;
} POOL_AddrInfo;
```

Fields

addr	Array of addresses containing the same address in different address spaces
size	Size of memory block in bytes

Comments

This structure is used for retrieving information about a buffer allocated from the pool.

Constraints

None.

See Also

None.

6.2 API

6.2.1 POOL_open

This function opens a specific pool referenced by the pool Id.

Syntax

```
DSP_STATUS POOL_open (Uint16 poolId, Pvoid params) ;
```

Arguments

IN	Uint16	PoolId

Pool Identification number.

IN	Pvoid	params

POOL Open specific parameters.

Return Value

DSP_SOK	Operation completed successfully.
DSP_SALREADYOPENED	The specified POOL has already been opened.
DSP_EACCESSDENIED	Access to the DSP is denied.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

When any client wishes to use a specific pool, it first needs to open the pool calling this API specifying the required pool ID. The pool ID corresponds to the index of the configured pool within the pool table in the CFG_<PLATFORM>.c file.

Every process that needs to use the specific pool must indicate this to *DSPLINK* by making a call to this API. Only the first call for opening a pool makes use of the passed parameters. For calls from other applications/processes to open the same pool, the pool configuration parameters, if provided, are ignored. Applications can even pass NULL as the pool parameters if they are aware that the pool has already been opened by another process.

This API carries out all mappings and initialization required to be able to use the specified pool 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

All applications using a specific pool ID must ensure that all their requirements are met with the pool configuration parameters provided by the first caller to POOL_open ().

See Also

POOL_close ()

6.2.2 POOL_close

This function closes a specific pool whose pool id is provided.

Syntax

```
DSP_STATUS POOL_close (Uint16 poolId) ;
```

Arguments

IN	Uint16	PoolId

Pool Identification number.

Return Value

DSP_SOK	Operation completed successfully.
DSP_SCLOSED	The final process has closed the specified POOL.
DSP_EINVALIDARG	Invalid argument.
DSP_EACCESSDENIED	The POOL was not opened.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

Any applications/processes that no longer need to use the opened pool call must call this API with the specific pool ID. The pool ID corresponds to the index of the configured pool within the pool table in the CFG_<PLATFORM>.c file. Once this API has been called, the process cannot make any *DSPLINK* API calls that make use of the pool with this ID, for example `MSGQ_alloc ()`, `POOL_alloc ()` etc.

This API removes out all mappings and finalizes the specific POOL for the calling process. This API can be successfully called once by every process in the system. However, if `POOL_open ()` was not called in this process for this pool ID, `POOL_close ()` must not be called for this pool ID. 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

`POOL_open ()`

6.2.3 POOL_alloc

This function allocates a buffer of the specified size from a pool.

Syntax

```
DSP_STATUS POOL_alloc (Uint16 poolId, Uint16 size, Pvoid * bufPtr) ;
```

Arguments

IN	Uint16	PoolId

Pool Identification number.

IN	Uint16	size

Size of buffer to be allocated.

OUT	Pvoid *	bufPtr

Location to receive a pointer to the allocated buffer.

Return Value

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

Comments

This function will call DRV_Invoke with specific command ID.

Constraints

None.

See Also

POOL_free ()

6.2.4 POOL_free

This function frees a buffer into the specified pool.

Syntax

```
DSP_STATUS POOL_free (Uint16 poolId, Uint16 size, Pvoid bufPtr) ;
```

Arguments

IN	Uint16	PoolId

Pool Identification number.

IN	Uint16	size

Size of buffer to be freed.

IN	Pvoid	bufPtr

Pointer to the buffer to be freed.

Return Value

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

Comments

This function will call DRV_Invoke with specific command ID.

Constraints

None.

See Also

POOL_alloc ()

6.2.5 POOL_writeback

This function writes the content of GPP buffer into DSP buffer (with offset in sync). On platforms like Davinci which are based on shared memory, this function returns success. On PCI based platforms this internally calls the DMA engine which keeps the synchronized pool (SYNC POOL) updated. This function "writebacks" the content so that the other processor can view latest copy of synchronized pool contents.

Syntax

```
DSP_STATUS POOL_writeback (IN Uint16 poolId,
                           IN Pvoid buf,
                           IN Uint32 size)
```

Arguments

IN	Uint16	PoolId

Pool Identification number.

IN	Pvoid	bufPtr

Pointer to the buffer to be written back.

IN	Uint32	size

Size of buffer to be written back.

Return Value

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

Comments

This function will call DRV_Invoke with specific command ID.

Constraints

None.

See Also

POOL_invalidate ()

6.2.6 POOL_invalidate

This function This function invalidates the content of the buffer.

On platforms like Davinci which are based on shared memory, this function returns success. On PCI based platforms this internally calls the DMA engine which keeps the synchronized pool (SYNC POOL) updated. This function "invalidates" the pool content so that the processor can view the latest copy of synchronized pool contents.

Syntax

```
DSP_STATUS POOL_invalidate (IN Uint16 poolId,
                            IN Pvoid buf,
                            IN Uint32 size)
```

Arguments

`IN` `Uint16` `PoolId`

Pool Identification number.

`IN` `Pvoid` `bufPtr`

Pointer to the buffer to be invalidated.

`IN` `Uint32` `size`

Size of buffer to be written back.

Return Value

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

Comments

This function will call DRV_Invoke with specific command ID.

Constraints

None.

See Also

`POOL_writeback ()`

6.2.7 POOL_translateAddr

This function translates addresses between two address spaces for a buffer that was allocated from the pool.

Syntax

```
DSP_STATUS
POOL_translateAddr (IN  Uint16          poolId,
                    OUT Pvoid *         dstAddr,
                    IN  AddrType        dstAddrType,
                    IN  Pvoid           srcAddr,
                    IN  AddrType        srcAddrType) ;
```

Arguments

IN	Uint16	PoolId	
	Pool Identification number.		
OUT	Pvoid *	dstAddr	
	Location to receive the translated address.		
IN	AddrType	dstAddrType	
	Type of address to be translated to.		
OUT	Pvoid	srcAddr	
	Address of the buffer to be translated.		
IN	AddrType	dstAddrType	
	Type of address to be translated to.		

Return Value

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

Comments

This function will call DRV_Invoke with specific command ID.

Constraints

None.

See Also

None.

7 SMA POOL

7.1 Typedefs & Data Structures

7.1.1 POOL_AddrXltFlag

This enum defines direction of buffer translation.

Definition

```
typedef enum  {
    USR_TO_KNL = (AddrType_Usr | (AddrType_Knl << 8u)),
    USR_TO_PHY = (AddrType_Usr | (AddrType_Phy << 8u)),
    USR_TO_DSP = (AddrType_Usr | (AddrType_Dsp << 8u)),
    PHY_TO_USR = (AddrType_Phy | (AddrType_Usr << 8u)),
    PHY_TO_KNL = (AddrType_Phy | (AddrType_Knl << 8u)),
    PHY_TO_DSP = (AddrType_Phy | (AddrType_Dsp << 8u)),
    KNL_TO_USR = (AddrType_Knl | (AddrType_Usr << 8u)),
    KNL_TO_PHY = (AddrType_Knl | (AddrType_Phy << 8u)),
    KNL_TO_DSP = (AddrType_Knl | (AddrType_Dsp << 8u)),
    DSP_TO_USR = (AddrType_Dsp | (AddrType_Usr << 8u)),
    DSP_TO_PHY = (AddrType_Dsp | (AddrType_Phy << 8u)),
    DSP_TO_KNL = (AddrType_Dsp | (AddrType_Knl << 8u))
} POOL_AddrXltFlag ;
```

Fields

USR_TO_KRNL	User to kernel address translation.
KRNL_TO_USR	Kernel to user address translation.
USR_TO_DSP	User to DSP address translation.
PHY_TO_USR	Physical to user address translation.
PHY_TO_KNL	Physical to kernel address translation.
PHY_TO_DSP	Physical to DSP address translation.
KNL_TO_USR	Kernel to user address translation.
KNL_TO_PHY	Kernel to physical address translation.
KNL_TO_DSP	Kernel to DSP address translation.
DSP_TO_USR	DSP to user address translation.
DSP_TO_PHY	DSP to physical address translation.
DSP_TO_KNL	DSP to kernel address translation.

Comments

This enum will be used at IOCTL level for mapping and unmapping buffers.

Constraints

None.

See Also

`POOL_translateAddr ()`

8 LDRV POOL

8.1 Typedefs & Data Structures

8.1.1 FnPoolInfInitialize

This type defines the function, which initialize the plugged memory allocator.

Syntax

```
typedef DSP_STATUS (*FnPoolInfInitialize) (OUT Pvoid * object) ;
```

Arguments

`OUT` `Pvoid *` `object`

Pointer to the object to be initialized.

Return Value

`DSP_SOK`	The plugged memory allocator component has been successfully initialized.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EFAIL`	General failure.

Comments

None.

Constraints

None.

See Also

`FnPoolInfFinalize ()`

8.1.2 FnPoolInfFinalize

This type defines the function, which finalize the plugged memory allocator

Syntax

```
typedef DSP_STATUS (*FnPoolInfFinalize) (IN Pvoid object) ;
```

Arguments

IN	Pvoid	object

Pointer to the object to be finalized.

Return Value

DSP_SOK	The plugged allocator component has been successfully finalized.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

None.

Constraints

None.

See Also

FnPoolInfInitialize ()

8.1.3 FnPoolInfOpen

This type defines the function, which creates the plugged memory allocator.

Syntax

```
typedef DSP_STATUS (*FnPoolInfOpen) (
                                    IN Pvoid object,
                                    IN POOL_OpenParams * poolOpenParams)
;
```

Arguments

`IN` `Pvoid` `object`

Pointer to the internal allocator object.

`IN` `POOL_OpenParams *` `poolOpenParams`

An argument for creating POOL and plugged memory allocator.

Return Value

`DSP_SOK`	Plugged memory allocator is successfully created.
`DSP_EINVALIDARG`	Invalid argument.
`DSP_EMEMORY`	Operation failed due to a memory error.
`DSP_EFAIL`	General failure.

Comments

None.

Constraints

None.

See Also

`FnPoolInfClose ()`

8.1.4 FnPoolInfClose

This type defines the function, which destroys plugged memory allocator.

Syntax

```
typedef DSP_STATUS (*FnPoolInfClose) (IN Pvoid object) ;
```

Arguments

IN	Pvoid	object

Pointer to the internal allocator object.

Return Value

DSP_SOK	Plugged memory allocator are successfully deleted.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

None.

Constraints

None.

See Also

FnPoolInfOpen ()

8.1.5 FnPoolInfAlloc

This type defines the function, which allocates a buffer, and returns the pointer to the user.

Syntax

```
typedef DSP_STATUS (*FnPoolInfAlloc) (IN  Pvoid    object,
                                      IN  Uint16   size,
                                      OUT Pvoid * bufPtr) ;
```

Arguments

IN	Pvoid	object

Pointer to the internal allocator object.

IN	Uint16	size

Size of the buffer to be allocated.

OUT	Pvoid *	bufPtr

Location to receive the allocated buffer.

Return Value

DSP_SOK	The buffer has been successfully allocated.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

None.

Constraints

None.

See Also

FnPoolInfFree ()

8.1.6 FnPoolInfFree

This type defines the function, which frees a buffer.

Syntax

```
typedef DSP_STATUS (*FnPoolInfFree) (IN Pvoid    object,
                                     IN Uint16   size,
                                     OUT Pvoid   bufPtr) ;
```

Arguments

IN	Pvoid	object

Pointer to the internal allocator object.

IN	Uint16	size

Size of the buffer to be freed.

OUT	Pvoid	bufPtr

Location to the buffer.

Return Value

DSP_SOK	The buffer has been successfully freed.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

None.

Constraints

None.

See Also

FnPoolInfAlloc ()

8.1.7 FnPoolInfReconfigure

This type defines the function, which reconfigure the plugged memory allocator.

Syntax

```
typedef DSP_STATUS (*FnPoolInfReconfigure) (IN Pvoid object,
                                            IN Pvoid args) ;
```

Arguments

IN	Pvoid	object

Pointer to the internal allocator object.

IN	Pvoid	args

Argument for reconfiguring the plugged memory allocator.

Return Value

DSP_SOK	Internal allocator is successfully reconfigured.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

None.

Constraints

None.

See Also

None.

8.1.8 POOL_Interface

This structure defines the control attributes required for both processors to work with buffer pools.

Definition

```
typedef struct POOL_Interface_tag {
    FnPoolOpen        open        ;
    FnPoolClose       close       ;
    FnPoolAlloc       alloc       ;
    FnPoolFree        free        ;
    FnPoolReconfigure reconfigure ;
    FnPoolWriteback   writeback   ;
    FnPoolInvalidate  invalidate  ;
    FnPoolXltBuf      xltBuf      ;
} POOL_Interface ;
```

Fields

`open`	Function pointer to the plugged allocator’s open function.
`alloc`	Function pointer to the plugged allocator’s close function.
`free`	Function pointer to the plugged allocator’s free function.
`reconfigure`	Function pointers to the plugged allocator’s reconfigure function.
`writeback`	Function pointer to the plugged pool's writeback function.
`invalidate`	Function pointer to the plugged pool's invalidate function.
`xltBuf`	Function pointer to the plugged pool's xltBuf function.

Comments

This structure will be initialized by GPP in LDRV Initialize section.

Constraints

None.

See Also

None.

8.1.9 POOL_OpenParams

This structure defines the allocator open specific parameters.

Definition

```
struct POOL_OpenParams_tag {
    Pvoid   params ;
    Uint32  physAddr ;
    Uint32  virtAddr ;
    Uint32  dspAddr ;
    Uint32  size ;
} ;;
```

Fields

`params`	Pointer to the user provided parameters.
`physAddr`	Physical address of memory block.
`virtAddr`	Address of memory block in kernel virtual address space.
`dspAddr`	Address of memory block in DSP address space (If the pool is in shared memory).
`size`	Size of memory block in bytes.

Comments

This structure will be used internally at IOCTL level for typecasting the open argument passed by user. Allocator open function will return mapping specific parameters in this structure, which will be used for mapping from User to Kernel and vice-versa.

Constraints

None.

See Also

`None.`

8.2 API Definition

8.2.1 LDRV_POOL_init

This function initializes the POOL component.

Syntax

```
DSP_STATUS LDRV_POOL_init () ;
```

Arguments

None.

Return Value

DSP_SOK	The POOL component has been successfully initialized.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.
DSP_ECONFIG	Incorrect configuration.

Comments

This function initializes the POOL component.

Constraints

None.

See Also

LDRV_POOL_exit ()

8.2.2 LDRV_POOL_exit

This function finalizes the POOL component.

Syntax

```
DSP_STATUS LDRV_POOL_exit () ;
```

Arguments

None.

Return Value

DSP_SOK	The POOL component has been successfully finalized.
DSP_EINVALIDARG	Invalid argument.
DSP_EMEMORY	Operation failed due to a memory error.
DSP_EFAIL	General failure.

Comments

This function internally calls the finalize function of all buffer pools through the function interface table.

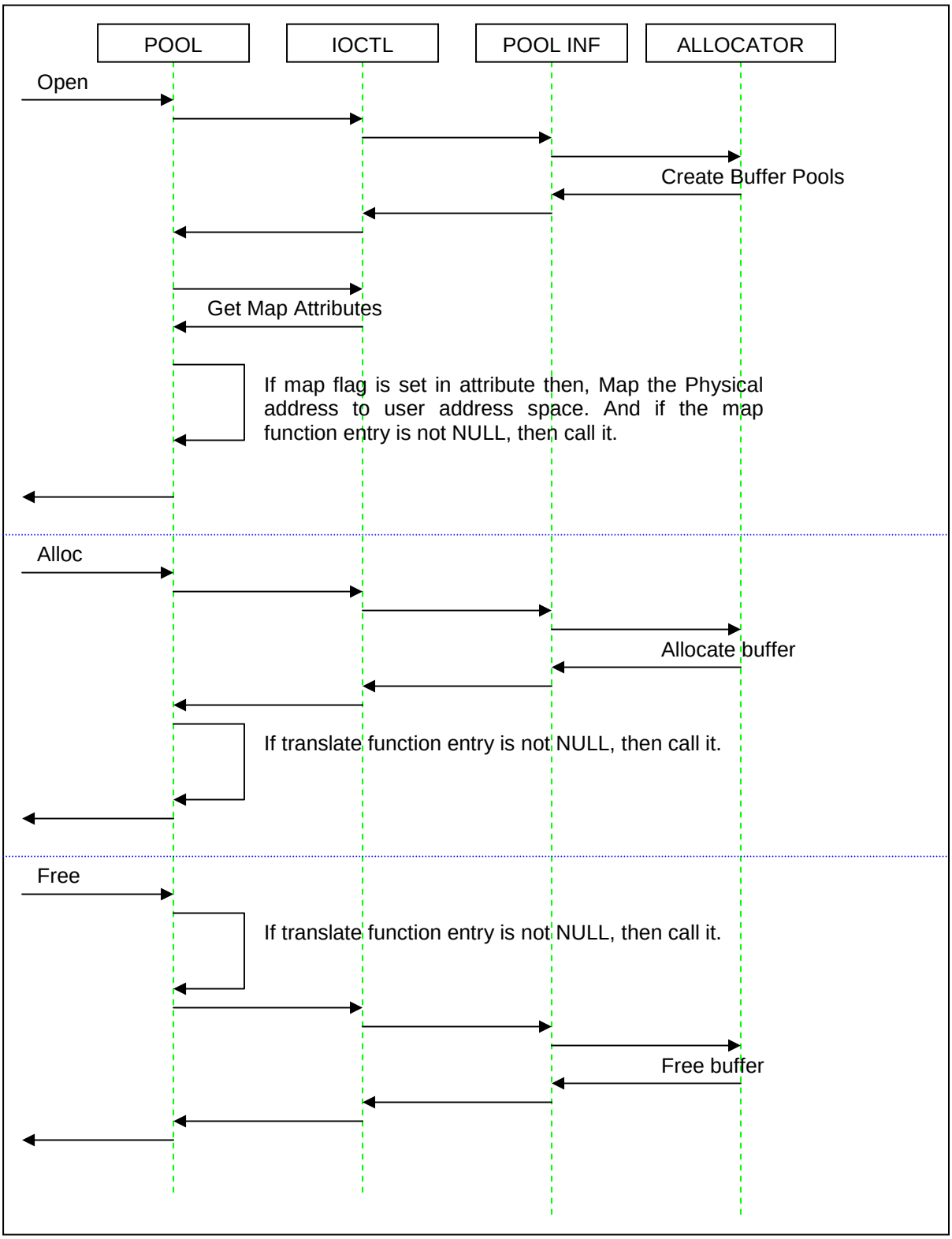
Constraints

None.

See Also

LDRV_POOL_init ()

9 Sequence Diagram



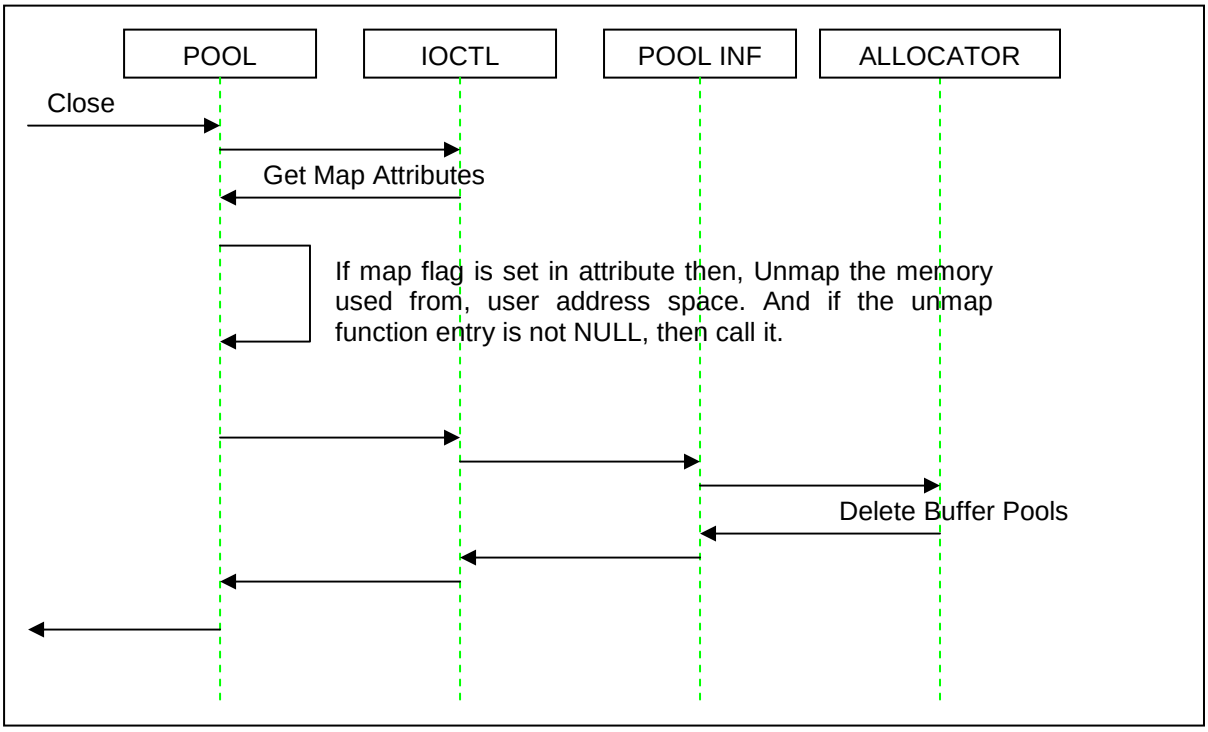


Figure 1 Dry Run for POOL Module.