DSP/BIOS™ LINK

PCI Link Driver Design

LNK 132 DES

Version 1.00

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

Purpose & Scope

This document describes the PCI Driver Redesign for DSP/BIOS™ LINK.

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

Terms & Abbreviations

DSPLINK	DSP/BIOS™ LINK
SMAPOOL	Shared Memory Allocator
DSP	Digital Signal Processor
GPP	General Purpose Processor
⚐	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.

References

1.	LNK 012 DES	DSP/BIOS™ LINK Link Driver design
2.	LNK_076_DES	DSP/BIOS™ LINK Buffer Pools design
3.	LNK_082_DES	DSP/BIOS™ LINK POOL design
4.		PCI specification document version 1.1

Overview

The current releases of PCI Link driver provide an implementation of message transfer that leverages the data-streaming code for transferring messages. The current implementation supports a high level portability since only the data streaming implementation needs to be ported to support messaging on new devices. However, due to performance concerns it is more efficient to provide an implementation that separates the message transfer path completely from the data streaming implementation. Such an implementation also results in lower code footprint in cases where only message transfer functionality is required.

2 Requirements

R1. This release shall provide a redesigned PCI link driver that separates the code path used for message transfer from the code path used for data streaming.

R2. This module shall enable applications to register a callback function with an associated parameter for events that occur on remote processors.

R3. This module shall enable applications to send specific event notification to remote processors. Application shall also be able to send an optional value with the event.

R4. This module shall enforce a priority on event notifications. This priority shall be controlled by the applications registering for such event notifications on the receiving side.

R5. Applications shall also be able to un-register their event callback functions at runtime if such event notification is no longer required.

R6. The product release shall support scaling out any of the new modules being added through this release.

R7. This release shall provide a sync buffer pool (similar to SMA) for MSGQ, CHNL and RINGIO protocols.

R8. This release shall provide critical section (lock) over PCI using IPS event.

R9. This release shall provide inter processor signaling component to realize events on top of interrupt.

3 Assumptions

None.

4 Constraints

None.

5 High Level Design

PCI device initialization

In Linux, when kernel boots up, it scans all PCI slots, and create a linked list of PCI devices. Besides creating a linked list, it also initialize the configuration space registers, create memory hole for PCI IO access, assign IRQ with help of IRQ router mapping table, etc.

Now in DSPLINK PCI device driver, the following steps are followed:

1) Find the DM642 cards, in the system. This is done by search the Linux pci device linked list with PCI vendor ID and device ID.

   `pci_find_device (PCI_TI_VENDOR, PCI_TI_DEVICE, dev)`

   Where PCI_TI_VENDOR is 104C and PCI_TI_DEVICE is 9065 for DM642.

2) After finding DM642 cards, next is to read the physical address of IO/memory region exposed by DM642 card. This is done with the help of APIs:

   a. pci_resource_start – gives the start of region.

   b. pci_resource_len – gives the length of the region.

   c. pci_resource_flags – tells what type of region such as IO register or IO memory.

   These APIs read this information from the PCI configuration space BAR settings. In case of DM642, BAR 0 if for prefetchable memory region and BAR 1 is for IO register region. Now these regions are mapped onto Linux kernel virtual memory space with the help of request_mem_region/ request_region and ioremap APIs. For IO register region one must call request_region and request_mem_region for memory regions.

3) The IRQ/interrupt number is read from either PCI configuration space or kernel structure. It is good to use Kernel structure for this. Then register a IRQ handler with request_irq API.

4) Since DM642 card has master capabilities (i.e. it can access HOST memory directly without HOST CPU intervention, with help of DMA engine located on the card), make the device as bus master.

5) Finally enable the device using pci_enable_device API.

After this, the device is ready to be used. (These steps must be carried for each of desired devices found in the system).

More on DMA

Devices like DM642, have DMA capabilities to transfer buffer between host and DSP.

For these to happen, Linux has to give mastership of PCI bus to the device. This is done using following APIs:

pci_set_dma_mask, pci_set_drvdata, pci_set_master, pci_set_mwi.

And after these are done, command register in PCI config space is updated to indicate mastership.

Once the device has attained mastership of PCI bus, it can DMA buffers using three register PCI Memory Address, DSP memory address and PCI memory count register. More details about these registers can be found in respectively device's documents.

Shared memory

Since in DSPLINK all protocols used between GPP and DSP are based on shared memory, the 4MB memory region exposed by DSP (DM642) is used to provide this shared memory. This 4MB memory region can point to any 4MB contiguous memory region from DSP address space with the help DSPP (DSP page pointer register). Only POOL is implemented in slight different manner then all other protocols. Where all control information are accessed through the 4MB shared memory region, 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.

POOL

POOL is basic backbone of ZCPY (zero copy) mechanism of transferring information in DSPLINK. But it requires shared memory, which may or may not available. So a slight modification in POOL, can fix this problem in case when shared memory is not present. Like in case of 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.

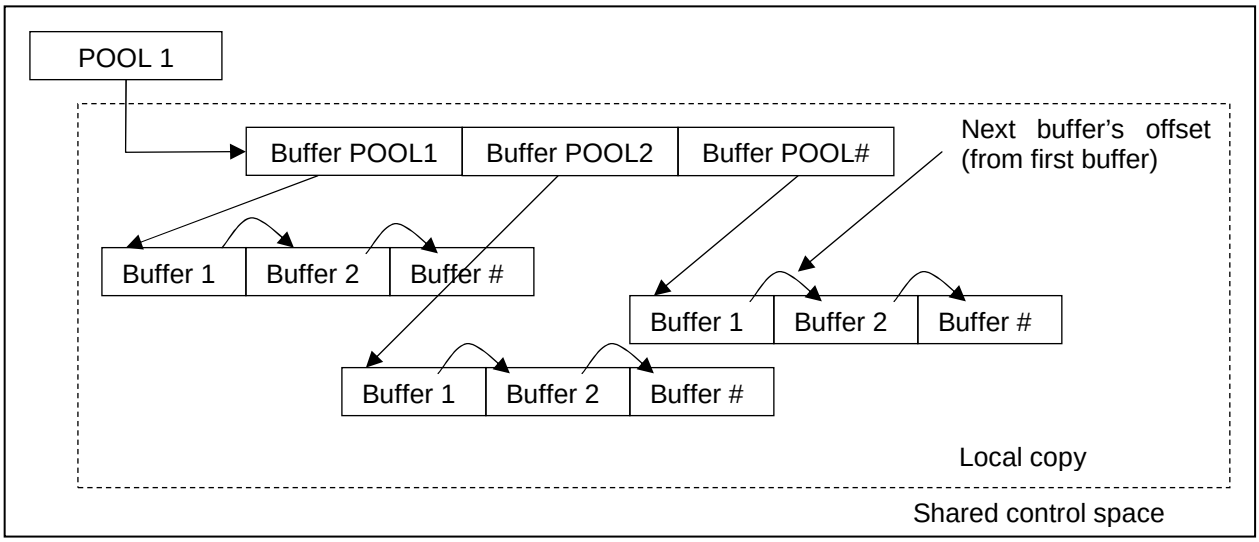
These copies are kept in sync with the help of DMA engine.

Standard Linux kernel is patched with big physical area patch. This patch enables to allocate big physical buffers, which may not be possible by using standard Linux kernel APIs. This big physical buffer becomes the local copy of buffer area on GPP side.

On the DSP side, a memory area from DSP memory map is left unused. This will subsequently become DSP side local copy of buffer area.

5.1.1 Create

GPP does the POOL initialization, including buffer list creation and control structure initialization. In this type of POOL, buffer list is created on the local copy and buffer list's next free buffer is now an offset instead of address. This helps in ease use of POOL on DSP side (no address translation required). Once all this initialization is done, GPP copy its local buffer area on to DSP side local copy using DMA engine. Please refer to below figure.



5.1.2 Alloc

Allocation is similar to normal POOL allocation technique, the only difference is that nextFree member of buffer POOL header contains next available buffer's offset. So this offset is converted into an address using local buffer area's base address and offset value contained in the just allocated buffer is written into the nextFree member. After this buffer POOL header is copied to the DSP side using DMA. This stands true for reverse case also.

5.1.3 Free

Free is similar to normal POOL Free technique, the only difference is that nextFree member of buffer POOL header contains next available buffer's offset. So buffer address (to be freed) is converted into an offset using local buffer area's base address and written to nextFree member of buffer POOL header. Also, previous value of nextFree member (an offset) is written to first word of freed buffer. After this buffer POOL header is copied to the DSP side using DMA and also the buffer is copied to corresponding DSP side copy using DMA. This stands true for reverse case also.

Two new APIs are added to GPP side POOL so that MSGQ, CHNL protocols can be used with this POOL. These APIs are

1) <internal POOL module>_writeBack

   This API writes the content of passed buffer to DSP side address (also passed to this API).

2) <internal POOL module>_invalidate

   This API reads DSP side buffer (also passed to this API) to the local buffer.

Other Modules

All other modules like MSGQ, CHNL, etc are exactly same as they are in ZCPY technique. Since they internally depend upon POOL for zero copy mechanism, thus they left unchanged.

6 Sequence Diagrams

6.1.1 SMAPOOL_init

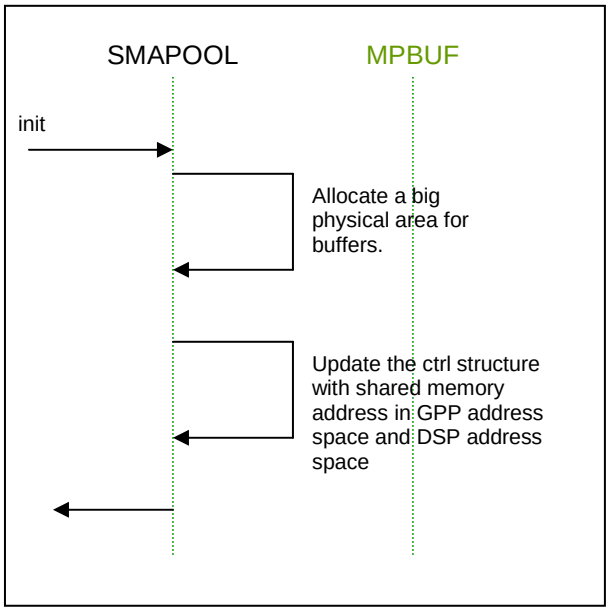


Figure 1. Initialization of SMAPOOL

6.1.2 SMAPOOL_exit

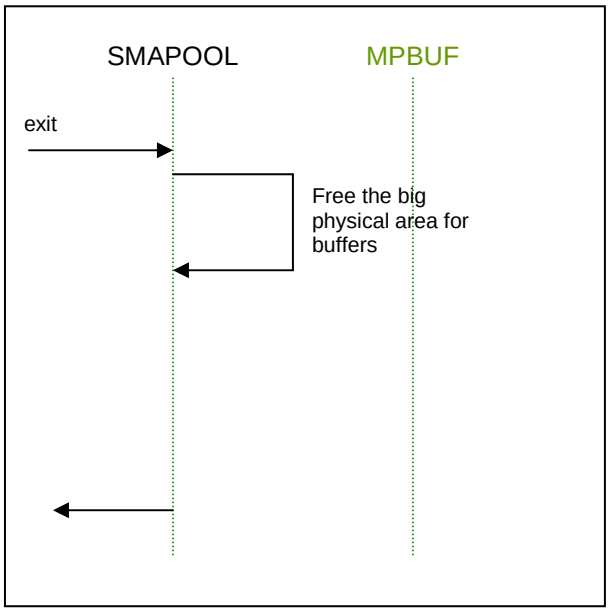


Figure 2. Finalization of SMAPOOL

6.1.3 SMAPOOL_Open

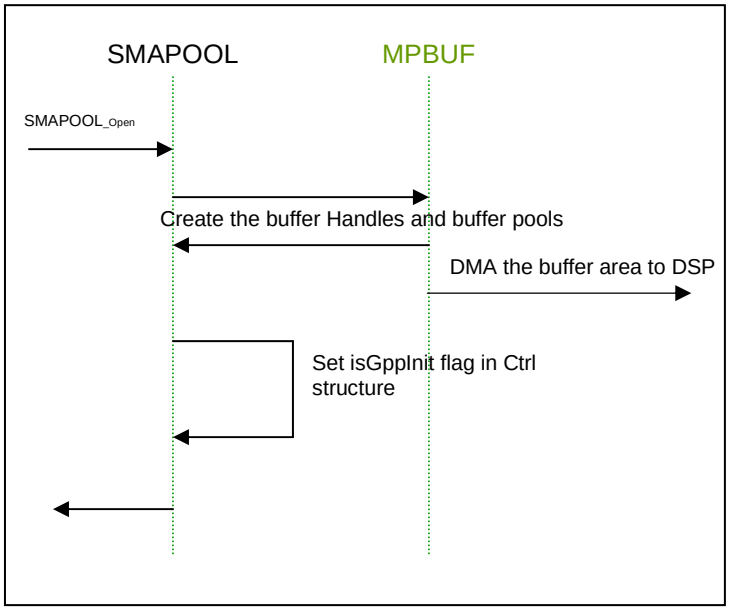


Figure 3. SMAPOOL_Open

6.1.4 SMAPOOL_Close

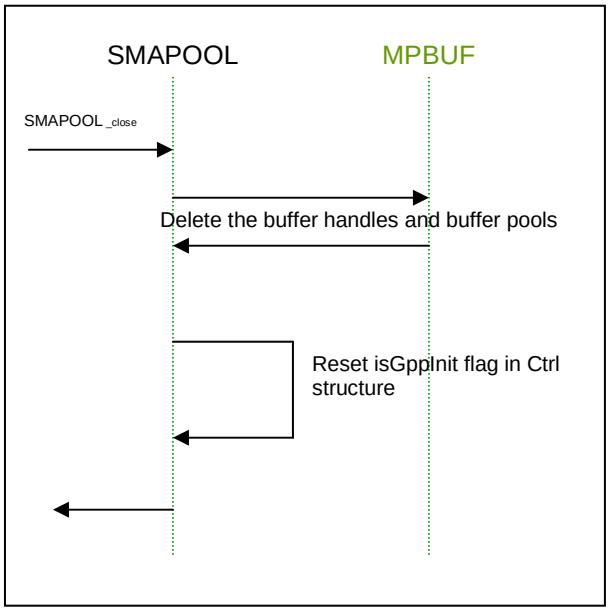


Figure 4. SMAPOOL_Close

6.1.5 SMAPOOL_Alloc

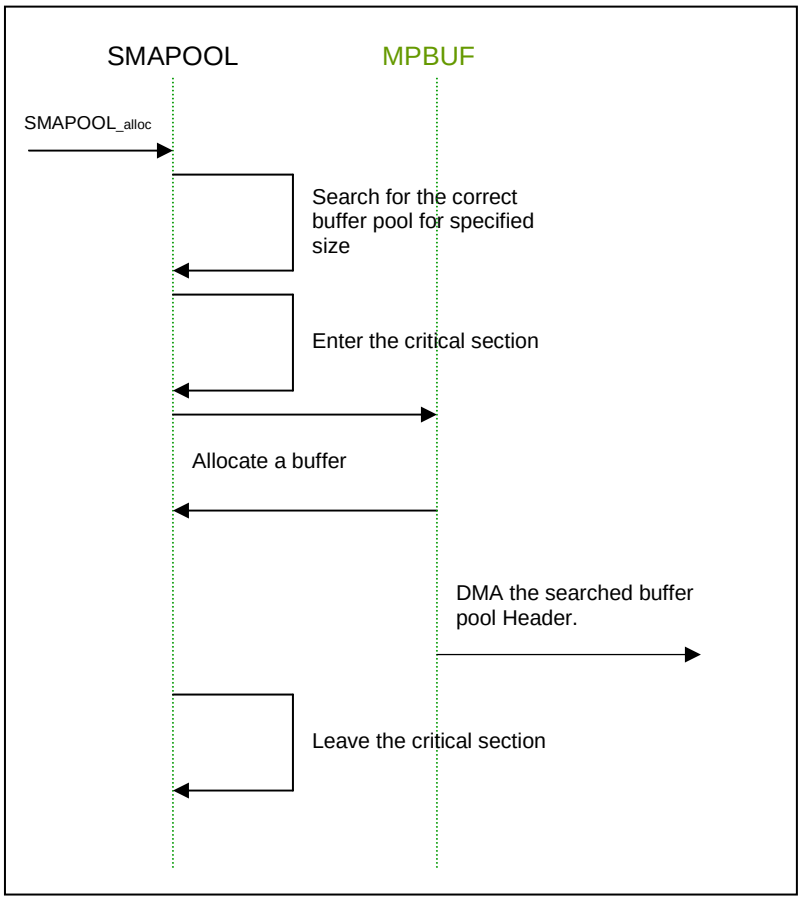


Figure 5. Allocation of a buffer using SMAPOOL

6.1.6 SMAPOOL_Free

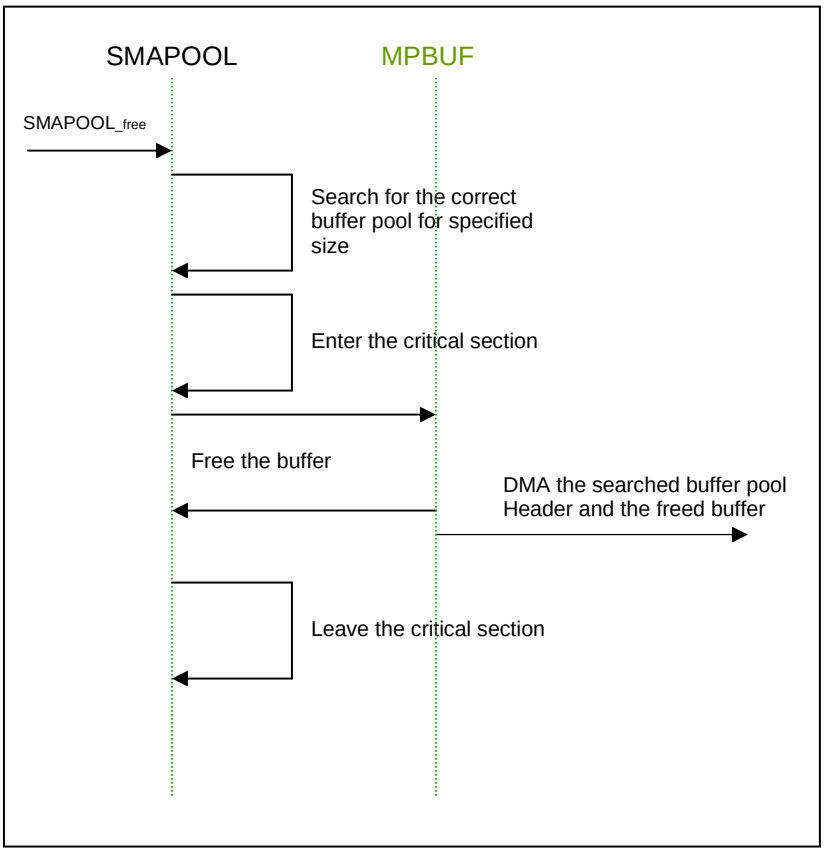


Figure 6. Freeing of a buffer using SMAPOOL

7 API Definition

7.1.1 SMAPOOL_writeback

This function writes the content of GPP buffer into DSP buffer (with offset in sync).

Syntax

```
DSP_STATUS SMAPOOL_writeback (IN  ProcessorId       dspId,
                              IN  Uint32            poolId,
                              IN  Pvoid             object,
                              IN  Pvoid             buf,
                              IN  Uint32            size) ;
```

Arguments

`IN` `ProcessorId` `dspId`

DSP Identifier.

`IN` `Uint32` `poolId`

Pool Identifier.

`IN` `Pvoid` `object`

Pointer to the pool-specific object.

`IN` `Pvoid` `buf`

Pointer to the buffer.

`IN` `Uint32` `size`

Size of the buffer to be written back.

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.

Comments

None.

Constraints

None.

See Also

`SMAPOOL_invalidate ()`

7.1.2 SMAPOOL_invalidate

This function reads the content of DSP buffer into GPP buffer (with offset in sync).

Syntax

```
DSP_STATUS SMAPOOL_invalidate (IN  ProcessorId       dspId,
                               IN  Uint32            poolId,
                               IN  Pvoid             object,
                               IN  Pvoid             buf,
                               IN  Uint32            size) ;
```

Arguments

IN	ProcessorId	dspId

DSP Identifier.

IN	Uint32	poolId

Pool Identifier.

IN	Pvoid	object

Pointer to the pool-specific object.

IN	Pvoid	buf

Pointer to the buffer.

IN	Uint32	size

Size of the buffer to be written back.

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.

Comments

None.

Constraints

None.

See Also

SMAPOOL_writeback ()