

## Debugging common DMA errors

Learn how to shoot down common direct memory access errors on Cell/B.E. systems

Skill Level: Intermediate

[VanDung To \(vto@us.ibm.com\)](mailto:vto@us.ibm.com)

Software Engineer

IBM

04 Nov 2008

To access main storage, Cell Broadband Engine(TM) SPEs use direct memory access commands (DMA), which transfer data between the main storage and their private local memory. Although this organization of distributed storage promotes high performance, it requires the SPE programmer to explicitly handle the DMA transfers between main and local storage. Errors during these transfers can be difficult to detect and debug. This article provides techniques for handling common problems with SPE-initiated DMA transfers.

The Cell/B.E.<sup>TM</sup> system is a multiprocessor system on a chip that is *not* a traditional shared-memory multiprocessor. The system consists of a PowerPC® Processing Element (PPE), which accesses main storage, and eight Synergistic Processing Elements (SPEs), which access their own private local storage.

To access main storage, the SPEs use direct memory access (DMA) commands, which transfer data between main storage and their private local memory. This distributed storage organization enables high performance, but it requires the SPE programmer to explicitly handle DMA transfers between main storage and local storage. Errors during these DMA transfers can be difficult to detect and debug. This article provides the techniques for handling common problems with SPE-initiated DMA transfers.

## Understanding DMA errors

All DMA transactions on Cell/B.E. systems must follow certain guidelines. Due to the dynamic nature of DMA processing, a command parameter that does not adhere to the guidelines might not cause an error during compilation. Instead, during runtime, the Memory Flow Controller (MFC) command queue processing is suspended, and an interrupt is raised to the PPE. The application is usually terminated with either a *bus error* (SIGBUS) or a *segmentation fault* (SEGSEGV). A partial DMA transfer might occur before the MFC encounters an invalid parameter in a DMA command, raising an interrupt to the PPE.

## Bus errors

Table 1 shows common DMA command errors, which hardware can detect and which cause bus errors.

**Table 1. DMA command errors that cause bus errors (and are detected by hardware)**

Error type	Error	Description
<b>DMA size errors</b>	Bad transfer size	Transfer size is not 0, 1, 2, 4, or 8 bytes or a multiple of 16 bytes
<b>DMA size errors</b>	Transfer size is too big	Transfer size that is greater than 16KB
<b>DMA size errors</b>	List transfer size is too big	List element with size that is greater than 16KB
<b>DMA alignment errors</b>	Local storage address alignment	Target and source addresses are not naturally aligned for sizes less than 16 bytes or are not aligned on 16-byte boundary for sizes >= 16 bytes
<b>DMA alignment errors</b>	Main storage address alignment	Target and source addresses are not naturally aligned for sizes less than 16 bytes or are not aligned on 16-byte boundary for sizes >= 16 bytes
<b>DMA alignment errors</b>	List address alignment	DMA list is not stored in SPE local store on an 8-byte boundary
<b>Tag ID errors</b>	Invalid tag ID	Tag ID is not between 0 and 31 inclusive
<b>Other</b>	List element crosses	For 64-bit

	a 4GB boundary	applications, list elements cannot cross a 4GB boundary; restriction does not apply to a 32-bit application
--	----------------	---

## Segmentation violations

Table 2 shows common DMA command errors that are not detected by the hardware, causing a segmentation violation or fault. Segmentation faults that occur on the SPE are always caused by DMA transfers to and/or from bad effective addresses.

**Table 2. DMA command errors that cause segmentation violations (and are not detected by hardware)**

Error type	Description
<b>Invalid target effective address</b>	A DMA PUT command with a target effective address that cannot be accessed; for example, an address of storage that was not allocated
<b>Invalid source effective address</b>	A DMA GET command with a source effective address that cannot be accessed; for example, an address of storage that was not allocated

## Using ppu-gdb to debug DMA errors

This section extends your knowledge on how to debug Cell/B.E. applications using the ppu-gdb and spu-gdb debuggers as originally found in Chapter 3 of the *SDK Programmer's Guide* (see [Resources](#)).

### GDB command `info spu dma`

The `info spu dma` command is one of the extended commands that GDB offers to help debug Cell/B.E. applications. It displays MFC DMA status.

For each pending DMA command, the opcode, tag, and class IDs are shown, followed by the current effective address, local store address, and transfer size (updated as the command is processed). For commands using a DMA list, the local store address and size of the list are shown. The *E* column indicates commands that the MFC flags as erroneous. The output is similar to Listing 1.

### Listing 1. Commands flagged as erroneous by the MFC

```
(gdb) info spu dma
Tag-Group Status 0x00000002
Tag-Group Mask 0x00000001 ('any' query pending)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag  TId  RId  EA                LSA      Size  LstAddr
LstSize E
put      0    0    0    0x00000000f77d0080 0x01000 0x04000
*
getb     0    0    0    0x00000000f7be8080 0x01000 0x04000
put      0    0    0    0x00000000f77d4080 0x05000 0x04000
*
getb     1    0    0    0x00000000f7bec080 0x05000 0x04000
```

Table 3 provides detailed descriptions for each of the fields displayed using the info spu dma command.

**Table 3. Details of info spu dma fields**

Field	Description
<b>Tag-Group Status</b>	Displays the current tag-group status.
<b>Tag-Group Mask</b>	Displays the tag-group query mask (MFC_WrTagMask channel). The statement in parentheses indicates whether a tag status query is currently pending or not. This should be either <i>no query pending</i> , <i>"any" query pending</i> , or <i>"all" query pending</i> .
<b>Stall-and-Notify</b>	Displays the content of the MFC Read List Stall-and-Notify Tag Status (MFC_RdListStallStat) channel. List elements for a list command contain a stall-and-notify flag. If the flag is set on a list element, the MFC stops executing the list command after completing the transfer this element requested. The flag sets the bit corresponding to the tag group of the list command in this channel.
<b>Atomic Cmd Status</b>	Displays the content of the MFC Read Atomic Command Status channel (MFC_RdAtomicStat). This channel contains the status of the last-completed, immediate atomic update DMA command.
<b>Opcode</b>	Displays the opcode for the DMA transfer. If the opcode is not valid,

	the actual input value is displayed, and the <i>E</i> bit is marked.
<b>Tag</b>	Displays the tag group ID for the DMA transfer. This field is a 5-bit field, which means it does not show values outside the 0-31 range. If the tag ID is not valid, the <i>E</i> bit is marked.
<b>TId</b>	Displays the Transfer Class ID. Generally set to 0.
<b>Rid</b>	Displays the Replacement Class ID. Generally set to 0.
<b>EA</b>	Displays the effective address of the DMA transfer. If an effective address is misaligned, this field shows the actual misaligned address, and the <i>E</i> bit is marked.
<b>LSA</b>	Displays the local store address of the DMA transfer. If a local store address is misaligned, this field does not show the actual misaligned address, because the hardware does not retain the four least significant bits of the local store address. However, the <i>E</i> bit is marked.
<b>Size</b>	Displays the size of the DMA transfer. If the command is one of the DMA list commands, this field displays the size of the current DMA list entry. If a given size is not valid (see <a href="#">Table 1</a> ), this field does not necessarily show the invalid size. Instead it shows the number of bytes yet to be transferred when the interrupt is raised.
<b>LstAddr</b>	Displays the local store address of the DMA list if the command is one of the DMA list commands. If partial transfers have been done, this field displays the local store address of the current DMA list entry.
<b>LstSize</b>	Displays the size (in bytes) of the DMA list if the command is one of the DMA list commands. If partial transfers have been done, this field displays the size (in bytes) of

	the list entries that have not been processed. Note that the size of each list entry is 8 bytes.
E	Displays an asterisk (*) if there is an error in the DMA transfer that the hardware detected.

## Examples

This section shows you how to debug some common DMA errors on Cell/B.E systems with example programs that intentionally contain errors. Some of the examples are somewhat contrived so that the errors are fairly easy to see. However, the general approach you use to debug DMA problems on a Cell/B.E. system still applies to most real-world scenarios.

The examples are modified versions of an example that does single buffering using a DMA list command. The example gathers data from main storage to SPE local storage, processes the data, and scatters the data back to main storage using DMA lists. You can find the complete listing for the modified examples in the `/opt/cell/sdk/src/examples/dma_errors/no_error` directory.

### Unaligned effective address

The first version of the example, `unaligned_ea_error`, contains a DMA transfer with an unaligned effective address. You can find the complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/unaligned_ea_error` directory. When the example is compiled and run, the output looks like Listing 2.

#### Listing 2. Output of `unaligned_ea_error`

```
[vto@qdc208 unaligned_ea_error]$ ./dma_error
Bus error
```

A bus error (such as that shown in Listing 2) can be caused by many different error conditions, as shown in [Table 3](#). But which error is the culprit? The first step is to recompile the program with the `-g` and `-O0` flags so that the program can be debugged more easily using the combined `gdb` debugger. Next, run the newly recompiled program using `ppu-gdb`, and you should get the output shown in Listing 3.

#### Listing 3. Finding the bus error

```
[vto@qdc208 unaligned_ea_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
```

```

This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/unaligned_ea_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 16671)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 16671)]
0x000004ac in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:147
147      mfc_read_tag_status_all ();

```

This output shows that the program stopped in the SPU thread. To verify that the error occurred in the SPU, run the command `show architecture` to show the current architecture, as shown in Listing 4.

#### Listing 4. Showing the current architecture

```

(gdb) show architecture
The target architecture is set automatically (currently
spu:256K)

```

In Listing 4, the architecture is `spu:256K`, which shows that the error location is definitely the SPU. Now look more closely at the SPU program in Listing 3. You see that the gdb output shows the program error occurred within a DMA transfer when it stopped on the `mfc_read_tag_status_all` line.

Use the `gdb info spu dma` command to examine all in-flight DMA transfers.

#### Listing 5. Examining all in-flight DMA transfers

```

(gdb) info spu dma
Tag-Group Status 0x00000000
Tag-Group Mask 0x00000001 (undefined query type)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag  TId  RiId  EA                LSA      Size  LstAddr
LstSize E
get      0    0    0    0x0000000010020088 0x21380 0x00020
*
```

In Listing 5, you can scan the shown parameters for effective addresses that are not aligned properly. The local store addresses cannot be examined for alignment errors using the same method because the hardware does not retain the four least significant bits of the local store addresses. The alignment rules and guidelines for DMA commands on Cell/B.E. systems are as follows:

- Source and destination addresses must have the same 4 least significant bits.
- For transfer sizes less than 16 bytes, the address must be naturally aligned (bits 28 through 31 must provide natural alignment based on the transfer size).
- For transfer sizes of 16 bytes or greater, the address must be aligned to at least a 16-byte boundary. In hexadecimal, the address must end with a 0.
- Peak performance is achieved when both source and destination addresses are aligned on a 128-byte boundary (bits 25 through 31 cleared to 0). In hexadecimal, the address must end with an 80 or 00.

Listing 5 shows a DMA with an effective address of 0x10020088. This address is not aligned to a 16-byte boundary and thus causes the observed bus error.

At this point, it might be useful to step through the program to detect the actual DMA that causes the bus error and to examine the effective address in that DMA. [Listing 2](#) shows that the SPE program stopped at line 147 in the SPE `dma_error_spu.c` file. Set a break point at line 140, and start stepping through the program, as shown in Listing 6.

### Listing 6. Stepping through the program

```
[vto@qdc208 unaligned_ea_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:140
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or
[n]) y

Breakpoint 1 (dma_error_spu.c:140) pending.
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/unaligned_ea_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 18864)]
[Switching to Thread 0x400012af230 (LWP 18864)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:143
143     mfc_get (&control_block, argp + 8, sizeof
(control_block_t), tag, 0, 0);
(gdb) print /x argp
$2 = 0x10020080
(gdb) c
```

```
Continuing.
Program received signal SIGBUS, Bus error.
0x00000528 in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:147
147      mfc_read_tag_status_all ();
```

Listing 6 shows that the bus error probably happened because of the DMA transfer (`mfc_get`) on line 143 in the `dma_error_spu.c` file.

Examine the effective address value (`argp + 8`) by printing out the `argp` symbol. Because `argp` has a value of `0x10020080`, `argp + 8` is definitely a misaligned effective address for the DMA command. Changing the effective address parameter to just `argp` fixes the bus error.

## Tag ID errors

All DMA commands except `getllar`, `putllc`, and `putlluc` can be tagged with a five-bit tag group ID (which defines up to 32 IDs). Programs can use this identifier to check for, or wait on the completion of, all queued DMA commands in one or more tag groups. Valid tag group IDs can be any value between 0 and 31, inclusive. Tag group IDs that are not in this range trigger the MFC unit to raise an interrupt to the PPE resulting in the program getting a bus error.

Use of the *tag manager* to reserve and release tag IDs is encouraged to ensure valid DMA tag IDs and to facilitate cooperative use of tag IDs among multiple software components. You can find more information about the tag manager in *C/C++ Language Extensions for Cell Broadband Engine Architecture* (see [Resources](#)).

The example in Listing 7 contains a DMA transfer with a bad tag group ID. The code shows how to walk through a gdb debugging session to detect this problem. You can find the complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/bad_tag_id_error` directory.

Use the `info spu dma` command to detect whether a DMA error has occurred.

Closely examine the DMA transfers shown with the *E* bit marked. The field *Tag* in the `info spu dma` output is a five-bit field, which means that this field does not show when a tag ID is out of range. The recommended way to detect an invalid tag group ID is to examine the inputs to the DMA command immediately preceding the command that issues the DMA request. Examining the inputs requires setting a breakpoint in the SPU source code and printing out the values of the DMA parameters. Listing 7 shows this in an example gdb session.

## Listing 7. Sample gdb session

```
[vto@qdc208 bad_tag_id_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
```

```

Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:145
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or
[n]) y
Breakpoint 1 (dma_error_spu.c:145) pending.
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/bad_tag_id_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 8766)]
[Switching to Thread 0x400012af230 (LWP 8766)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:146
146      mfc_get (&control_block, argp, sizeof
(control_block_t), tag, 0, 0);

(gdb) print tag
$4 = 32

```

In Listing 7, a breakpoint is set to stop at line 145 in the SPU source code, which is before the line where the first DMA transfer is issued. The program is then run using the `r` command. When the breakpoint is hit, the tag ID is displayed using the `print tag` command. Note `tag` is the name of the variable containing the tag ID in this example. The tag ID is 32, which is out of the acceptable range of 0 to 31, inclusive. Changing the tag ID to 0 and recompiling the program enables it to run to completion.

## Transfer size errors

The example code `bad_dma_size_error` contains a DMA transfer with an illegal size. The control block data structure `control_block_t` defined in `dma_error.h` is not padded to be a multiple of 16 bytes. The size of the control block data structure is 24 bytes.

You can find the complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/bad_dma_size_error` directory.

Use the `gdb info spu dma` command to examine all in-flight DMA transfers, as shown in Listing 8.

## Listing 8. Examining in-flight transfers with an illegal size

```

[vto@qdc208 bad_dma_size_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

```

```

This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /home/vto/sdk/src/examples/dma_errors/bad_dma_size_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12328)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 12328)]
0x000004f0 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:147
147      mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status 0x00000000
Tag-Group Mask   0x00000001 (undefined query type)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag  TId  RId  EA                      LSA      Size  LstAddr  LstSize  E
get      0    0    0    0x0000000010020080    0x21280  0x00008

```

The output in Listing 8 shows that there is one pending DMA transfer. The *E* bit is checked on, indicating that there is a DMA transfer error that the hardware detected. Use the [bus error table](#) to check each condition. Following the steps shown in the unaligned effective address example, you can see that the effective address is properly aligned, because the low effective address ends in 0x80. The size of the DMA transfer, however, is reported as 8 (in Listing 8). At first, this appears to be valid, but it is not necessarily the size specified in the DMA command. What is reported here is actually the number of bytes yet to be transferred when the interrupt is raised.

Use the `info symbol ...` command to determine the symbol closest to the LSA specified in the output from Listing 8.

### Listing 9. Determining the closest to LSA-specified symbol

```

(gdb) info symbol 0x21280
control_block in section .bss
(gdb) print &control_block
$1 = (control_block_t *) 0x21280

```

In Listing 9, the gdb output shows that the LSA of 0x21280 is the local address of the variable `control_block` in this DMA transfer. Use gdb to take a closer look at the DMA transfer that fetches the content of the control block from main memory. Because Listing 8 shows that the program breaks around line 147, set a breakpoint to stop at line 140, and step through the program.

### Listing 10. Stepping through the program for a closer inspection

```

[vto@gdc208 bad_dma_size_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.

```

```
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:140
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or
[n]) y

Breakpoint 1 (dma_error_spu.c:140) pending.
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/bad_dma_size_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12169)]
[Switching to Thread 0x400012af230 (LWP 12169)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:143
143     mfc_get (&control_block, argp, sizeof
(control_block_t), tag, 0, 0);
(gdb) print tag
$1 = 0
(gdb) print /x &control_block
$3 = 0x21280
```

The output in Listing 10 shows that the tag ID is 0, which is in the range (between 0 and 31, inclusive). The target local storage address (address of the `control_block` variable) is properly aligned because it ends in 0x80. Next, query for the size of this data structure.

### Listing 11. Querying the size of the data structure

```
(gdb) print sizeof(control_block)
$1 = 24
```

Because the size of the control block in Listing 11 is 24, one of two things has happened:

- The size of the DMA request was set to 24 bytes, which is an error because 24 is not a multiple of 16.
- The DMA request size was set to 32 bytes, which is an error because the block is only 24 bytes in size.

In this example, the request size was set to 24. To fix this error, pad the data structure to be a multiple of 16 bytes by adding two integers at the end of the control block structure. The new data structure in `dma_error.h` is shown in Listing 12.

### Listing 12. The new data structure

```

typedef struct _control_block
{
    unsigned long long in_addr;    //beginning address of the
    input array
    unsigned long long out_addr;  //beginning address of the
    output array
    unsigned int    num_elements_per_spe; //number of elmts
    assigned to this spe
    unsigned int    id;            //spe id
    unsigned int    pad[2];       //pad this data structure to be
    multiple of 16
} control_block_t;

```

The DMA transfer size should now be 32 bytes. After the code is recompiled with these modifications the example runs successfully.

Setting a breakpoint to detect errors in the DMA command can be done easily for programs with only few DMA transfers. For DMA transfers in loops with thousands or more iterations, this technique is not very practical, because it requires you to look at the parameters for thousands or more DMA transfers. In these cases, learn as much from the output of `info spu dma` as possible. If errors are not easily detected from such output, use `gdb` to look at the code listing before and after the bus error to determine the DMA transfer that caused the bus error. Once you detect the DMA command that caused the bus error, look at the inputs for the specific DMA command to find the invalid parameter.

Listing 13 shows a DMA list transfer in which one of the list elements contains an invalid transfer size. The invalid DMA list command is in a loop with 32 iterations. The complete listing of the code for this example can be found in the `/opt/cell/sdk/src/examples/dma_errors/bad_dma_size_loop_error` directory.

### Listing 13. One list element contains an invalid transfer size

```

[vto@qdc208 bad_dma_size_loop_error]$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/bad_dma_size_loop_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 11930)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 11930)]
0x000008a4 in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:210
210         mfc_read_tag_status_all ();
(gdb) info spu dma

```

```

Tag-Group Status 0x00000001
Tag-Group Mask 0x00000001 ('all' query pending)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA          LSA      Size   LstAddr
LstSize E
getl    0   0   0   0x000004000044f080 0x20500 0x00001 0x21618
0x00008 *

```

Listing 13 shows that the program stopped at line 210 in the `dma_error_spu.c` file. The output of the `info spu dma` command indicates that a DMA list transfer caused the bus error. Looking at the parameters in the command, you can see that the size displayed is 1, which is a suspicious value for this parameter. Furthermore, the list size (`LstSize`) parameter is shown as `0x00008`, which means the offending DMA list element is probably at the end of the list. Now take a closer look at the code to determine the actual invalid DMA command.

### Listing 14. The culprit: The invalid command

```

(gdb) list
205      /* issue a DMA get list command to gather the
NUM_LIST_ELEMENT chunks of
      data from system memory */
206      mfc_getl (local_buffer, in_addr, dma_list,
NUM_LIST_ELEMENTS
      * sizeof(mfc_list_element_t), tag, 0, 0);
207
208      /* wait for the DMA get list command to complete */
209      mfc_write_tag_mask (1 << tag);
210      mfc_read_tag_status_all ();
211
212      /* invoke process_data to work on the data that's
just been moved into
      local store*/
213      process_data_simd (local_buffer, CHUNK_SIZE *
NUM_LIST_ELEMENTS);
214

```

Listing 14 shows the source code listing centered on line 210; it is the stopping point of the SPU program. The DMA list command on line 206 seems to be the DMA that caused the bus error. The `dma_list` parameter is shown in Listing 15.

### Listing 15. Looking at the `dma_list` parameter

```

(gdb) print i
$1 = 992
(gdb) print dma_list
$2 = {{notify = 0, reserved = 0, size = 4096, eal = 4391040},
{notify = 0, reserved = 0,
size = 4096, eal = 4395136}, {notify = 0, reserved = 0,
size = 4096, eal = 4399232},
{notify = 0, reserved = 0, size = 4096, eal = 4403328},
{notify = 0, reserved = 0,
size = 4096, eal = 4407424}, {notify = 0, reserved = 0,
size = 4096, eal = 4411520},
{notify = 0, reserved = 0, size = 4096, eal = 4415616},

```

```

{
    notify = 0, reserved = 0, size = 4096, eal = 4419712},
{notify = 0, reserved = 0,
    size = 4096, eal = 4423808}, {notify = 0, reserved = 0,
size = 4096, eal = 4427904},
    {notify = 0, reserved = 0, size = 4096, eal = 4432000},
{notify = 0, reserved = 0,
    size = 4096, eal = 4436096}, {notify = 0, reserved = 0,
size = 4096, eal = 4440192},
    {notify = 0, reserved = 0, size = 4096, eal = 4444288},
{
    notify = 0, reserved = 0, size = 4096, eal = 4448384},
{notify = 0, reserved = 0,
    size = 4096, eal = 4452480}, {notify = 0, reserved = 0,
size = 4096, eal = 4456576},
    {notify = 0, reserved = 0, size = 4096, eal = 4460672},
{notify = 0, reserved = 0,
    size = 4096, eal = 4464768}, {notify = 0, reserved = 0,
size = 4096, eal = 4468864},
    {notify = 0, reserved = 0, size = 4096, eal = 4472960},
{
    notify = 0, reserved = 0, size = 4096, eal = 4477056},
{notify = 0, reserved = 0,
    size = 4096, eal = 4481152}, {notify = 0, reserved = 0,
size = 4096, eal = 4485248},
    {notify = 0, reserved = 0, size = 4096, eal = 4489344},
{notify = 0, reserved = 0,
    size = 4096, eal = 4493440}, {notify = 0, reserved = 0,
size = 4096, eal = 4497536},
    {notify = 0, reserved = 0, size = 4096, eal = 4501632},
{
    notify = 0, reserved = 0, size = 4096, eal = 4505728},
{notify = 0, reserved = 0,
    size = 4096, eal = 4509824}, {notify = 0, reserved = 0,
size = 4096, eal = 4513920},
    {notify = 0, reserved = 0, size = 4097, eal = 4518016}}

```

Listing 15 shows the content of the `dma_list` parameter. A close look at the content of the list elements at the end of the list confirms suspicion. The last list element has a size of 4097, which is not a valid value for a DMA transfer. Changing the size of the last list element to a multiple of 16 bytes fixes the bus error.

### Unaligned local store address

You can detect unaligned local store addresses using `gdb` in a similar manner to the techniques shown already. The following example contains a DMA transfer with a misaligned SPE local store address. You can find a complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/unaligned_isa_error` directory.

Once the program execution breaks due to a bus error, you can use the command `info spu dma` to determine if any of the DMA commands have an error. Examine closely the DMA transfers shown with the *E* bit turned on. The `gdb` output in Listing 16 shows the result of running the example and the command `info spu dma`.

### Listing 16. Running `info spu dma` again

```
[vto@qdc208 unaligned_lsa_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/unaligned_lsa_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 10793)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 10793)]
0x000004f0 in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:152
152      mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status 0x00000000
Tag-Group Mask   0x00000001 (undefined query type)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA          LSA      Size  LstAddr
LstSize E
get     0  0  0  0x0000000010020080 0x21300 0x00020
*
```

From Listing 16, it is not obvious which of the four DMA parameters cause an error: tag ID, effective address, local store address, or size.

### Listing 17. Eliminating the misbehaving parameters

```
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/unaligned_lsa_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12581)]
[Switching to Thread 0x400012af230 (LWP 12581)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:140
140      if (tag == MFC_TAG_INVALID)
(gdb) step 1
148      mfc_get (&control_block_data[4], argp, sizeof
(control_block_t), tag, 0, 0);
(gdb) print tag
$1 = 0
(gdb) print sizeof(control_block_t)
$2 = 32
(gdb) print /x &control_block_data[4]
$4 = 0x21304
```

Following the steps shown in the previous examples, you can quickly eliminate the effective address, the tag ID, and the transfer size as potential invalid DMA parameters (see Listing 17). The local store address (`&control_block_data[4]`)

however, ends with 0x04. This address is not aligned on a 16-byte boundary. This is the cause of the bus error.

## Segmentation faults

DMA transfers to or from bad effective addresses can cause segmentation faults. Debugging segmentation faults caused by invalid effective addresses is similar to the techniques used to debug segmentation faults in other types of code. The following example shows a gdb debugging session to resolve a segmentation violation resulting from a bad effective address in a DMA list element. You can find a complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/bad_eal_in_dma_list_error` directory.

The example session within gdb begins in Listing 18.

### Listing 18. gdb debugging session to resolve a bad effective address segmentation violation

```
[vto@qdc208 bad_eal_in_dma_list_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law. Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/bad_eal_in_dma_list_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x40000aef230 (LWP 11714)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x40000aef230 (LWP 11714)]
0x000006d4 in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:181
181      mfc_read_tag_status_all ();
```

Load and start the program with the `r` command. When the segmentation fault occurs, gdb stops execution. Next issue a standard backtrace command using the `bt` command to display the stack frames that led to the segmentation fault, as shown in Listing 19.

### Listing 19. Backtracing to find the stack frames leading to the fault

```
(gdb) bt
#0 0x000006d4 in main (speid=268566544, argp=268566656,
envp=0) at dma_error_spu.c:181
#1 0x0000008c in _start () from dma_error_spu@0x10001780 <5>
#2 <cross-architecture call>
#3 0x000000800bdeff00 in .syscall () from /lib64/libc.so.6
```

```
#4 0x0000040000003567c in ._base_spe_context_run () from
/usr/lib64/libspe2.so.2
#5 0x00000400000029e24 in .spe_context_run () from
/usr/lib64/libspe2.so.2
#6 0x0000000010000c1c in ppu_thread_function
(argp=0xfffffc8ea38) at dma_error.c:69
#7 0x000000800c00bcf0 in .start_thread () from
/lib64/libpthread.so.0
#8 0x000000800bdf49fc in .__clone () from /lib64/libc.so.6
```

Listing 19 does not show any obvious problems, so issue the `info spu dma` command to see if there is an error in the DMA commands, as shown in Listing 20.

## Listing 20. Looking for DMA command errors

```
(gdb) info spu dma
Tag-Group Status 0x00000001
Tag-Group Mask 0x00000001 (undefined query type)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag  TId  RId  EA                LSA      Size  LstAddr
LstSize E
getl    0    0    0    0x0000040000000000  0x09300  0x01000  0x11360
0x00040
```

There is only one outstanding DMA list transfer shown in the output, and the *E* bit is not turned on. This indicates that all the input parameters to the DMA command are valid. Checks for transfer size, alignment, and list element crossing 4GB boundary problems are not needed.

The size of this DMA list transfer (`LstSize`) is shown as `0x40` bytes (64 bytes). Examining the source code shows the DMA list has a total size of 128 bytes (16 entries \* 8 bytes/entries). These two facts mean half of the DMA list has been processed successfully, and the problem occurs in the second half of the DMA list.

Using this clue, take a closer look at the DMA list.

## Listing 21. Closer look to find the segmentation violation

```
(gdb) print /x dma_list
$1 = {{notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0x50080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0x51080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0x52080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0x53080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0x54080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0x55080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0x56080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0x57080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x0},
{notify = 0x0,
```

```

    reserved = 0x0, size = 0x1000, eal = 0x59080}, {notify =
0x0,
    reserved = 0x0, size = 0x1000, eal = 0x5a080}, {notify =
0x0, reserved = 0x0,
    size = 0x1000, eal = 0x5b080}, {notify = 0x0,
    reserved = 0x0, size = 0x1000, eal = 0x5c080}, {notify =
0x0, reserved = 0x0,
    size = 0x1000, eal = 0x5d080}, {notify = 0x0,
    reserved = 0x0, size = 0x1000, eal = 0x5e080}, {notify =
0x0, reserved = 0x0,
    size = 0x1000, eal = 0x5f080}}

```

The gdb output shows the EAL (the 32-bit, low-order effective address) of the ninth DMA list entry as 0. This causes the segmentation violation. Looking through the code reveals that the `fill_dma_list` function (lines 112 to 115) sets the EAL of the ninth entry to 0. Remove this part of the code, recompile the program, and the program runs successfully.

The example is a little bit contrived because it purposely sets the EAL of a specific DMA list element to an invalid value. However, the general approach used to debug a segmentation fault still applies to most real-world scenarios.

### List element crossing 4GB boundary

The Cell Broadband Engine Architecture (CBEA) specifies that the EAL (the 32-bit, low-order effective address) for each list element in a DMA list must be in the 4GB aligned area defined by the EAH (the 32-bit high-order effective address). Although each EAL starting address is in a single 4GB area, a list element transfer might cross the 4GB boundary.

However, in the Cell/B.E. and IBM PowerXCell™ 8i processors, a DMA list element that crosses a 4GB boundary results in a Class0 DMA Alignment Error exception. The Linux® operating system makes no effort to detect or recover from this error. Therefore, having a list element crossing a 4GB boundary in a DMA list results in a bus error during execution. This error occurs only for 64-bit applications, because 32-bit applications cannot address 4GB of storage.

In Listing 22, the `mmap( )` function is used to force the allocation of the input data in `dma_error.c` to straddle a 4GB boundary. Additionally, the DMA list constructed on the SPU contains a list element that crosses the 4GB boundary. The example results in an interrupt being raised to the PPE, which causes a bus error. You can find the complete listing of the code for this example in the `/opt/cell/sdk/src/examples/dma_errors/4GB_crossing_error` directory.

In Listing 22, the code is run under gdb. Once the program execution breaks because of the bus error, the `info spu dma` command is issued. Listing 22 shows the output from gdb.

### Listing 22. Output shows error in LstAddr

```
[vto@gdc208 4GB_crossing_error]$ ~/ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law. Type
"show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program:
/home/vto/sdk/src/examples/dma_errors/4GB_crossing_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x40000e9f230 (LWP 865)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x40000e9f230 (LWP 865)]
0x00000694 in main (speid=268566544, argp=268566656, envp=0) at
dma_error_spu.c:176
176      mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status 0x00000001
Tag-Group Mask 0x00000001 (undefined query type)
Stall-and-Notify 0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA          LSA      Size    LstAddr
LstSize E
getl    0  0  0  0x00000001ffffff080 0x08280 0x01000 0x092d8
0x00008 *
(gdb) info symbol 0x08280
local_buffer + 28672 in section .bss
(gdb) info symbol 0x092d8
dma_list + 56 in section .bss
```

The output shows an error in the DMA get list command. Use the `info symbol ...` command to look up the symbols associated with the LSA and the LstAddr. Partial transfers have been completed, and the LSA and LstAddr are not the same as the original LSA and LstAddr in the MFC command.

Next, take a closer look at the actual DMA list, as shown in Listing 23.

### Listing 23. Looking at the actual DMA list

```
(gdb) print /x dma_list
$1 = {{notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0xffff8080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0xffff9080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0xffffa080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0xffffb080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0xffffc080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0xffffd080}, {
notify = 0x0, reserved = 0x0, size = 0x1000, eal =
0xffffe080}, {notify = 0x0,
reserved = 0x0, size = 0x1000, eal = 0xfffff080}}
```

Listing 23 shows the content of the current DMA list. The last DMA entry has an EAL of 0xffff080 and a size of 0x1000. This means the list entry actually crosses the 4GB boundary. Changing either the allocation of the original input data or issuing another individual DMA transfer for the last DMA list entry fixes the problem.

For more information explaining the limitations of DMA list elements crossing 4GB boundaries, refer to the SDK programmer's guide (with more in the newly released Version 3.1).

## Understanding DMA race conditions

Debugging race conditions caused by DMA transfers on Cell/B.E. systems is quite a difficult task. The IBM SDK for Multicore Acceleration Version 3.1 provides some utilities that help you find the causes.

The race check library delivers a set of routines that support the software detection of frequently encountered race conditions involving local store data transfers and SPE local storage accesses. These race conditions occur as a result of the indeterminate ordering of the transactions performed on the local store memory.

You can also use the IBM Full System Simulator for the Cell Broadband Engine to detect potential race conditions in SPU programs.

# Resources

## Learn

- Refer to Chapter 7 of [Cell Broadband Engine Architecture](#) (IBM, October 2007, v1.02) for all the available MFC commands and the appropriate values for each of the parameters in the commands.
- Refer to Chapter 19 of the [Cell BE Programming Handbook Including PowerXCell 8i](#) (IBM, May 2008, v1.11) for information about how to use the DMA transfers on Cell/B.E. systems.
- Refer to Chapter 3 of the [Programmer's Guide for the SDK for Multicore Acceleration version 3.0](#) (IBM, October 2007, v3.0) for information about how to debug Cell/B.E. applications using GDB. (This article is also appropriate for the Version 3.1 of the SDK and this guide.)
- Check out the [Cell BE Programming Tutorial](#) (IBM, October 2007, v3.0) if you are interested in developing applications or libraries.
- Refer to Chapter 11 of the [Example Library API Reference](#) (IBM, October 2007, v3.0) for detailed information about how to use the race check library to detect race conditions.
- Refer to the document [PPU & SPU C/C++ Language Extensions for Cell Broadband Engine Architecture](#) (IBM, September 2007, v2.5) for an excellent source of more information about the tag manager.
- Learn more about Cell/B.E. programming from the developerWorks series:
  - ["Programming high-performance applications on the Cell/B.E. processor"](#)
  - ["PS3 fab-to-lab"](#)
  - ["The little broadband engine that could"](#)
- Refer to the [Cell Broadband Engine documentation](#) section of the IBM Semiconductor Solutions Technical Library for a wealth of downloadable manuals, specifications, and more.
- Sign up for the [developerWorks newsletter](#) and get the latest developer news and Cell/B.E. happenings delivered to your inbox each week. Check *Power Architecture*® when you sign up to receive Cell/B.E. news in your newsletter.

## Get products and technologies

- Use the [IBM Full-System Simulator for the Cell Broadband Engine Processor](#) to help you detect potential race conditions in SPU programs.
- Get your copy of the [IBM SDK for Multicore Acceleration 3.1](#) or browse through the [library of Cell/B.E. documentation](#).

- Find all Cell/B.E.-related articles, discussion forums, downloads, and more at the IBM developerWorks [Cell Broadband Engine resource center](#): your definitive resource for all things Cell/B.E.
- Contact IBM about [custom Cell/B.E.-based or custom-processor based solutions](#).

## Discuss

- [Participate in the discussion forum for this content](#).
- Check out the [Cell Broadband Engine Architecture forum](#) to get your technical questions about the processor answered. Juicy problems and answers from the forums are rounded up periodically and highlighted in the ["Forum watch" blog series](#).
- Go to the [Cell Broadband Engine/Power Architecture blog](#) for [news](#), downloads, instructional resources, and event notifications for Cell/B.E. and other Power Architecture-related technologies. You can find the popular ["Forum watch"](#) blog series (Q&A roundup), the ["FixIt"](#) technology updates, and the [Infobomb](#) quick-read technology introductions.

## About the author

VanDung To

VanDung To received a computer science degree from Rice University in 2002 then joined IBM shortly afterwards. She is an advisory software engineer in the Quasar Design Center at IBM, and she has been working with Cell/B.E. technology since 2002.

## Trademarks

IBM, developerWorks, Power Architecture, PowerPC, and PowerXCell are trademarks of IBM Corporation in the United States, other countries, or both. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc. Other company, product, or service names may be trademarks or service marks of others.