



IBM Systems & Technology Group  
Cell/Quasar Ecosystem & Solutions Enablement

# Developing Code for Cell – DMA & Mailboxes

Cell Programming Workshop  
Cell/Quasar Ecosystem Solutions Enablement

## Class Objectives – Things you will learn

- How MFC commands are used to access main storage and maintain synchronization with other processors and devices in the system
- DMA transfer and how to initiate a DMA transfer from an SPE
- Double buffering and multi-buffering DMA transfers
- Mailboxes for communications messaging

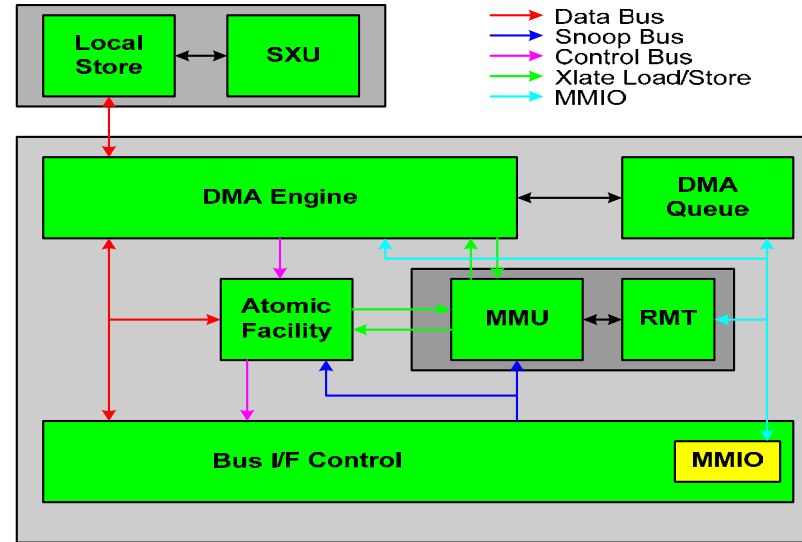
**Trademarks** - Cell Broadband Engine and Cell Broadband Engine Architecture are trademarks of Sony Computer Entertainment, Inc.

## Class Agenda

- MFC Commands
- DMA Commands
- DMA-Command Tag Groups
- DMA Transfers
- DMA To/From Another SPE
- DMA Command Status
- DMA Transfers Example
- Mailboxes
- Reading and Writing Mailboxes
- SPU Write Outbound Mailboxes
- SPU Read Inbound Mailbox
- PPE Mailbox Queue – PPE Calls, SPU Calls
- SPU Mailbox Queue – PPE Calls, SPU Calls

# Cell's Primary Communication Mechanisms

- DMA transfers, mailbox messages, and signal-notification
- All three are implemented and controlled by the SPE's MFC



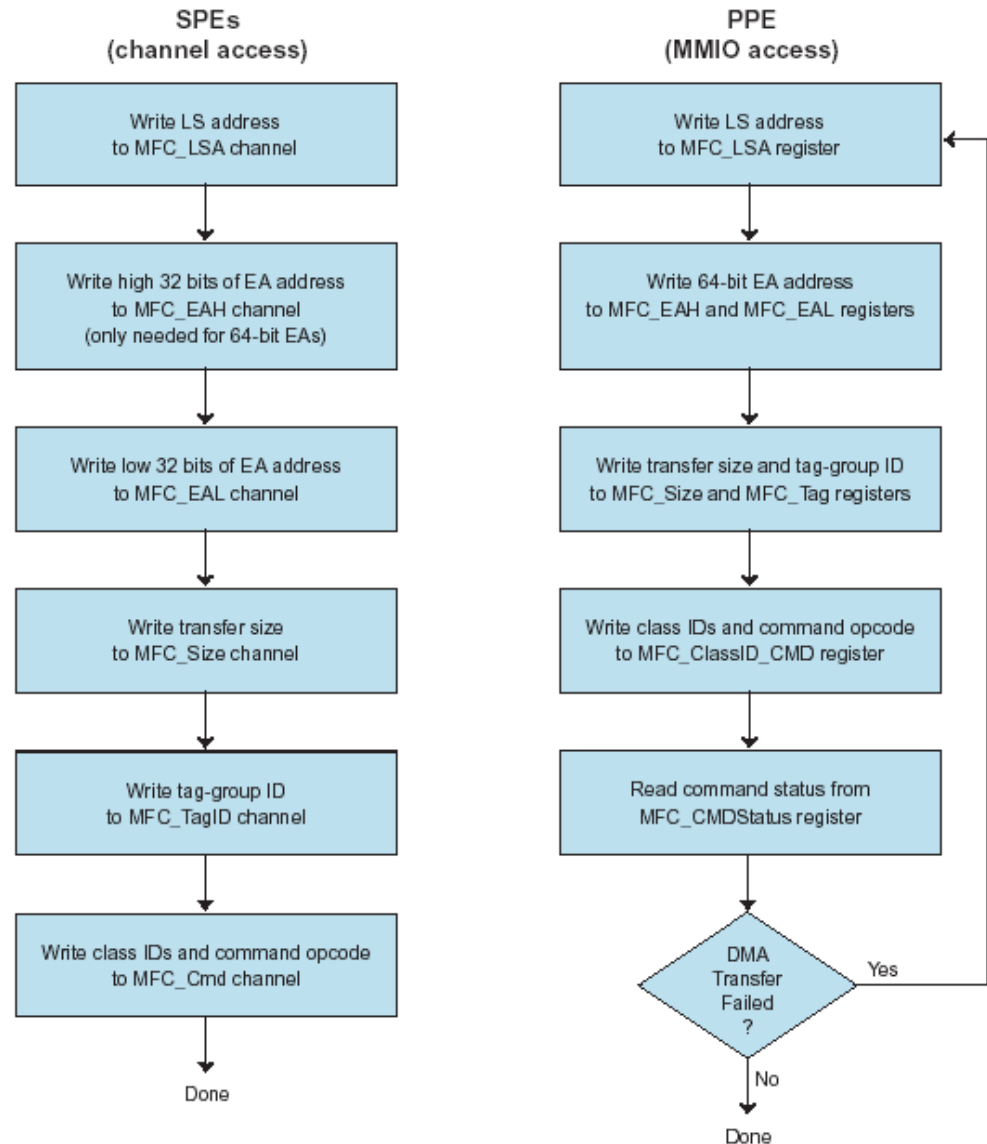
Mechanism	Description
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called <i>signaling</i> ) uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.

## MFC Commands

- Main mechanism for SPUs to
  - access main storage (DMA commands)
  - maintain **synchronization** with other processors and devices in the system (Synchronization commands)
- Can be issued either SPU via its MFC by PPE or other device, as follows:
  - Code running on the SPU issues an MFC command by executing a series of writes and/or reads using **channel instructions** - read channel (rdch), write channel (wrch), and read channel count (rchcnt).
  - Code running on the PPE or other devices issues an MFC command by performing a series of stores and/or loads to **memory-mapped I/O** (MMIO) registers in the MFC
- MFC commands are queued in one of two independent MFC command queues:
  - MFC SPU Command Queue — For channel-initiated commands by the associated SPU
  - MFC Proxy Command Queue — For MMIO-initiated commands by the PPE or other device

## Sequences for Issuing MFC Commands

- ✓ All operations on a given channel are unidirectional (they can be only read or write operations for a given channel, not bidirectional)
- ✓ Accesses to channel-interface resources through MMIO addresses do not stall
- ✓ Channel operations are done in program order
- ✓ Channel read operations to reserved channels return '0's
- ✓ Channel write operations to reserved channels have no effect
- ✓ Reading of channel counts on reserved channels returns '0'
- ✓ Channel instructions use the 32-bit preferred slot in a 128-bit transfer



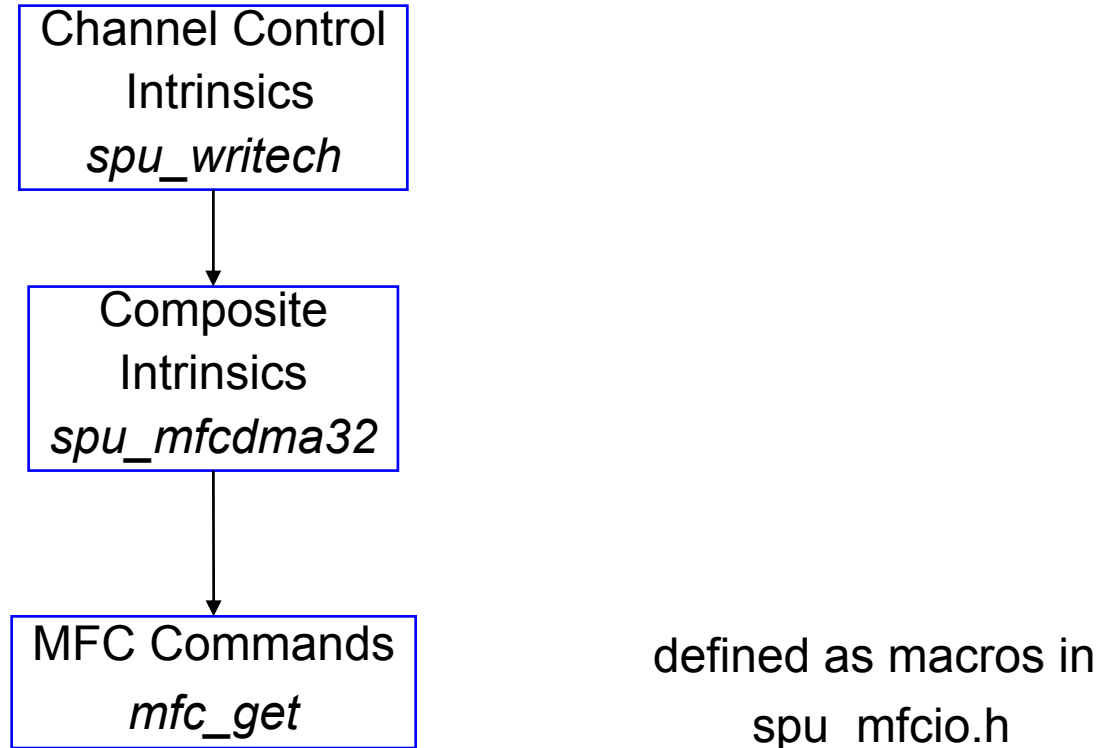
## DMA Overview

## DMA Commands

- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**



# DMA Commands



For details see: SPU C/C++ Language Extensions

## DMA Get and Put Command (SPU)

- **DMA get from main memory into local store**

```
(void) mfc_get( volatile void *ls, uint64_t ea, uint32_t size,  
              uint32_t tag, uint32_t tid, uint32_t rid)
```

- **DMA put into main memory from local store**

```
(void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size,  
              uint32_t tag, uint32_t tid, uint32_t rid)
```

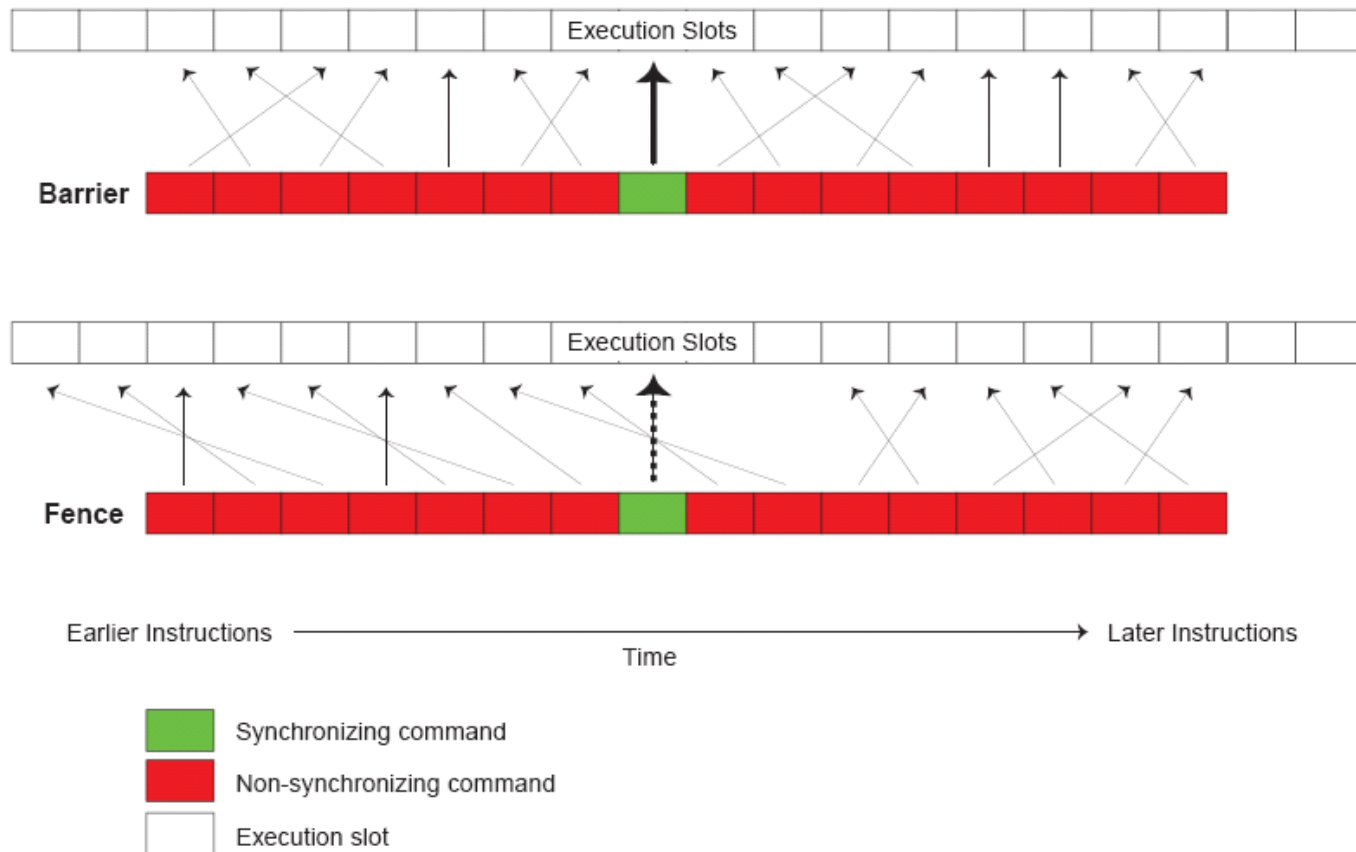
- To ensure order of DMA request execution:

- mfc\_putf : **fenced** (all commands executed before within the same tag group must finish first, later ones could be before)
- mfc\_putb : **barrier** (the barrier command and all commands issued thereafter are not executed until all previously issued commands in the same tag group have been performed)

## DMA-Command Tag Groups

- 5-bit DMA Tag for all DMA commands (except getllar, putllc, and putlluc)
- Tag can be used to
  - determine status for entire group or command
  - check or wait on the completion of all queued commands in one or more tag groups
- Tagging is optional but can be useful when using barriers to control the ordering of MFC commands within a single command queue.
- Synchronization of DMA commands within a tag group: fence and barrier
  - Execution of a fenced command option is delayed until all previously issued commands within the same tag group have been performed.
  - Execution of a barrier command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.

# Barriers and Fences



# DMA Characteristics

- **DMA transfers**
  - transfer sizes can be 1, 2, 4, 8, and  $n \cdot 16$  bytes ( $n$  integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable
- **DMA command queues per SPU**
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
  - SPU-initiated DMA is always preferable
- **DMA tags**
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands
- **DMA lists**
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests

## PPE – SPE DMA Transfer

## Transfer from PPE (Main Memory) to SPE

- **DMA get from main memory**

```
mfc_get(lsaddr,ea,size,tag_id,tid,rid);
```

- lsaddr = target address in SPU local store for fetched data (SPU local address)
- ea = effective address from which data is fetched (global address)
- size = transfer size in bytes
- tag\_id = tag-group identifier
- tid = transfer-class id
- rid = replacement-class id

- **Also available via “composite intrinsic”:**

```
spu_mfcdma64(lsaddr, eahi, ealow, size, tag_id, cmd);
```

## DMA Command Status (SPE)

- **DMA read and write commands are non-blocking**
- **Tags, tag groups, and tag masks used for:**
  - checking status of DMA commands
  - waiting for completion of DMA commands
- **Each DMA command has a 5-bit tag**
  - commands with same tag value form a “tag group”
- **Tag mask is used to identify tag groups for status checks**
  - tag mask is a 32-bit word
  - each bit in the tag mask corresponds to a specific tag id:

$\text{tag\_mask} = (1 \ll \text{tag\_id})$



## DMA Tag Status (SPE)

- **Set tag mask**

```
unsigned int tag_mask;
```

```
mfc_write_tag_mask(tag_mask);
```

- tag mask remains set until changed

- **Fetch tag status**

```
unsigned int result;
```

```
result = mfc_read_tag_status(); /* or mfc_stat_tag_status(); */
```

- tag status is logically ANDed with current tag mask
- tag status bit of '1' indicates that no DMA requests tagged with the specific tag id (corresponding to the status bit location) are still either in progress or in the DMA queue

## Waiting for DMA Completion (SPE)

- **Wait for any tagged DMA:**

`mfc_read_tag_status_any()`:

- wait until **any** of the specified tagged DMA commands is completed

- **Wait for all tagged DMA:**

`mfc_read_tag_status_all()`:

- wait until **all** of the specified tagged DMA commands are completed

- **Specified tagged DMA commands = command specified by current tag mask setting**

## DMA Example: Read into Local Store

```
inline void dma_mem_to_ls(unsigned int mem_addr,  
    volatile void *ls_addr,unsigned int size)  
{  
    unsigned int tag = 0;  
    unsigned int mask = 1;  
    mfc_get(ls_addr,mem_addr,size,tag,0,0);  
    mfc_write_tag_mask(mask);  
    mfc_read_tag_status_all();  
}
```

Read contents  
of mem\_addr  
into ls\_addr

Set tag mask

Wait for all tag  
DMA completed

## DMA Example: Write to Main Memory

```
inline void dma_ls_to_mem(unsigned int mem_addr, volatile
void *ls_addr, unsigned int size)
{
    unsigned int tag = 0;
    unsigned int mask = 1;
    mfc_put(ls_addr, mem_addr, size, tag, 0, 0);
    mfc_write_tag_mask(mask);
    mfc_read_tag_status_all();
}
```

Write contents of  
mem\_addr into  
ls\_addr

Set tag mask

Wait for all tag  
DMA completed

## SPE – SPE DMA Transfer

## SPE – SPE DMA

- Address in the other SPE's local store is represented as a 32-bit effective address (global address)
- SPE issuing the DMA command needs a pointer to the other SPE's local store as a 32-bit effective address (global address)
- PPE code can obtain effective address of an SPE's local store:  

```
#include <libspe.h>  
speid_t speid;  
void *spe_ls_addr;  
  
..  
spe_ls_addr = spe_get_ls(speid);
```
- Effective address of an SPE's local store can then be made available to other SPEs (e.g. via DMA or mailbox)

## Tips to Achieve Peak Bandwidth for DMAs

- The performance of a DMA data transfer is best when the source and destination addresses have the same quadword offsets within a PPE cache line.
- Quadword-offset-aligned data transfers generate full cache-line bus requests for every unrolling, except possibly the first and last unrolling.
- Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively.

## Mailboxes Overview



## Uses of Mailboxes

- To communicate messages up to 32 bits in length, such as buffer completion flags or program status
  - e.g., When the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then writes to an outbound mailbox to notify the PPE that its computation is complete
- Can be used for any short-data transfer purpose, such as sending of storage addresses, function parameters, command parameters, and state-machine parameters
- Can also be used for communication between an SPE and other SPEs, processors, or devices
  - Privileged software needs to allow one SPE to access the mailbox register in another SPE by mapping the target SPE's problem-state area into the EA space of the source SPE. If software does not allow this, then only atomic operations and signal notifications are available for SPE-to-SPE communication.

## Mailboxes - Characteristics

**Each MFC provides three mailbox queues of 32 bit each:**

**1. PPE (“SPU write outbound”) mailbox queue**

- SPE writes, PPE reads
- 1 deep
- SPE stalls writing to full mailbox

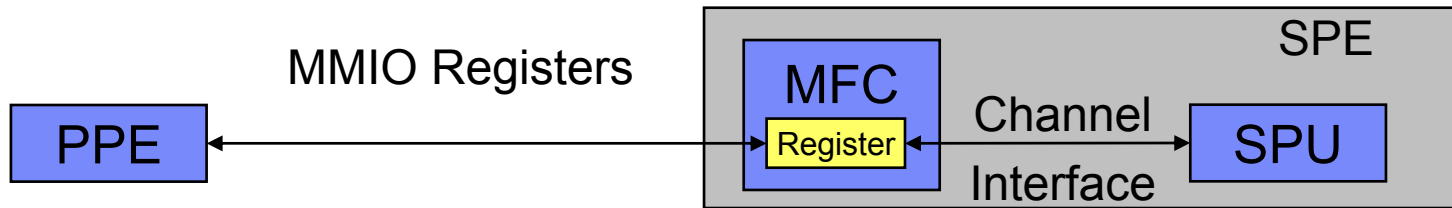
**2. PPE (“SPU write outbound”) interrupt mailbox queue**

- like PPE mailbox queue, but an interrupt is posted to the PPE when the mailbox is written

**3. SPU (“SPU read inbound”) mailbox queue**

- PPE writes, SPE reads
- 4 deep
- can be overwritten

➤ **Each mailbox entry is a fullword**



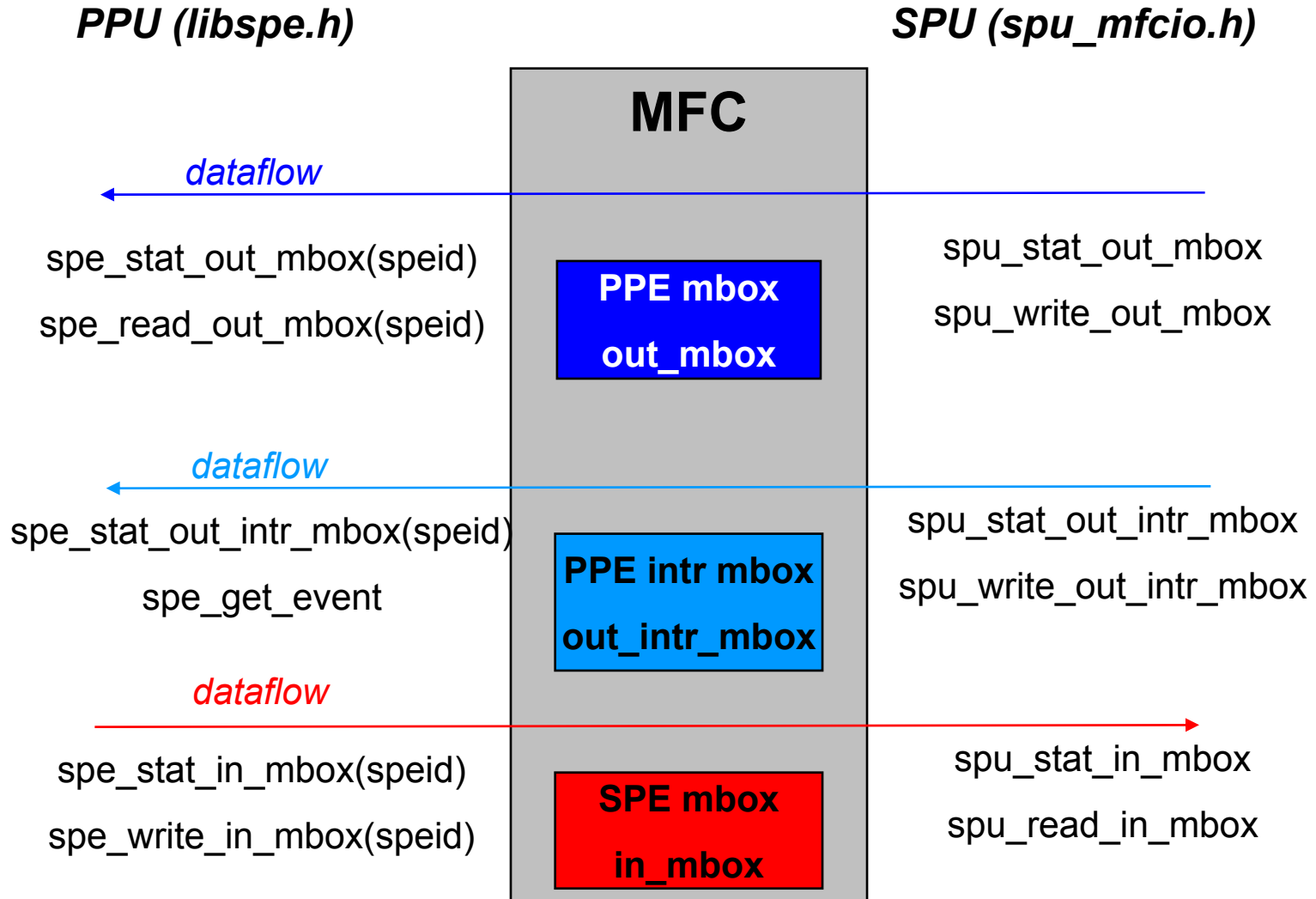
## Channels

- SPE (outgoing)
  - write the 32-bit message value to either its two outbound mailbox channels
- SPE (incoming)
  - reads a message in the inbound mailbox
- PPE and other devices (incoming)

- read message in outbound mailbox by reading the MMIO register in the SPE's MFC
- PPE and other devices (outgoing)
  - send by writing the associated MMIO register
- For interrupts associated with the SPU Write Outbound Interrupt Mailbox,
  - **no ordering of the interrupt and previously issued MFC commands**

## MMIO Registers

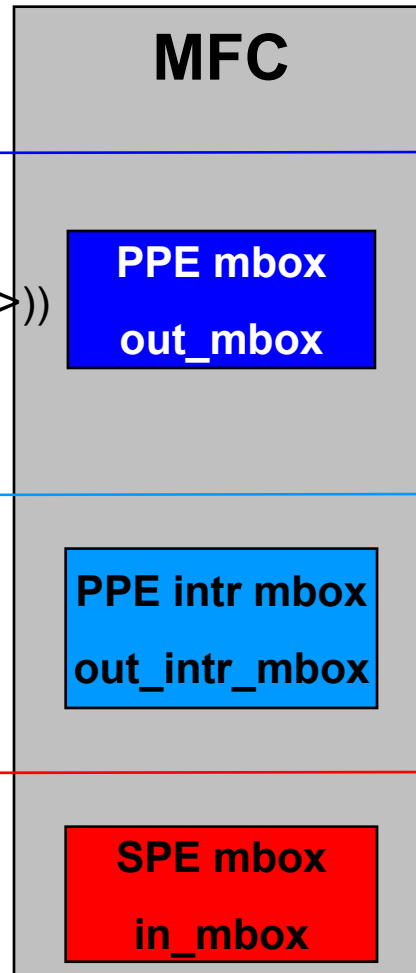
# Mailboxes API – libspe1.x



## Mailboxes API – libspe2 (See more from libspe2 documents)

**PPU (libspe2.h)**

**SPU (spu\_mfcio.h)**



*dataflow*



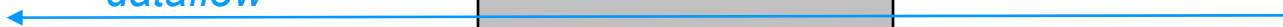
spe\_out\_mbox\_status(<speid>)

spe\_out\_mbox\_read(<speid>, &<data>))

**PPE mbox  
out\_mbox**

spu\_stat\_out\_mbox  
spu\_write\_out\_mbox

*dataflow*



spe\_out\_intr\_mbox\_status(<speid>)

spe\_get\_event

**PPE intr mbox  
out\_intr\_mbox**

spu\_stat\_out\_intr\_mbox  
spu\_write\_out\_intr\_mbox

*dataflow*



spe\_in\_mbox\_status(<speid>)

spe\_in\_mbox\_write(<speid>, <data>)

**SPE mbox  
in\_mbox**

spu\_stat\_in\_mbox  
spu\_read\_in\_mbox

## SPU Write Outbound Mailboxes

## SPU Write Outbound Mailbox

- The value **written** to the SPU Write Outbound Mailbox channel SPU\_WrOutMbox is entered into the outbound mailbox in the MFC if the mailbox has capacity to accept the value.
- If the mailbox can **accept** the value, the channel count for SPU\_WrOutMbox is **decremented** by '1'.
- If the outbound mailbox is **full**, the channel count will read as '0'.
- If SPE software writes a value to SPU\_WrOutMbox when the channel count is '0', the SPU will **stall** on the write.
- The SPU **remains stalled** until the PPE or other device reads a message from the outbound mailbox by reading the MMIO address of the mailbox.
- When the mailbox is **read** through the MMIO address, the channel count is incremented by '1'.

## SPU Write Outbound Interrupt Mailbox

- The value **written** to the SPU Write Outbound Interrupt Mailbox channel (SPU\_WrOutIntrMbox) is entered into the outbound interrupt mailbox if the mailbox has capacity to accept the value.
- If the mailbox can **accept** the message, the channel count for SPU\_WrOutIntrMbox is decremented by '1', and an **interrupt is raised** in the PPE or other device, depending on interrupt enabling and routing.
- There is no ordering of the interrupt and previously issued MFC commands.
- If the outbound interrupt mailbox is **full**, the channel count will read as '0'.
- If SPE software writes a value to SPU\_WrOutIntrMbox when the channel count is '0', the SPU will **stall** on the write.
- The SPU **remains stalled** until the PPE or other device reads a mailbox message from the outbound interrupt mailbox by reading the MMIO address of the mailbox.
- When this is done, the channel count is **incremented** by '1'.



## Waiting to Write SPU Write Outbound Mailbox Data

- To **avoid SPU stall**, SPU can use the read-channel-count instruction on the SPU Write Outbound Mailbox channel to determine if the queue is empty before writing to the channel.
- If the read-channel-count instruction returns '0', the SPU Write Outbound Mailbox Queue is full.
- If the read channel-count instruction returns a non-zero value, the value indicates the number of free entries in the SPU Write Outbound Mailbox Queue.
- When the queue has free entries, the SPU can write to this channel without stalling the SPU.

### Polling SPU Write Outbound Mailbox or SPU Write Outbound Interrupt Mailbox.

```
/* To write the value 1 to the SPU Write Outbound Interrupt Mailbox instead
 * of the SPU Write Outbound Mailbox, simply replace SPU_WrOutMbox
 * with SPU_WrOutIntrMbox in the following example.*/
unsigned int mb_value;
do {
    /* Do other useful work while waiting.*/
} while (!spu_readchcnt(SPU_WrOutMbox)); // 0 → full, so something useful
spu_writtech(SPU_WrOutMbox, mb_value);
```

## Polling for or Block on an SPU Write Outbound Mailbox Available Event

```
#define MBOX_AVAILABLE_EVENT 0x00000080
unsigned int event_status;
unsigned int mb_value;
spu_writch(SPU_WrEventMask, MBOX_AVAILABLE_EVENT);
do {
    /*
     * Do other useful work while waiting.
     */
} while (!spu_readchcnt(SPU_RdEventStat));
event_status = spu_readch(SPU_RdEventStat); /* read status */
spu_writch(SPU_WrEventAck, MBOX_AVAILABLE_EVENT); /* acknowledge event */
spu_writch(SPU_WrOutMbox, mb_value); /* send mailbox message */
```

- **NOTES:** To block, instead of poll, simply delete the do-loop above.

## PPU reads SPU Outbound Mailboxes

- PPU must check Mailbox Status Register first
  - **check that unread data is available in the SPU Outbound Mailbox or SPU Outbound Interrupt Mailbox**
  - **otherwise, stale or undefined data may be returned**
- To determine that unread data is available
  - **PPE reads the Mailbox Status register**
  - **extracts the count value from the SPU\_Out\_Mbox\_Count field**
- count is
  - **non-zero → at least one unread value is present**
  - **zero → PPE should not read but poll the Mailbox Status register**

## SPU Read Inbound Mailbox

## SPU Read Inbound Mailbox Channel

- Mailbox is **FIFO** queue
  - If the SPU Read Inbound Mailbox channel (SPU\_RdInMbox) has a message, the value read from the mailbox is the oldest message written to the mailbox.
- Mailbox Status (empty: channel count =0)
  - If the inbound mailbox is empty, the SPU\_RdInMbox channel count will read as '0'.
- **SPU stalls on reading empty mailbox**
  - If SPE software reads from SPU\_RdInMbox when the channel count is '0', the SPU will stall on the read. The SPU remains stalled until the PPE or other device writes a message to the mailbox by writing to the MMIO address of the mailbox.
- When the mailbox is **written** through the MMIO address, the channel count is **incremented** by '1'.
- When the mailbox is **read** by the SPU, the channel count is **decremented** by '1'.

## SPU Read Inbound Mailbox Characteristics

- The SPU Read Inbound Mailbox can be **overrun** by a PPE in which case, mailbox message data will be lost.
- A **PPE writing** to the SPU Read Inbound Mailbox will **not stall** when this mailbox is full.

## PPE Access to Mailboxes

- PPE can derive “addresses” of mailboxes from spe thread id
- First, create SPU thread, e.g.:

```
speid_t spe_id;  
spe_id = spe_create_thread(0,spu_load_image,NULL,NULL,-1,0);
```

  - spe\_id has type speid\_t (normally an int)
- PPE mailbox calls use **spe\_id** to identify desired SPE’s mailbox
- Functions are in **libspe.a**

## Read: PPE Mailbox Queue – PPE Calls (libspe.h)

- **“SPU outbound” mailbox**
- **Check mailbox status:**
  - `unsigned int count;`
  - `count = spe_stat_out_mbox(spe_id);`
  - `count = 0` → no data in the mailbox
  - otherwise, `count` = number of incoming 32-bit words in the mailbox
- **Get mailbox data:**
  - `unsigned int data;`
  - `data = spe_read_out_inbox(spe_id);`
  - `data` contains next 32-bit word from mailbox
  - routine is non-blocking
  - routine returns `MFC_ERROR (0xFFFFFFFF)` if no data in mailbox



## Write: PPE Mailbox Queues – SPU Calls (spu\_mfcio.h)

- **“SPU outbound” mailbox**
- **Check mailbox status:**
  - `unsigned int count;`
  - `count = spu_stat_out_mbox();`
  - `count = 0` → mailbox is full
  - otherwise, `count` = number of available 32-bit entries in the mailbox
- **Put mailbox data:**
  - `unsigned int data;`
  - `spu_write_out_mbox(data);`
  - data written to mailbox
  - routine blocks if mailbox contains unread data

## PPE Interrupting Mailbox Queue – PPE Calls

- **“SPU outbound” interrupting mailbox**
- **Check mailbox status:**
  - unsigned int count;
  - count = spe\_stat\_out\_intr\_mbox(spe\_id);
  - count = 0 → no data in the mailbox
  - otherwise, count = number of incoming 32-bit words in the mailbox
- **Get mailbox data:**
  - interrupting mailbox is a privileged register
  - user PPE applications read mailbox data via `spe_get_event`

## PPE Interrupting Mailbox Queues – SPU Calls

- **“SPU outbound” interrupting mailbox**
- **Put mailbox data:**
  - unsigned int data;
  - spe\_write\_out\_intr\_mbox(data);
    - data written to interrupting mailbox
    - routine blocks if mailbox contains unread data
- **defined in spu\_mfcio.h**

## Write: SPU Mailbox Queue – PPE Calls (libspe.h)

- **“SPU inbound” mailbox**
- **Check mailbox status:**
  - unsigned int count;
  - count = spe\_stat\_in\_mbox(spe\_id);
  - count = 0 → mailbox is full
  - otherwise, count = number of available 32-bit entries in the mailbox
- **Put mailbox data:**
  - unsigned int data, result;
  - result = spe\_write\_in\_mbox(spe\_id,data);
  - data written to next 32-bit word in mailbox
  - mailbox can overflow
  - routine returns 0xFFFFFFFF on failure

## Read: SPU Mailbox Queue – SPU Calls (spu\_mfcio.h)

- **“SPU inbound” mailbox**
- **Check mailbox status:**
  - unsigned int count;
  - count = spu\_stat\_in\_mbox();
  - count = 0 → no data in the mailbox
  - otherwise, count = number of incoming 32-bit words in the mailbox
- **Get mailbox data:**
  - unsigned int data;
  - data = spu\_read\_in\_mbox();
  - data contains next 32-bit word from mailbox
  - routine blocks if no data in mailbox

## Mailbox Channels and their Associated MMIO Registers

SPE Channel #	Name	Channel Interface					MMIO Register Interface				
		Mnemonic	Max. Entries	Blocking	R/W	Width (bits)	Offset From Base	Mnemonic	Max. Entries	R/W	Width (bits)
28	SPU Write Outbound Mailbox	SPU_WrOutMbox	1	yes	W	32	X'04004'	SPU_Out_Mbox	1	R	32
29	SPU Read Inbound Mailbox	SPU_RdInMbox	4	yes	R	32	X'0400C'	SPU_In_Mbox	4	W	32
30	SPU Write Outbound Interrupt Mailbox <sup>1</sup>	SPU_WrOutIntrMbox	1	yes	W	32	X'04000'	SPU_Out_Intr_Mbox	1	R	64
—	SPU Mailbox Status	—	—	—	—	—	X'04014'	SPU_Mbox_Stat	1	R	32

1. Access to this MMIO register is available only to privileged PPE software.

### ***Functions of Mailbox Channels (SPU)***

Channel Interface	SPU Read or Write	Functions
SPU_WrOutMbox	W	Writes message data to the outbound mailbox.
SPU_RdInMbox	R	Returns the next message data from the inbound mailbox
SPU_WrOutIntrMbox	W	Writes message data to the outbound interrupt mailbox.

## Functions of Mailbox MMIO Registers (PPU)

MMIO Register	PPE Read or Write	Functions
SPU_Out_Mbox	R	Returns the message data from the corresponding SPU outbound mailbox.
SPU_In_Mbox	W	Writes message data to the SPU inbound mailbox.
SPU_Out_Intr_Mbox <sup>1</sup>	R	Returns the message data from the corresponding SPU outbound interrupt mailbox.
SPU_Mbox_Stat	R	Returns the number of available mailbox entries.

1. Access to the SPU\_Out\_Intr\_Mbox MMIO register is available only to privileged PPE software.

## BACKUP - Reference APIs



# MFC Command Suffixes

	Suffix	Description
<b>Start SPU</b>	s	Starts the execution of the SPU at the current location indicated by the SPU Next Program Counter Register after the data has been transferred into or out of the local store.
<b>Fenced</b>	f	Tag-specific fence. Commands with a tag-specific fence are locally ordered with respect to all previously-issued commands within the same tag group and command queue.
<b>Barrier</b>	b	Tag-specific barrier. Commands with a tag-specific barrier are locally ordered with respect to all previously-issued commands within the same tag group and command queue and all subsequently-issued commands to the same command queue with the same tag.
<b>List</b>	l	List command. Executes a list of DMA transfer elements located in local store. The maximum number of elements is 2,048, and each element describes a transfer of up to 16 KB.

## MFC DMA Commands

Mnemonic	Supported By <sup>1</sup>	Description
<b>Put Commands</b>		
<b>put</b>	PPE, SPE	Moves data from local store to the effective address.
<b>puts</b>	PPE	Moves data from local store to the effective address and starts the SPU after the DMA operation completes.
<b>putf</b>	PPE, SPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>putb</b>	PPE, SPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>putfs</b>	PPE	Moves data from local store to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
<b>putbs</b>	PPE	Moves data from local store to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
<b>putl</b>	SPE	Moves data from local store to the effective address using an MFC list.
<b>putlf</b>	SPE	Moves data from local store to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>putlb</b>	SPE	Moves data from local store to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

## MFC DMA Commands (Cont'd)

Get Commands		
<b>get</b>	PPE, SPE	Moves data from the effective address to local store.
<b>gets</b>	PPE	Moves data from the effective address to local store, and starts the SPU after the DMA operation completes.
<b>getf</b>	PPE, SPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>getb</b>	PPE, SPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>getfs</b>	PPE	Moves data from the effective address to local store with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after the DMA operation completes.
<b>getbs</b>	PPE	Moves data from the effective address to local store with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after the DMA operation completes.
<b>getl</b>	SPE	Moves data from the effective address to local store using an MFC list.
<b>getlf</b>	SPE	Moves data from the effective address to local store using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
<b>getlb</b>	SPE	Moves data from the effective address to local store using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

## Synchronization Commands

# MFC Synchronization Commands

## MFC synchronization commands

- Used to control the order in which DMA storage accesses are performed
  - Four atomic commands (**getllar**, **putllc**, **putlluc**, and **putqlluc**),
  - Three send-signal commands (**sndsig**, **sndsigf**, and **sndsigb**), and
  - Three barrier commands (**barrier**, **mfcsync**, and **mfceieio**).

Command	Supported By <sup>1</sup>	Description
<b>barrier</b>	PPE, SPE	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the barrier command within the same command queue. The barrier command has no effect on the immediate DMA commands: <b>getllar</b> , <b>putllc</b> , and <b>putlluc</b> .
<b>mfceieio</b>	PPE, SPE	Controls the ordering of get commands with respect to put commands, and of get commands with respect to get commands accessing storage that is caching inhibited and guarded. Also controls the ordering of <b>put</b> commands with respect to <b>put</b> commands accessing storage that is memory coherence required and not caching inhibited.
<b>mfcsync</b>	PPE, SPE	Controls the ordering of DMA <b>put</b> and <b>get</b> operations within the specified tag group with respect to other processing units and mechanisms in the system.
<b>sndsig</b>	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE.
<b>sndsigb</b>	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with barrier.
<b>sndsigf</b>	PPE, SPE	Update SPU Signal Notification Registers in an I/O device or another SPE with fence.

1. There is a channel (for SPEs) and/or MMIO register (for PPE) to support the operation.

# MFC Atomic Commands

Command	Supported By <sup>1</sup>	Description
<b>getllar</b>	SPE	Get lock line and create a reservation (executed immediately).
<b>putllc</b>	SPE	Put lock line conditional on a reservation (executed immediately).
<b>putlluc</b>	SPE	Put lock line unconditional (executed immediately).
<b>putqlluc</b>	SPE	Put lock line unconditional (queued form).

1. There is a channel to support the operation.

## Special Notices -- Trademarks

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquires, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Many of the features described in this document are operating system dependent and may not be available on Linux. For more information, please check: [http://www.ibm.com/systems/p/software/whitepapers/linux\\_overview.html](http://www.ibm.com/systems/p/software/whitepapers/linux_overview.html)

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

Revised January 19, 2006

## Special Notices (Cont.) -- Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries: alphaWorks, BladeCenter, Blue Gene, ClusterProven, developerWorks, e business(logo), e(logo)business, e(logo)server, IBM, IBM(logo), ibm.com, IBM Business Partner (logo), IntelliStation, MediaStreamer, Micro Channel, NUMA-Q, PartnerWorld, PowerPC, PowerPC(logo), pSeries, TotalStorage, xSeries; Advanced Micro-Partitioning, eServer, Micro-Partitioning, NUMACenter, On Demand Business logo, OpenPower, POWER, Power Architecture, Power Everywhere, Power Family, Power PC, PowerPC Architecture, POWER5, POWER5+, POWER6, POWER6+, Redbooks, System p, System p5, System Storage, VideoCharger, Virtualization Engine.

A full list of U.S. trademarks owned by IBM may be found at: <http://www.ibm.com/legal/copytrade.shtml>.

Cell Broadband Engine and Cell Broadband Engine Architecture are trademarks of Sony Computer Entertainment, Inc. in the United States, other countries, or both.

Rambus is a registered trademark of Rambus, Inc.

XDR and FlexIO are trademarks of Rambus, Inc.

UNIX is a registered trademark in the United States, other countries or both.

Linux is a trademark of Linus Torvalds in the United States, other countries or both.

Fedora is a trademark of Redhat, Inc.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.

Intel, Intel Xeon, Itanium and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.

AMD Opteron is a trademark of Advanced Micro Devices, Inc.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).

SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECcapc, SPECchpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corp (SPEC).

AltiVec is a trademark of Freescale Semiconductor, Inc.

PCI-X and PCI Express are registered trademarks of PCI SIG.

InfiniBand™ is a trademark the InfiniBand® Trade Association

Other company, product and service names may be trademarks or service marks of others.

Revised July 23, 2006



## Special Notices - Copyrights

(c) Copyright International Business Machines Corporation 2005.  
All Rights Reserved. Printed in the United States September 2005.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

IBM	IBM Logo	Power Architecture
-----	----------	--------------------

Other company, product and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Microelectronics Division  
1580 Route 52, Bldg. 504  
Hopewell Junction, NY 12533-6351

The IBM home page is <http://www.ibm.com>  
The IBM Microelectronics Division home page is  
<http://www.chips.ibm.com>