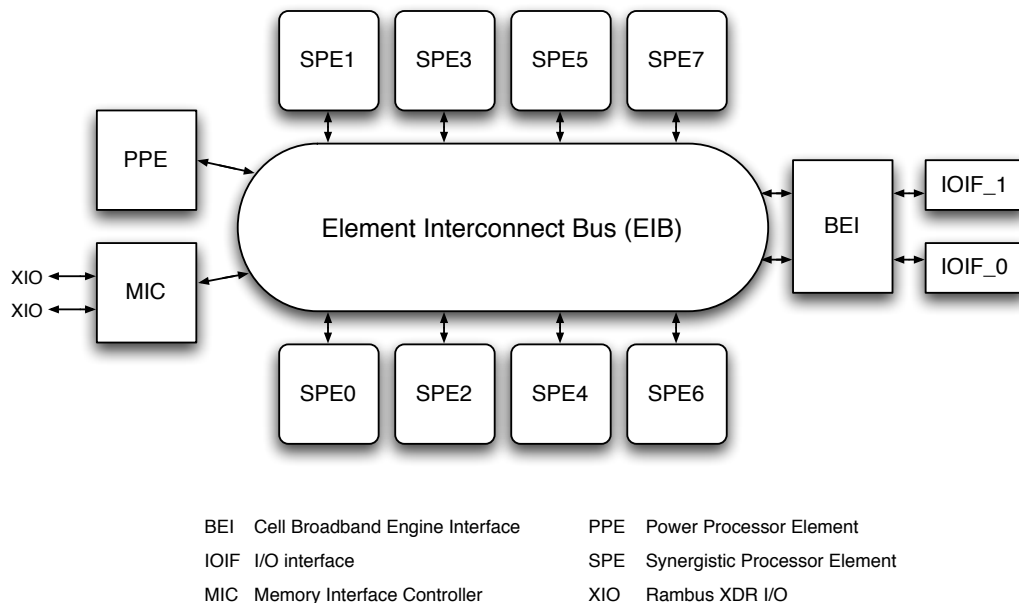


DIKU Cell Tutorial

Martin Rehr
Department of Computer Science
University of Copenhagen
rehr@diku.dk

1 Introduction

The Cell processor is a heterogeneous multi-core processor consisting of a Power Processor Element (PPE) and eight Synergistic Processing Element (SPE) all connected by an Element Interconnect Bus (EIB). An overview of the Cell architecture is shown in figure 1. The PPE is



Figur 1: An overview of the Cell architecture

the main core of the Cell processor which communicates with the operating system and there through the executing applications. The architecture of the PPE is the same as used in the IBM PowerPC series which makes it possible to use a standard Linux PowerPC distribution with the Cell processor.

A simplified overview of the Cell PPE architecture is shown in figure 2.

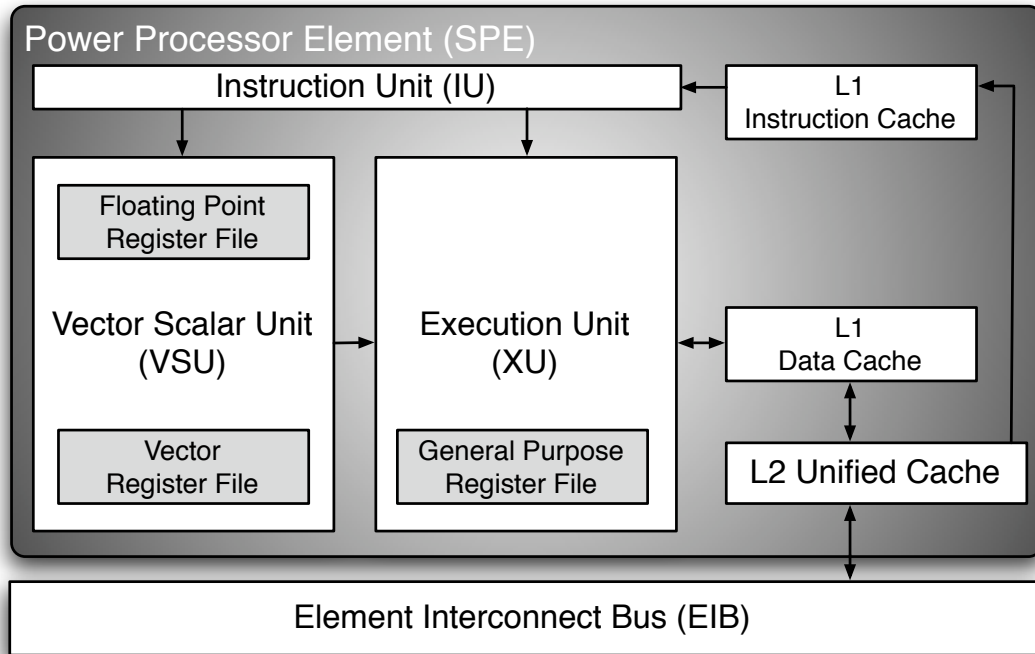
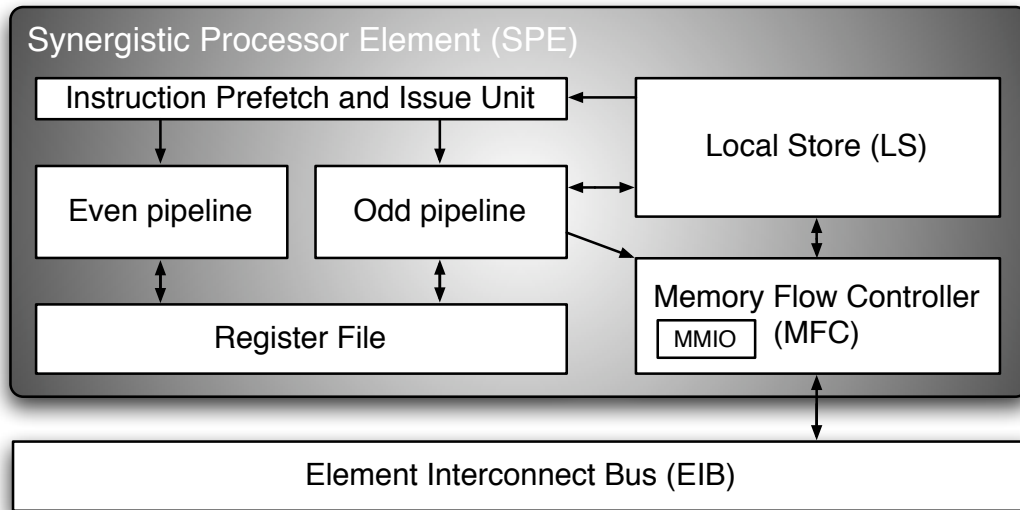


Figure 2: An overview of the Cell PPE architecture

The PPE of the Cell has the following specifications:

- 64 bit RISC processor with SIMD extensions
- 2 hardware threads
- Dual pipelined
 - XU: Fixed point and Load/Store instructions
 - VSU: Floating point and Vector instructions
- 32 kB data and instruction cache
- 512 kB unified L2 Cache
- 32x64 bit general purpose registers per thread
- 32x64 bit floating point registers per thread
- 32x128 bit vector SIMD registers per thread

The eight SPE's are the interesting parts of the Cell processor, these are powerful working cores designed for high computation throughput. A simplified overview of the Cell SPE architecture is shown in figure 3.



Figur 3: An overview of the Cell SPE architecture

Each SPE has the following specification:

- 128 bit RISC SIMD vector processor
- Dual pipelined
 - Even: Computation pipeline
 - Odd: Management pipeline
- 128x128 bit (2kB) general purpose Register File
- 256 kB Local Store (LS) (cache speed memory)
- Memory Flow Controller (MFC)
 - Provides asynchronous DMA transfers between main memory and LS

As it's seen the SPE's of the Cell has an architecture which differs greatly from the traditional architectures we know.

The key differences are:

- No direct access to main memory
 - All access to Main memory through the MFC
 - Both data and code must reside in the LS
- No cache mechanism
- No branch prediction
- Single threaded
- All instructions operates on Vectors
 - Scalar operations are slow
- Dual instructions
 - Due to the pipeline layout it's possible to issue 2 instructions in one clock cycle

Last but not less, the PPE and SPE's are connected through the Element Interconnect Bus (EIB) which has a peek bandwidth of 204 GB/s at 3.2 GHz, the current operating frequency of the Cell processor. For a more detailed description of the Cell processor, one is encouraged to take a look at the master thesis by Mohammad Jowkar:

- http://www.diku.dk/~rehr/cell/docs/mohammad_jowkar_thesis.pdf

2 Getting started

The Cell processor is a 8 node cluster on a chip, and programming it to gain max performance requires hard work, as well as skill and knowledge of distributed computing. However getting started is not so hard as C compiles and libraries exists both for the PPE and the SPE's. Your ANSI-C code compiles out of the box for the PPE and you are up and running, however the fun begins with the SPE's. To develop Cell applications you need the IBM Cell SDK, this includes compilers both for the PPE and SPE's as well as libraries, debuggers, examples and a Cell simulator, which you can't live without when developing Cell applications. The Cell SDK is available for the X86 architecture, which means you don't need a Cell based machine to get started, actually the only reason to do real hardware is to make performance tests. This tutorial is based on the IBM Cell SDK 2.1 and the GCC compiler, but the code represented works with the IBM XLC compiler as well.

3 Hello World

In this section various versions of a Hello World program is shown, starting with the simple PPE/SPE versions, where no intercommunication occurs between the PPE and the SPE, proceeding with the PPE starting a SPE printing "Hello World", ending up with the PPE starting all SPE's which each prints "Hello World".

3.1 Simple PPE

A hello world program for the PPE is straight forward:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     printf("Hello World, from the PPE!\n");
6     return 0;
7 }
```

This is compiled by the GCC compiler for the PPU architecture:

```
ppu-gcc -m64 -o hello_ppe hello_ppe.c
```

3.2 Simple SPE

Similar a Hello World for the SPE look like this:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World, from a SPE.\n");
6     return 0;
7 }
```

This is compiled by the GCC compiler for the SPU architecture:

```
spu-gcc -o hello_spe hello_spe.c
```

3.3 SPE started by PPE

The first two examples was the easy part, now we want to start a SPE's from the PPE and make it print "Hello World". The SPE code is quite similar to the code printing "Hello World" in the example above, the only difference is the 'speid' which is used to identify the SPE¹.

```

1 #include <stdio.h>
2
3 int main(unsigned long long speid)
4 {
5     printf("Hello _SPE_(0x%llx)\n", speid);
6     return 0;
7 }

```

The PPE is the interesting part of this example, the code is shown and explained below:

```

1 #include <libspe2.h>
2
3 /* This is the pointer to the SPE code to be executed */
4 extern spe_program_handle_t hello_spehandle;
5
6 /* This is the handle used by the PPE to control the SPE */
7 spe_context_ptr_t speid;
8
9 /* The initial value of the SPU's instruction pointer */
10 unsigned int entry = SPE_DEFAULT_ENTRY;
11
12 int main()
13 {
14     /* Create context */
15     speid = spe_context_create(0, NULL);
16
17     /* Load program into context */
18     spe_program_load(speid, &hello_spehandle);
19
20     /* Give control of this thread to the SPE,
21        this blocks until the SPE returns
22        and control is given back to the PPE */
23     spe_context_run(speid, &entry, 0, NULL, NULL, NULL);
24
25     return(0);
26 }

```

The IBM Cell SDK provides C libraries to operate the SPE's from the PPE, the header file for those is included in line 1. Even though the PPE and SPE architectures are different their binaries can be compiled into one executable ELF file. In line 4 a handle is declared which is used to identify the SPE code within that binary.

¹Actually it's the main: memory address where the PPE stores the information it needs to control and communication with each SPE.

In order to control the SPE's, the PPE need a handle to a control structure keeping track of the SPE's execution information, this is declared in line 7. When the SPE is started, information about which instruction to fetch first is given to the SPE from the PPE, in this example we use the default entry declared in line 10.

This was the declaration of control structs, now we proceed with the main function and the actual execution. The SPE control struct needed by the PPE is created in line 15, line 18 takes care of loading the actual SPE instructions into main memory, note the "hello_spehandle", which indicates the location of the SPE code in the binary, is used here. When the SPE code is loaded into main memory it's time to start the execution on the SPE, this is done in line 23. When control is given to the SPE, the PPE thread blocks until the SPE returns. When the SPE returns the PPE returns from it's main loop with exit code 0, this is done in line 25.

To compile the "Hello World from SPE started by PPE" do the following:

```
1 ppu-gcc -m64 -c hello_ppe.c
2 spu-gcc -o hello_spe hello_spe.c
3 ppu-embedspu -m64 hello_spehandle hello_spe hello_spe_embedded.o
4 ppu-gcc -m64 -o hello hello_ppe.o hello_spe_embedded.o -lspe2
```

First the PPE code is compiled in line 1, thereafter the SPE code is compiled in line 2. The ppu-embedspu used in line 3 embeds the SPE code in a ELF object file and gives it the label "hello_spehandle" used to identify the SPE code within the final binary. The final binary is created in line 4 where the PPE and SPE code is linked together.

3.4 All SPE's started by the PPE

As mentioned in the last example the PPE thread is suspended while the SPE executes which means only one SPE can be started from the PPE main thread. This example shows how it's possible to start all available SPE's using POSIX threads. The SPE code is the same as in the last example, all modifications resides within the PPE code:

```

1 #include <libspe2.h>
2 #include <pthread.h>
3
4 #define SPE_THREADS 8
5
6 /* This is the pointer to the SPE code to be executed */
7 extern spe_program_handle_t hello_spehandle;
8
9 /* This is the handle used by the PPE to control the SPE */
10 spe_context_ptr_t speid [SPE_THREADS];
11
12 /* The initial value of the SPE's instruction pointer */
13 unsigned int entry [SPE_THREADS];
14
15 void *start_spe_thread(void *arg) {
16     int i;
17     i = ((int) arg);
18
19     /* Set the initial value of the SPE's instruction pointer */
20     entry [i] = SPE_DEFAULT_ENTRY;
21
22     /* Give control of this thread to the SPE,
23      * this block until the SPE returns
24      * and control is given back to the PPE */
25     spe_context_run(speid [i], &entry [i], 0, NULL, NULL, NULL);
26
27     pthread_exit(NULL);
28 }

```

The main function is show on next page.

```

29 int main()
30 {
31     int i;
32     pthread_t pthreads[SPE_THREADS];
33
34     /* Create SPE threads to execute 'hello world' */
35     for(i=0; i<SPE_THREADS; i++) {
36         /* Create context */
37         speid[i] = spe_context_create(0, NULL);
38
39         /* Load program into context */
40         spe_program_load(speid[i], &hello_spehandle);
41
42         /* Create POSIX thread which starts SPE execution */
43         pthread_create(&pthreads[i], NULL, &start_spe_thread,
44                       (void *)i);
45     }
46
47     /* Wait for SPE threads to finish */
48     for (i=0; i<SPE_THREADS; i++) {
49         pthread_join(pthreads[i], NULL);
50     }
51     return (0);
52 }

```

As it's seen this example is roughly the same as the last one, the only differences is the usage of arrays for for the data-types needed by the PPE to control the SPE's, and the usage of POSIX threads line 15, 43 and 49.

The function "start_spe_thread" declared line 15, is called by the "pthread_create" function line 43 which spawns a new POSIX thread with "start_spe_thread" as start function. The control is then handed over to the SPE, and the PPE POSIX thread is suspended until the SPE execution terminates. As all SPE threads starts out as PPE POSIX threads, a barrier is needed to prevent the PPE main thread from terminating before the SPE's are finished executing, as this would result in an abnormal termination of the SPE threads. The "pthread_join" line 49 provide such a barrier.

This example is compiled the same way as the previous one, section 3.3 page 7

4 Communication between the PPE and SPE's

There are four ways of communication between the PPE and the SPE's

- Signals
PPE \rightarrow SPE
- Mailbox messages
PPE \leftrightarrow SPE
- DMA transfers
PPE \leftrightarrow SPE
- Arguments to the SPE main function
PPE \rightarrow SPE
Used for initial DMA transfers

For simplicity, only one SPE is used in the examples covered in this section.

4.1 Signals and Mailboxes

Signal and mailbox messages are both seen from a SPE centric point of view, and is represented as physical registers (MMIO) at the MFC see figure 3 page 3. The SPE access it's own MMIO locally, where the PPE and SPE's access remote MMIO registers by the EIB. Mailbox and signal communication between SPE's is not covered in this tutorial, as this has to be done explicitly by DMA transfers, opposed to the built-in functions of the SDK presented in this section.

4.1.1 Signals

Each SPE has two 32 bit signal registers which can be written by the PPE through the EIB and read locally by the SPE. The default policy for signals is that old signals are overwritten when new signals arrives. The default policy can be changed to OR'ing the signals received instead of overwriting. By this a many-to-one signaling can be achieved. A many-to-one signaling scheme requires the SPE's to signal each other through DMA this is out of the scope for this tutorial and therefore the OR'ing scheme is not covered. An example of using signals is shown below, the PPE code is presented on the next page.

```

1 #include <libspe2.h>
2 #include <pthread.h>
3
4 /* This is the pointer to the SPE code to be executed */
5 extern spe_program_handle_t signal_spehandle;
6
7 /* This is the handle used by the PPE to control the SPE */
8 spe_context_ptr_t speid;
9
10 /* The initial value of the SPE's instruction pointer */
11 unsigned int entry = SPE_DEFAULT_ENTRY;
12
13 void *start_spe_thread(void *arg) {
14     /* Give control of this thread to the SPE,
15      * this block until the SPE returns
16      * and control is given back to the PPE */
17     spe_context_run(speid, &entry, 0, NULL, NULL, NULL);
18
19     pthread_exit(NULL);
20 }
21
22 int main()
23 {
24     unsigned int message;
25     int status;
26     pthread_t pthread;
27
28     /* Create context */
29     speid = spe_context_create(0, NULL);
30
31     /* Load program into context */
32     spe_program_load(speid, &signal_spehandle);
33
34     /* Create POSIX thread which starts SPE execution */
35     pthread_create(&pthread, NULL, &start_spe_thread, NULL);
36
37     /* Send signal to register 1 */
38     message = 100;
39     status = spe_signal_write(speid, SPE_SIG_NOTIFY_REG.1, message);
40     printf("PPE_signal1_status: %i\n", status);
41
42     /* Send signal to register 2 */
43     message = 200;
44     status = spe_signal_write(speid, SPE_SIG_NOTIFY_REG.2, message);
45     printf("PPE_signal2_status: %i\n", status);
46
47     /* Wait for SPE thread to finish */
48     pthread_join(pthread, NULL);
49
50     return (0);

```

The lines 1 to 35 are roughly the same as in the previous examples. The real action occurs in the lines 38 and 43 where a message is send from the PPE to the SPE's signal register 1 and 2, messages send are all 32 bit unsigned integers. The first parameter to the function "spe_signal_write" is the speid (section 3.3 page 6), the second parameter is a constant telling which signal register to use, the third parameter is the message send to SPE. The function is NON-blocking and the return code can be one of the following:

- -1: Error
- 0: Ok

The code for receiving the the signal register messages on the SPE looks like this:

```

1 #include <spu_mfcio.h>
2 #include <stdio.h>
3
4 int main(unsigned long long speid)
5 {
6     unsigned int message;
7
8     message = spu_read_signal1 ();
9     printf("SPE_signal1_message: %u\n", message);
10
11    message = spu_read_signal2 ();
12    printf("SPE_signal2_message: %u\n", message);
13
14    return 0;
15 }

```

The message in signal register 1 is read in line 8 and the message in signal register 2 is read in line 11, note that the "spu_read_signal" performs a atomic read + clean of the register, this way the same message can't be read twice. As the "spu_read_signal" function is blocking functions to check if data has arrived at the signal registers is provided:

- status = spu_stat_signal1();
- status = spu_stat_signal2();

These returns:

- 0: If no pending signals
- 1: If pending signals

4.2 Mailboxes

The main difference between mailboxes and signals is the fact that mailboxes are unidirectional and blocked when filled, opposed to signals where messages are overwritten or OR'ed together. Each SPE has an associated inbox and outbox as with signals the mailboxes are seen from the SPE's point of view and the registers are located physically at each SPE core. Each SPE has an associated inbox and outbox, the inbox is where messages for the SPE is put by the other cores through the EIB. The outbox is where the SPE writes messages for the other cores to read through the EIB. The mailboxes has the following properties:

Attribute	Inbox	Outbox
Direction	Messages from the PPE ^a	Messages to the PPE ^a
Read	Local SPE	PPE ^a
Write	PPE ^a	Local SPE
#Mailboxes	1	2
#Entries	4	1
Entry type	32 bit unsigned int	32 bit unsigned int
Buffer type	FIFO	FIFO
Buffer empty	Local SPE blocks on read	Depends on PPE's read method ^b
Buffer full	Depends on PPE's write method ^b	Local SPE blocks on write

^aRead/Write between SPE's is possible but not covered in this tutorial

^bThe different PPE read/write methods is described in detail later in this section

4.2.1 Inbox

The Cell SDK provides a framework for communicating with the SPE inboxes, an overview of the available functions is shown in the following table:

Description	PPE function	Blocking	SPE function	Blocking
Header file	libspe2.h		mpu_mfcio.h	
Read			spu_read_in_mbox ^a	Yes
Write	spe_in_mbox_write ^b	User ^c		
Status	spe_in_mbox_status ^d	No	spu_stat_in_mbox ^e	No

^a *uint32_t spu_read_in_mbox(void)*

Returns next entry from local SPE inbox, blocks until entry available

^b *int spe_in_mbox_write(
spe_context_ptr_t spe,
unsigned int *mbox_data,
int count,
unsigned int behavior)*

Writes *count* entries from *mbox_data* to *spe* inbox, returning:

- 1: Error
- 0: Nothing written
- >0: Number of entries written

^c The blocking behavior of ^b is defined by the *behavior* parameter:

SPE_MBOX_ALL_BLOCKING

Blocks until *count* messages has been written

SPE_MBOX_ANY_BLOCKING

Blocks until at least one of *count* messages has been written

SPE_MBOX_ANY_NONBLOCKING

Try to write as many of *count* messages as possible without blocking

^d *int spe_in_mbox_status (spe_context_ptr_t spe)*

Returns number of free entries in the *spe* inbox

^e *uint32_t spu_stat_in_mbox(void)*

Returns number of entries in local SPE inbox ready to be read by local SPE

An example of writing to the SPE's inbox from the PPE is shown on the next page

```

1 #include <libspe2.h>
2 #include <pthread.h>
3
4 /* This is the pointer to the SPE code to be executed */
5 extern spe_program_handle_t mailbox_spehandle;
6
7 /* This is the handle used by the PPE to control the SPE */
8 spe_context_ptr_t speid;
9
10 /* The initial value of the SPE's instruction pointer */
11 unsigned int entry = SPE_DEFAULT_ENTRY;
12
13 void *start_spe_thread(void *arg) {
14     /* Give control of this thread to the SPE,
15      * this block until the SPE returns
16      * and control is given back to the PPE */
17     spe_context_run(speid, &entry, 0, NULL, NULL, NULL);
18
19     pthread_exit(NULL);
20 }
21
22 int main()
23 {
24     int i;
25     unsigned int message;
26     int status;
27     pthread_t pthread;
28
29     /* Create context */
30     speid = spe_context_create(0, NULL);
31
32     /* Load program into context */
33     spe_program_load(speid, &mailbox_spehandle);
34
35     /* Create POSIX thread which starts SPE execution */
36     pthread_create(&pthread, NULL, &start_spe_thread, NULL);
37
38     /* Write message to the SPE's inbox */
39     message = 100;
40     status = spe_in_mbox_write(speid, &message, 1,
41                               SPE_MBOX_ALL_BLOCKING);
42     printf("Status: %i\n", status);
43
44     /* Wait for SPE thread to terminate */
45     pthread_join(pthread, NULL);
46
47     return (0);
48 }

```

The "spe_in_mbox_write" line 40 is responsible for the actual write to SPE's inbox, the first parameter is the speid (section 3.3 page 6), the second is the message to be send (32 bit unsigned int), the third parameter is the number of entry's to write to the inbox, and the last parameter is the behavior of the "spe_in_mbox_write". The behavior can be one of the following:

- SPE_MBOX_ALL_BLOCKING
Blocks until all messages^a are written to inbox
- SPE_MBOX_ANY_BLOCKING
Blocks until at least one of the messages^a is written to the inbox
- SPE_MBOX_ANY_NONBLOCKING
Tries to write as many of the messages^a to the inbox

^a The number of messages is given by the third parameter *count* to "spe_in_mbox_write"

The return code from "spe_in_mbox_write" is one of the following:

- -1: Error
- 0: Nothing written
- >0: Number of entries written

Sending several messages in a row to the inbox can be done by calling "spe_in_mbox_write" with an array of 32 bit unsigned int in this fashion:

```

1 /* Write message to the SPE's inbox */
2 unsigned int message[2]
3 message[0] = 100;
4 message[1] = 200;
5 status = spe_in_mbox_write(speid, &message[0], 2,
6                             SPE_MBOX_ALL_BLOCKING);

```

This will generate two entries (message[0] and message[1]) and send them to the SPE. The call to "spe_in_mbox_write" will block until both entries are delivered to the SPE's inbox.

It should be noted that the IBM Cell simulator only supports one message per "spe_in_mbox_write" (count=1) with a blocking nature (SPE_MBOX_ALL_BLOCKING) whereas real hardware supports all combinations of count and behavior.

Reading the inbox messages at the SPE side is quite straight forward, the code is shown next

```

1 #include <spu_mfcio.h>
2 #include <stdio.h>
3
4 int main(unsigned long long speid)
5 {
6     unsigned int message;
7
8     printf("Waiting_for_inbox_message\n");
9     message = spu_read_in_mbox();
10    printf("SPE_got_inbox_message:_%i\n", message);
11
12    return 0;
13 }

```

The "spu_read_in_mbox()" function is used to get the next entry in the inbox FIFO queue, it blocks until data is read, if one wish to check the inbox for new messages without blocking, the "spu_stat_in_mbox" can be use, this function returns the amount of inbox messages in the FIFO.

4.3 Outboxes

Each SPE has two outboxes registers capable of containing one 32 bit unsigned int entry per mailbox. One of the outboxes is a so called interrupting mailbox (INTR) which means an interrupt is raised upon a read/write to the outbox. An overview of the Cell SDK functions used to operate the outboxes is shown in the table below.

Description	PPE function	Blocking	SPE function	Blocking
Header file	libspe2.h		mpu_mfcio.h	
Read	spe_out_mbox_read ^a	No		
Write			spu_write_out_mbox ^b	Yes
Status	spe_out_mbox_status ^c	No	spu_stat_out_mbox ^d	No
Read INTR	spe_out_intr_mbox_read ^e	User ^f		
Write INTR			spu_write_out_intr_mbox ^g	Yes
Status INTR	spe_out_intr_mbox_status ^h	No	spu_stat_out_intr_mbox ⁱ	No

^a *int spe_out_mbox_read(spe_context_ptr_t spe, unsigned int *mbox_data, int count)*

Returns next *count* entries from *spe* outbox, returning:

-1: Error

0: Nothing read

>0: Number of entries read

^b (void) *spu_write_out_mbox* (uint32_t *data*)

Writes *data* to local spe outbox

^c int *spe_out_mbox_status* (spe_context_ptr_t *spe*)

Returns number of entries in *spe* outbox ready for the PPE to read

-1: Error

0: Nothing in outbox

>0: Number of entries ready in outbox

^d uint32_t *spu_stat_out_mbox*(void)

Returns number of free entries in local SPE outbox

-1: Error

0: No free entries

>0: Number of free outbox entries

^e int *spe_out_intr_mbox_read*(
 spe_context_ptr_t *spe*,
 unsigned int **mbox_data*,
 int *count*,
 unsigned int *behavior*)

Reads *count* entries from *spe* outbox and writes them to *mbox_data* , returning:

-1: Error

0: Nothing read

>0: Number of entries read²

^f The blocking behavior of ^e is defined by the *behavior* parameter:

SPE_MBOX_ALL_BLOCKING

Blocks until *count* messages has been read

SPE_MBOX_ANY_BLOCKING

Blocks until at least one of *count* messages has been read

SPE_MBOX_ANY_NONBLOCKING

Try to read as many of *count* messages as possible without blocking

^g (void) *spu_write_out_intr_mbox* (uint32_t *data*)

Writes *data* to local spe interrupt outbox

² It should be noted that the return code with behavior SPE_MBOX_ALL_BLOCKING is `_NOT_` *count* , as expected but 1. This is either a bug in the Cell SDK 2.1 or due to bad documentation and might be changed in future versions of the Cell SDK

^h `int spe_out_intr_mbox_status (spe_context_ptr_t spe)`

Returns number of entries in `spe` interrupt outbox ready for the PPE to read

-1: Error

0: Nothing in interrupt outbox

>0: Number of entries ready in interrupt outbox

ⁱ `uint32_t spu_stat_out_intr_mbox(void)`

Returns number of free entries in local SPE interrupt outbox

-1: Error

0: No free entries

>0: Number of free entries in interrupt outbox

An example of an SPE writing to it's outbox is shown below:

```

1 #include <spu_mfcio.h>
2 #include <stdio.h>
3
4 int main(unsigned long long speid)
5 {
6     unsigned int message;
7
8     message = 100;
9     spu_write_out_mbox(message);
10
11     return 0;
12 }
```

The PPE code for reading the written messages is shown next.

```

1 #include <libspe2.h>
2 #include <pthread.h>
3 #include <sys/time.h>
4
5 /* This is the pointer to the SPE code to be executed */
6 extern spe_program_handle_t mailbox_spehandle;
7
8 /* This is the handle used by the PPE to control the SPE */
9 spe_context_ptr_t speid;
10
11 /* The initial value of the SPE's instruction pointer */
12 unsigned int entry = SPE_DEFAULT_ENTRY;
13
14 void *start_spe_thread(void *arg) {
15     /* Give control of this thread to the SPE,
16      * this block until the SPE returns
17      * and control is given back to the PPE */
18     spe_context_run(speid, &entry, 0, NULL, NULL, NULL);
19
20     pthread_exit(NULL);
21 }
```

```

22 |
23 | int main()
24 | {
25 |     unsigned int message;
26 |     int status;
27 |
28 |     pthread_t pthread;
29 |
30 |     /* Create context */
31 |     speid = spe_context_create(0, NULL);
32 |
33 |     /* Load program into context */
34 |     spe_program_load(speid , &mailbox_spehandle);
35 |
36 |     /* Create POSIX thread which starts SPE execution */
37 |     pthread_create(&pthread , NULL,
38 |                  &start_spe_thread ,
39 |                  NULL);
40 |
41 |     while((status = spe_out_mbox_status(speid)) == 0 ) {
42 |         printf("Waiting for spe_outbox_message.\n");
43 |         sleep(1);
44 |     }
45 |     status = spe_out_mbox_read(speid , &message , 1);
46 |     printf("Outbox_read_status: %i\n" , status);
47 |     printf("Outbox_message_read: %u\n" , message);
48 |
49 |     /* Wait for SPE thread to terminate */
50 |     pthread_join(pthread , NULL);
51 |
52 |     return (0);
53 | }

```

By using the interrupt mailbox the same functionality can be archived without busy waiting for the arrival of the message line 9 of the SPE code from above needs to be changed to code is shown below:

```

9 |     spu_write_out_intr_mbox(message);

```

And line 41 to 45 of the PPE code from the example above needs to be changed to:

```

41 | status = spe_out_intr_mbox_read(speid , &message , 1,
42 |                               SPE_MBOX_ALL_BLOCKING);

```

The read/write functions for the outboxes and their parameters works the same way as the read/write functions for the SPE's inbox section 4.2.1 page 14.

5 DMA transfers

Mailboxes and signals are handy for sending a few short messages between the PPE and the SPE's. If larger amount of data needs to be transferred, to and from the SPE's local store and main memory, DMA transfers should be used. DMA transfers are always initiated by the SPE who wishes to send/receive data to/from main memory and are done asynchronously by the MFC figure 3 page 3. This means the SPE can perform computations while transferring data between the local store and main memory which is one of the features that makes the Cell so powerful. As it's always the SPE who initiates the DMA transfer, the main memory address where the data resides, needs to be known by the SPE, this address can be sent either as a mailbox message or as an argument to the main function of the SPE, when it's started from the PPE.

In this example we will show how to obtain a main memory address from the PPE through the main function of the SPE, and initiate a DMA transfer from that address. The PPE code for this is shown below:

```

1 #include <libspe2.h>
2 #include <pthread.h>
3 #include <malloc_align.h>
4
5 #define DATA_BUFFER_SIZE 32
6
7 /* This is the pointer to the SPE code to be executed */
8 extern spe_program_handle_t dma_spehandle;
9
10 /* This is the handle used by the PPE to control the SPE */
11 spe_context_ptr_t speid;
12
13 /* The initial value of the SPE's instruction pointer */
14 unsigned int entry = SPEDEFAULTENTRY;
15
16 void *start_spe_thread(void *arg) {
17     unsigned int i;
18     unsigned int *data;
19
20     /* Malloc align 2^7=128 byte aligned
21      * as SPE has memory lines of 128 byte */
22     data = (unsigned int *)_malloc_align(DATA_BUFFER_SIZE*sizeof(int),
23                                         7);
24     data[0] = data[1] = 1;
25     for (i=2; i<DATA_BUFFER_SIZE; i++) {
26         data[i] = data[i-1] + data[i-2];
27     }
28
29     /* Give control of this thread to the SPE,
30      * this block until the SPE returns
31      * and control is given back to the PPE */
32     spe_context_run(speid, &entry, 0, data, NULL, NULL);
33
34     pthread_exit(NULL);
35 }

```

```

36 int main()
37 {
38     pthread_t pthread;
39
40     /* Create context */
41     speid = spe_context_create(0, NULL);
42
43     /* Load program into context */
44     spe_program_load(speid, &dma_spehandle);
45
46     /* Create POSIX thread which starts SPE execution */
47     pthread_create (&pthread, NULL, &start_spe_thread, NULL);
48
49     /* Wait for SPE-thread to complete execution.
50     */
51     if (pthread_join (pthread, NULL)) {
52         perror("Failed pthread_join");
53         exit (1);
54     }
55
56     return (0);
57 }

```

The interesting parts are the lines 22 to 32, the rest doesn't differ from the previous examples. The data to be read by the SPE is allocated in line 22 by a special malloc (`_malloc_align`) function provided by the Cell SDK, this assures the allocated data to be 128 byte aligned which is necessary as the SPE's local stores operates with 128 byte lines. When memory is allocated it's initialized with the Fibonacci numbers, line 24-27. The SPE thread is started in line 32 where the the main memory address of the data is given as an argument. The actual DMA transfer is done by the SPE, which is shown next:

```

1 #include <spu_mfcio.h>
2 #include <stdio.h>
3
4 #define DATA_BUFFER_SIZE 32
5
6 int main(unsigned long long speid, unsigned long long mainmem_addr)
7 {
8     unsigned int i;
9     int tagid = 31;
10
11     // It's important that local data are 128 byte aligned
12     unsigned int local_data[DATA_BUFFER_SIZE]
13         __attribute__((aligned (128)));
14
15     printf("SPE_DMA_transfer_from_addr:_0x%lX_to_addr:_0x%X\n",\
16         mainmem_addr, (unsigned int) &local_data[0]);

```

```

17  /* Here is the actual DMA call */
18  * First parameter:
19  * Is the address in local store to place the data
20  *
21  * Second parameter:
22  * Holds the main memory address
23  *
24  * Third parameter:
25  * Holds the number of bytes to DMA
26  *
27  * Fourth parameter:
28  * Identifies a "tag" to associate with this DMA
29  * (this should be a number between 0 and 31, inclusive)
30  *
31  * The last two parameters are only useful if you have
32  * implemented your own cache replacement management policy.
33  * Otherwise set them to 0.*/
34  mfc_get(&local_data[0], mainmem_addr, sizeof(local_data), tagid,
35         0, 0);
36
37  /* Now, we set the "tag bit" into the correct channel
38  * on the hardware this is always 1 left-shifted by the tag
39  * specified with the DMA for whose completion
40  * you wish to wait. */
41  mfc_write_tag_mask(1<<tagid);
42
43  /* Now, issue the read and wait to guarantee DMA completion
44  * before we continue. */
45  mfc_read_tag_status_all();
46
47  /* Verify that the data array
48  * contains a valid Fibonacci sequence. */
49
50  for (i=2; i<DATA_BUFFER_SIZE; i++) {
51      printf("data[%u]: %u\n", i, local_data[i]);
52      if (local_data[i] != local_data[i-1] + local_data[i-2]) {
53          printf("ERROR: _Fibonacci_sequence_error_at_entry_%d.", i);
54          printf(" _Expected_%d, _Got_%d\n",
55                 local_data[i-1] + local_data[i-2], local_data[i]);
56          return (1);
57      }
58  }
59
60  printf("SPE_finished\n");
61
62  return 0;
63 }

```

The local store memory used to store the data received by the DMA is allocated in line 12, it's vital for a successful transfer that the local store data is 128 byte aligned. The "mfc_get" line 34 initiates the transfer, and requires the local store address, the main memory address, the amount of bytes to be transfered and a tagid, to identify the DMA transfer. The last to parameters can be used for implementing a cache scheme and is not covered in this tutorial. To achieve maximum bandwidth usage of the EIB bus, the size of the data transfered in each DMA transaction should be a multiplum of 128 bytes, and both source and destination addresses should be 128 bit aligned. The "mfc_read_tag_status_all" call line 45 blocks until DMA transfers specified with call to "mfc_write_tag_mask" line 41 has completed. The data received at the local store is verified at line 50 to 58.

6 Error handling and debugging

The Cell architecture is quite complex as a high level of parallelism is needed to make it perform well. As with all parallel systems it's hard to debug your Cell applications when more than 1 SPE is used. The Cell SDK framework functions used in this tutorial all supports "errno.h" to provide error information if anything goes wrong, these has been left out until now for simplicity. The example "All SPE's started by the PPE" section 3.4 page 8 with error handling is shown below, only the PPE code is shown, as the SPE code is the same.

```

1 #include <stdlib.h>
2 #include <libspe2.h>
3 #include <pthread.h>
4 #include <errno.h>
5 #include <string.h>
6
7 #define SPE_THREADS 8
8
9 /* This is the pointer to the SPE code to be executed */
10 extern spe_program_handle_t hello_spehandle;
11
12 /* This is the handle used by the PPE to control the SPE */
13 spe_context_ptr_t speid [SPE_THREADS];
14
15 /* This is used by the PPE to check how SPE terminated */
16 spe_stop_info_t stop_info [SPE_THREADS];
17
18 /* The initial value of the SPE's instruction pointer */
19 unsigned int entry [SPE_THREADS];
20
21 void *start_spe_thread(void *arg) {
22     int i;
23     i = ((int) arg);
24
25     /* Set the initial value of the SPE's instruction pointer */
26     entry [i] = SPE_DEFAULT_ENTRY;
27
28     /* Give control of this thread to the SPE,
29      * this block until the SPE returns
30      * and control is given back to the PPE */
31     if (spe_context_run(speid [i], &entry [i], 0, NULL, NULL,
32                       &stop_info [i]) < 0) {
33         fprintf (stderr, "PPE: _thread_nr: %i -> _FAILED_", i);
34         fprintf (stderr, "spe_context_run (errno=%d _strerror=%s)\n",
35                errno, % strerror(errno));
36         exit(1);
37     }
38
39     pthread_exit(NULL);
40 }

```

```

42 int main()
43 {
44     int i;
45     pthread_t pthreads[SPE_THREADS];
46
47     /* Create SPE threads to execute 'hello world'
48     */
49     for(i=0; i<SPE_THREADS; i++) {
50         /* create SPE context */
51         if ((speid[i] = spe_context_create(0, NULL)) == NULL) {
52             fprintf (stderr, "PPE: %i: _FAILED_", i);
53             fprintf (stderr, "spe_context_create(errno=%d_strerror=%s)\n",
54                     errno, strerror(errno));
55             exit (1);
56         }
57
58         /* load the SPE program into the SPE context */
59         if (spe_program_load(speid[i], &hello_spehandle) != 0) {
60             fprintf (stderr, "PPE: %i: _FAILED_", i);
61             fprintf (stderr, "spe_program_load(errno=%d_strerror=%s)\n",
62                     errno, strerror(errno));
63             exit (1);
64         }
65
66         /* Create a POSIX thread for each SPE context */
67         if (pthread_create (&pthreads[i], NULL, &start_spe_thread,
68                             (void *)i)) {
69             perror ("PPE: _Failed_creating_spe_thread\n");
70             exit (1);
71         }
72     }

```

Continued next page.

```

73  /* Wait for SPU-thread to complete execution. */
74  for (i=0; i<SPE_THREADS; i++) {
75      if (pthread_join(pthread_s[i], NULL)) {
76          perror("PPE: Failed pthread_join\n");
77      }
78
79      /* Free SPE context info */
80      if (spe_context_destroy(speid[i]) != 0) {
81          fprintf(stderr, "PPE: %i: FAILED_spe_context_destroy(", i);
82          fprintf(stderr, "errno=%d_strerror=%s)\n",
83                  errno, strerror(errno));
84      }
85
86      /* Check the SPE status */
87      if (stop_info[i].stop_reason != SPE_EXIT) {
88          fprintf(stderr, "PPE: %i: FAILED_");
89          fprintf(stderr, "SPE_abnormally_terminated\n", i);
90      }
91      else if (stop_info[i].result.spe_exit_code != 0) {
92          fprintf(stderr, "PPE: %i: SPE_returned_a_exit_code: %i\n",
93                  i, stop_info[i].result.spe_exit_code);
94      }
95  }
96  return (0);
97 }

```

It's seen that all calls to initiate and start the SPE's used in the previous examples has been wrapped within code for checking whether the call went through or not. Furthermore deallocation of the SPE context has been added in line 80. Last but not least information about how execution went on the SPE's has been added, this is seen in line 17, line 33 and the lines 87 to line 94.

Beside error handling the Cell SDK provides a debugger (GDB) with SPE extensions, this is a very useful tool combined with the Cell simulator. As with all debugging the most used debugging tool for the Cell BE is "printf", it should be noted though that all "printf" calls used in SPE code, results in a DMA transfer to main memory of the string to display. This results in EIB bandwidth usage and a possible interference with other DMA calls. Therefore it's not advisable to use printf's from SPE's i production code, instead one should send a mailbox message or DMA the data to main memory, and let the PPE perform the actual displaying using "printf".

7 The End

This tutorial shows the initial steps towards programming the Cell BE processor, used in both Cell Blades and the Playstation 3. The Cell BE processor is a highly parallelized processor which makes it complex to program. We have not covered all levels of parallelization, but only the basic level, which is starting the SPE's from the PPE and performing communicating with them. Levels of parallelism not covered are:

- Single Instruction Multiple Data (SIMD)
Operating on multiple data (vectors) in one instruction
- Instruction parallelism
Performing two (one at each pipeline) instructions in one clock cycle
- Double/Multi buffering
Prefetching data from main memory while processing previous fetched data

These three methods complies with the SPE only. Other topics mentioned but not covered are the Cell SDK tools such as the simulator and the debugger (GDB). For more information on programming the Cell BE processor one is encouraged to look at the following:

- PPU libspe2 reference manual
<http://www.diku.dk/~rehr/cell/docs/libspe-v2.0.pdf>
- SPU C/C++ language extensions
http://www.diku.dk/~rehr/cell/docs/SPU_language_extensions_2.1.pdf
- Programming manual
<http://www.redbooks.ibm.com/abstracts/sg247575.html>

The examples used in this tutorial can be found here:

- http://www.diku.dk/~rehr/cell/files/Cell_Tutorial_Examples.tar.bz2