Lesson 6
PPU and SPU Threads

Playstation 3 Development

Sam Serrels and Benjamin Kenwright[1]

Abstract
This article explains how to initiate, control, and communicate between threads on both the PPU and SPU. Multi-threading is the key to performance on the Playstaiton 3 and this article will get you started on the path to writing faster and more optimised applications.

Keywords
Sony, PS3, PlayStation, Setup, Windows, Target Manager, ELF, PPU, SPU, Programming, ProDG, Visual Studio, Memory alignment

[1] *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

Contents

1. Introduction

About the Edinburgh Napier University Game Technology Playstation 3 Development Lessons Edinburgh Napier University Game Technology Lab is one of the leading game teaching and research groups in the UK - offering students cutting edge facilities that include Sony's commercial development kits. Furthermore, within the Edinburgh Napier Game Technology group are experienced developers to assist those students aspiring to releasing their own games for PlayStation. Students have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [1].

Threads With the multi-parallel architecture of the Playstation 3, organising the execution of your programs into threads is the only way to make use of the full potential of the system. The threading libraries are comparatively straightforward to use when compared to to other subsystems and libraries available on the PS3.

SPUs The four SPUs avalaible to the user have two modes of operation, Threaded and Raw. A raw SPU has no time/task management processing, you are given full access to the hardware and left to do whatever you want with it. Threaded SPUs still run specialised spu code, but the execution is managed by the operating system, the factors of what spu is running what code and when is abstracted away from the user. A thread or groups of threads are told to run by the user and the operating system makes sure that they do. There are multiple of advantages and disadvantages to both modes of operation but that is beyond the scope of this tutorial.

SPURS A middle of the road method of running code on SPUs is to use the SPURS(SPU Runtime System) library. In a SPURS environment, SPU threads are managed by SPUs. For this reason, thread switching is more efficient than under PPU management and requires no PPU resources. Using SPURS also makes it easier to synchronize threads and adjust the load balance on multiple SPUs. Essentially it gives the benefits of using a managed thread library on the SPUs without the need for the PPU to watch over it, it leaves the SPUs to manage their own threads.

2. PPU threads

Listing 1. Multithreading on the PPU

```
#include <stdio.h>    // Printf
#include <stdlib.h>   // exit function
#include <sys/timer.h>  // Sleeps
#include <sys/ppu_thread.h> // PPU threads

#define THREADS 4
```

```
 7
 8 void ppu_thread_function ( uint64_t arg ) {
 9   printf ("PPU thread[%i] processing\n",arg);
10   sys_timer_sleep(arg+2);
11   printf ("PPU thread[%i] Finished\n",arg);
12   sys_ppu_thread_exit (0);
13 }
14
15 int main(void)
16 {
17   printf ("PPU MAIN: Creating threads\n");
18   sys_ppu_thread_t threads[THREADS];
19   int return_val;
20
21   for( int i = 0; i < THREADS; i ++ )
22     {
23     return_val = sys_ppu_thread_create (
24       &threads[i],
25       ppu_thread_function, i,
26       200, 4096,
27       SYS_PPU_THREAD_CREATE_JOINABLE,
28       (char*)"simple thread");
29
30     if ( return_val != CELL_OK ) {
31       printf("thread creation failed:%i\n", return_val);
32       exit (return_val);
33     }
34     }
35
36   //Wait for the threads to terminate
37   for( int i = 0; i < THREADS; i ++ )
38     {
39     //Processing will halt here until thread finishes
40     sys_ppu_thread_join(threads[i],0);
41     }
42
43   printf ("PPU MAIN: Exiting ...\n");
44   return 0;
45 }
```

The line of most interest is sys_ppu_thread_create(), on line 26. Here is a breakdown of the parameters it takes:

Thread id
(OUT) Pointer to storage for the PPU thread ID

Entry point
A function pointer to the starting point of the thread code.

An argument
A generic argument to pass to the thread function.

Priority
Priority of the thread in range 0 (highest) from 3071.

Stacksize
Stack size in bytes

Flags
PPU thread flags

Threadname
The name of this thread (used by debugger)

Thread Flags The accepted values for the flags are:
0 (non-joinable non-interrupt),
SYS_PPU_THREAD_CREATE_JOINABLE(joinable),
SYS_PPU_THREAD_CREATE_INTERRUPT (interruptible).
A joinable thread can have another thread ”join” onto the end of it. So if thread A joins onto thread B, thread A will halt execution, wait for thread B to finish and then continue. This can be used to detect when a thread is completed, as the main application code is actually a thread in it's own right. We see this in use on line 40.

3. Event Queues

Data transfer and Communication Using threads to do useful processing, more than just printing ”Hello world”, requires two things: a means of communication with other threads and a means of obtaining data to work on. This tutorial will cover just the first aspect, thread communication. Data transfer will be covered in a future tutorial. For PPU threads data transfer can be just a simple matter of sending pointers around, but for SPUs it's a much more involved task as data has to be transferred form main memory to local storage.

Communication While ”communicating” is still actually a form of data transfer, there are systems designed specifically for easily sending very small amount of data to communicate the status of threads and to send simple control data. This communication data is normally associated with the term ”Flags”. Normally data is sent to a thread/device to set a binary flag that will be read and acted upon in some manor. While we can do this with the PS3 (and we *will* use it in the later data transfer tutorial) we can make use of a higher level system to send more meaningful data, called Events.

Events ”The event mechanism is a one-way synchronous communication mechanism that realizes asynchronous communication between threads.” Events can be fired from multiple sources. You can send your own events manually from PPU and SPU thread functions (user events). System events are also fired from SPU threads in error conditions, and the printf function on SPUs is implemented in such that it has it's own event category. The data that can be sent in an event various on what type of event it is. User events can always send atleast one piece of 24-bit data and one piece 32-bit data. This is covered in more detail after some additional information about events are explained.

Event Queues We have established that events can be sent from a multitude of sources, but where do they go? Assuming they have been directed to do so, they get stored in an event Queue. An event queue is exaclty what it sounds like, a queue that can receive events simultaneously form multiple sources. The most common scenario is to create a queue in main memory from the main application thread. A separate ”Queue Manager” PPU thread is then created to read through the queue and process the events one by one. When additional ”worker threads” are created to ”do work” they are linked to an event queue to send events to (Via a certain port). Both raw and threaded spus can send events to an event queue, and SPUS can host their own queue to *receive* events from the PPU or other SPUs.

Event ports A mentioned above, events are sent to a queue *via a certain port*, this is a neat system of separating events for different purposes. A thread can have multiple ports, and a queue can receive data from multiple ports, but one port can

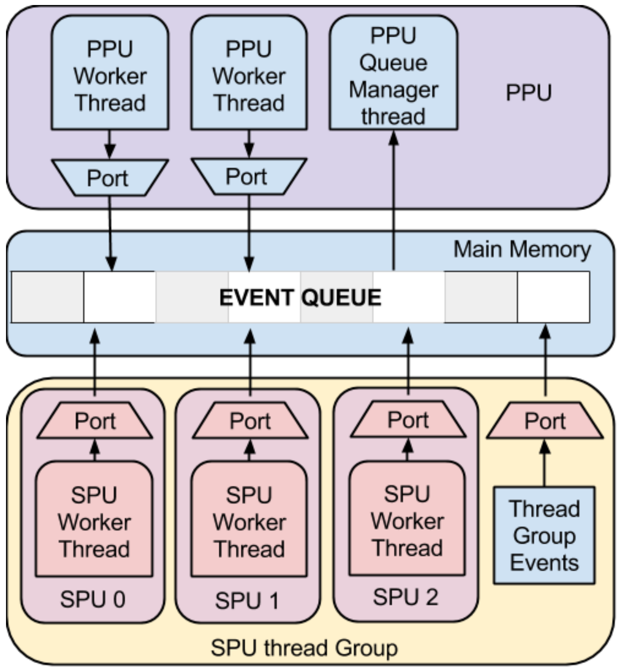


Figure 1. Event Queue - Visual depiction of the interworking of the SPU and PPU.

only connect to one queue, I.e if you want to send events to two separate queues from one thread, you would need two ports.

PPU Event ports Setting up event ports on a PPU thread is super easy. Here is the function to do it:

```
sys_event_port_create(
   sys_event_port_t *event_port_id,
   int port_type,
   uint64_t name
);
```

***event_port_id** is the location to store the created port ID, this is used when sending events.
port_type is always set to: "SYS_EVENT_PORT_LOCAL"
name can be any unique 64-bit int that you want, this is passed to the event manager thread. If you don't need to specify a name, you must use: "SYS_EVENT_PORT_NO_NAME"

You must now link the port to a queue, and then you are ready to send events. This will all be shown in code later, but just to how show easy the system is, this is how we send an event:

```
sys_event_port_create(&port, SYS_EVENT_PORT_LOCAL, ↩
      SYS_EVENT_PORT_NO_NAME);
sys_event_port_connect_local(port, queue);
sys_event_port_send(port, data1, data2, data3);
```

3.1 Event Data

Event Data Unfortunately, the data that comes from different source types have a different format of data. An event is always of the type "sys_event_t", a struct with 4 integers, called: source, data1, data2 and data3. How these events are sent and received for each type of event is detailed below.

PPU user event data A PPU user event is sent with sys_event_port_send(port, data1, data2, data2).
The "sys_event_t" received by the event manager has the following data:

Source
Source port name
data1
First data
data2
Second data
data3
Third data

SPU user event data An SPU user event is sent with sys_spu_thread_send_event(port, data0, data1).
Notice how we can only send 2 sets of data, and also note that only the lower 24 bits of data0 are sent.
The "sys_event_t" received by the event manager has the following data:

Source
Always "SYS_SPU_THREAD_EVENT_USER_KEY"
data1
SPU thread ID
data2
(upper 32 bits) – SPU port number
(lower 32 bits) – (only the lower 24 bits) of the second argument of sys_spu_thread_send_event()
data3
The third argument of sys_spu_thread_send_event()

SPU Thread Group Run Event The "sys_event_t" received by the event manager has the following data:

Source
"SYS_SPU_THREAD_GROUP_EVENT_RUN_KEY"
data1
SPU thread group ID
data2 and data3
N/A

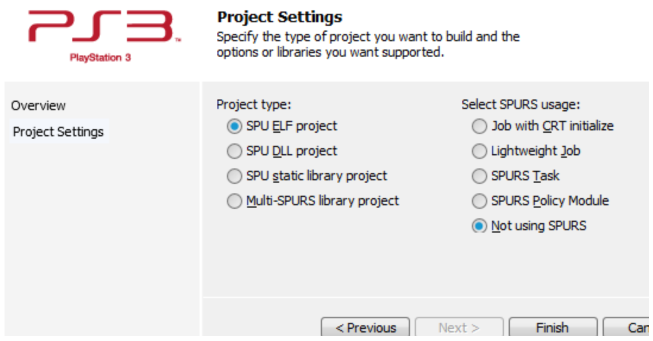
Event Data Conclusion As you can probably guess, the Event manager is going to have to have a large nest of switches to correctly manage all the possible types of incoming data. We will now look at the code for implementing an event system with only PPU threads.

3.2 PPU threading with events, code

Listing 2. PPU threading, with events

```
//#define DBG_HALT { __asm__ volatile( "trap" ); }
#define DBG_HALT { exit (1); }
// call DBG_HALT on assert fail
```

```c
#define DBG_ASSERT(exp) { if ( !(exp) ) {DBG_HALT;} }
// Prints the suplied string on assert fail, then call DBG_HALT
#define DBG_ASSERT_MSG( exp, smsg ) { if ( !(exp) ) {puts (smsg); DBG_HALT;} }
// Calls the suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC( exp, func) { if ( !(exp) ) {func; DBG_HALT;} }

#include <stdio.h>    // printf
#include <stdlib.h>   // exit function
#include <sys/timer.h>  // Sleeps
#include <sys/ppu_thread.h> // PPU threads
#include <sys/event.h>    // process events

#define PPU_STACK_SIZE 4096
#define PPU_PRIORITY 200

static sys_event_queue_t event_queue;
void ppu_queue_manager_thread_entry(uint64_t arg);
void ppu_woker_thread_entry(uint64_t arg);

//Priority and stack size of the primary PPU thread of the game process
SYS_PROCESS_PARAM (1001, 0x10000);

int main(void)
{
  sys_ppu_thread_t qmgr_thread;
  sys_ppu_thread_t worker_thread;
  sys_event_queue_attribute_t queue_attr;
  int err;

  printf ("\n\nPPU MAIN: Hello world !\n");

  // Create and init an Event queue ------------------------------------

  sys_event_queue_attribute_initialize ( queue_attr );
  //The maxmimum events a queue can hold is 127
  err = sys_event_queue_create (&event_queue, &queue_attr, SYS_EVENT_PORT_LOCAL, 127);
  DBG_ASSERT_FUNC((err == CELL_OK), printf("PPU MAIN: sys_event_queue_create failed %i\n", err) );

  // Create a PPU thread to watch the event queue ------------------------------------

  err = sys_ppu_thread_create (&qmgr_thread, ppu_queue_manager_thread_entry, 0, PPU_PRIORITY, PPU_STACK_SIZE, SYS_PPU_THREAD_CREATE_JOINABLE, (char*)"queue manager");
  DBG_ASSERT_FUNC((err==CELL_OK),printf("PPU MAIN: Queue manager thread creation failed: %i\n",err));

  // Create a worker thread ------------------------------------

  err = sys_ppu_thread_create (&worker_thread, ppu_woker_thread_entry, 0, PPU_PRIORITY, PPU_STACK_SIZE, SYS_PPU_THREAD_CREATE_JOINABLE, (char*)"simple worker thread");
  DBG_ASSERT_FUNC((err == CELL_OK), printf("PPU MAIN: PPU worker thread creation failed : %i\n", err) );

  //Wait for the worker thread to terminate ------------------------------------

  //Processing will halt here until the specifed thread finishes
  sys_ppu_thread_join(worker_thread,0);

  printf ("PPU MAIN: worker finished\n");

  //destroy the Queue
  sys_event_queue_destroy (event_queue, SYS_EVENT_QUEUE_DESTROY_FORCE );
  //Using the Force mode, causes sys_event_queue_receive() in the manager thread to return ECANCELED.

  //Let's wait for the manager thread to finish, just to be polite.
  sys_ppu_thread_join(qmgr_thread,0);

  printf ("PPU MAIN: Exiting ...\n");
  return 0;
}



//This function waits for incoming events from the Event Queue attached to our SPU thread, and outputs them as text strings.
void ppu_queue_manager_thread_entry ( uint64_t arg ) {
  int return_val;
  sys_event_t event;
  sys_spu_thread_t spu;

  //keep looping, and checking for new events
  while (1) {
    return_val = sys_event_queue_receive (event_queue, &event, SYS_NO_TIMEOUT);
    if ( return_val != CELL_OK ) {
      if( return_val == ECANCELED ) {
        printf ("PPU:Q MGR: Event Queue destroyed ! Exiting ... \n");
      } else {
        printf ("PPU:Q MGR: Event Queue receive failed : %i\n", return_val);
      }
      break ;
    }

    printf ("PPU:Q MGR: New Event!, Port %i, Data: %i, %i, %i\n",event.source,event.data1,event.data2,event.data3);
  }

  printf ("PPU:Q MGR: Event Queue receive failed : %i\n", return_val);
  sys_ppu_thread_exit (0);
}


void ppu_woker_thread_entry ( uint64_t arg ) {
  int return_val;
  printf ("Worker thread is alive\n");

  //create an Event Port
  sys_event_port_t port;
  return_val = sys_event_port_create(&port, SYS_EVENT_PORT_LOCAL, SYS_EVENT_PORT_NO_NAME);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("PPU worker : sys_event_port_create failed %x\n", return_val ));

  //link port to Main event queue (which is global and static so we can use it in this scope)
  return_val = sys_event_port_connect_local(port, event_queue);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU worker: sys_event_port_connect_local failed%x\n", return_val ));

  //Let's send an event!
  //we can send 3x 64-bit data (e.g 3 ints), wow such data!
  int a = 32;
  int b = 1337;
  int c = 999;
  printf ("PPU worker: Sending Event, Data: %i, %i, %i\n",a,b,c);
  return_val = sys_event_port_send(port, a, b, c);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("PPU worker : sys_event_port_send failed %x\n", return_val ));
```



Project Settings
Specify the type of project you want to build and the options or libraries you want supported.

Project type: [✓ SPU ELF project] SPU DLL project, SPU static library project, Multi-SPURS library project

Select SPURS usage: Job with CRT initialize, Lightweight Job, SPURS Task, SPURS Policy Module, [✓ Not using SPURS]

Figure 3. spuprojectsetup -

```

//sleep for 5 seconds
sys_timer_sleep(5);

//send another event
a = 64;
b = 0;
c = 9001;
printf ("PPU worker: Sending Event, Data: %i, %i, %i\n",a,b,c);
return_val = sys_event_port_send(port, a, b, c);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("PPU worker : sys_event_port_send failed %x\n", return_val ));

//sleep for 4 seconds and end
sys_timer_sleep(4);

//disconnect the port
return_val = sys_event_port_disconnect(port);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("PPU worker : sys_event_port_disconnect %x\n", return_val ));

printf ("PPU worker : Shutting down\n");
//close thread
sys_ppu_thread_exit (0);
}
```

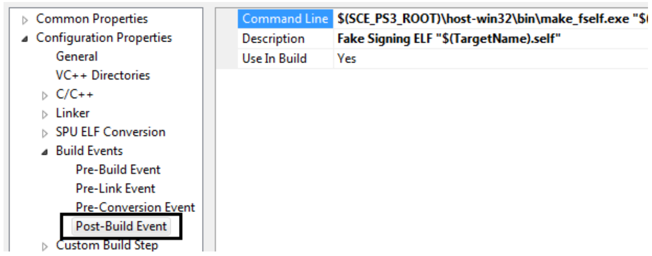
4. SPU threads

SPU compiling As the Spus use a separate instruction set to the the PPU, code must be compiled separately for the SPUs. This ste p is made easy by the project wizard intergrated into visual studio, simply create a new project, and select "PS3 SPU project" see Figure 3.

SPU Elfs A few extra steps are needed to get the compiled output ready for use. Firstly, the compiled .elf file need to be digitally signed into a .self file. The application to do this is make_fself.exe, located in /host-win32/bin/. We can automate this easily by adding a post build process to the project. see Figure 4 and use the following code as the command:

```
$(SCE_PS3_ROOT)\host−win32\bin\make_fself.exe "$(OutDir)$(TargetName)$(TargetExt)" "$(OutDir)$(TargetName).self"
```

SPU .self location The Spu.self file will be loaded by the main ppu program, and as with previous examples we can load files that are located on the developer pc as if they were stored locally. However this does mean that the self file needs to be in the correct folder, "The file serving directory" set by the target manager, which is accessed with code as "SYS_APP_HOME". The quickest solution is to find where this directory is (usually the same directory as the project you are debugging/running), and copy the compiled .self file there. A better solution would be to change the output directory in the SPU project build configuration.



Command Line	$(SCE_PS3_ROOT)\host-win32\bin\make_fself.exe "$(
Description	Fake Signing ELF "$(TargetName).self"
Use In Build	Yes

Figure 4. postbuildevent -

Thread Groups A SPU thread group is a collection of SPU threads. All SPU threads in an SPU thread group are always **concurrently executed in parallel** on SPUs. Therefore, the maximum number of SPU threads that an SPU thread group can consist of is **equal to the number of SPUs available for SPU threads**. You can specify a priority to every thread group, and the kernel assigns the groups to SPUs in priority order.

Thread Groups Events Events that are sent by SPU thread groups are called SPU thread group events. There are two types of these events: The run event and The exception event. The Run event is triggered when the SPU thread group is first loaded and executed on SPUs. When this event is triggered, the loaded self file in main memory could be deleted to free memory if it is not going to be used again.

PrintF on SPUs As the SPU doesn't have native access to the standard out stream, it has to route all printf() commands through the PPU. This is done using the events system, through a reserved port (0x1), therefore this port should not be used by your code unless you are never making use of print commands. It is possible to receive the print events through the standard event manger thread, however the SPU expects certain data to be sent in return to tell it that the print was successful. Unfortunately this protocol is undocumented in the SDK, fortunately there is a set of helper function that do all this work for us. Behind the scenes they do the same work we would normally do to setup/link/process an event queue but only for print statements and it means we don't have to any hard work. You will see these commands in use on the following set of code.

4.1 SPU Threading code

Listing 3. SPU threading, SPU code

```
#include <spu_printf.h>

int main(void)
```

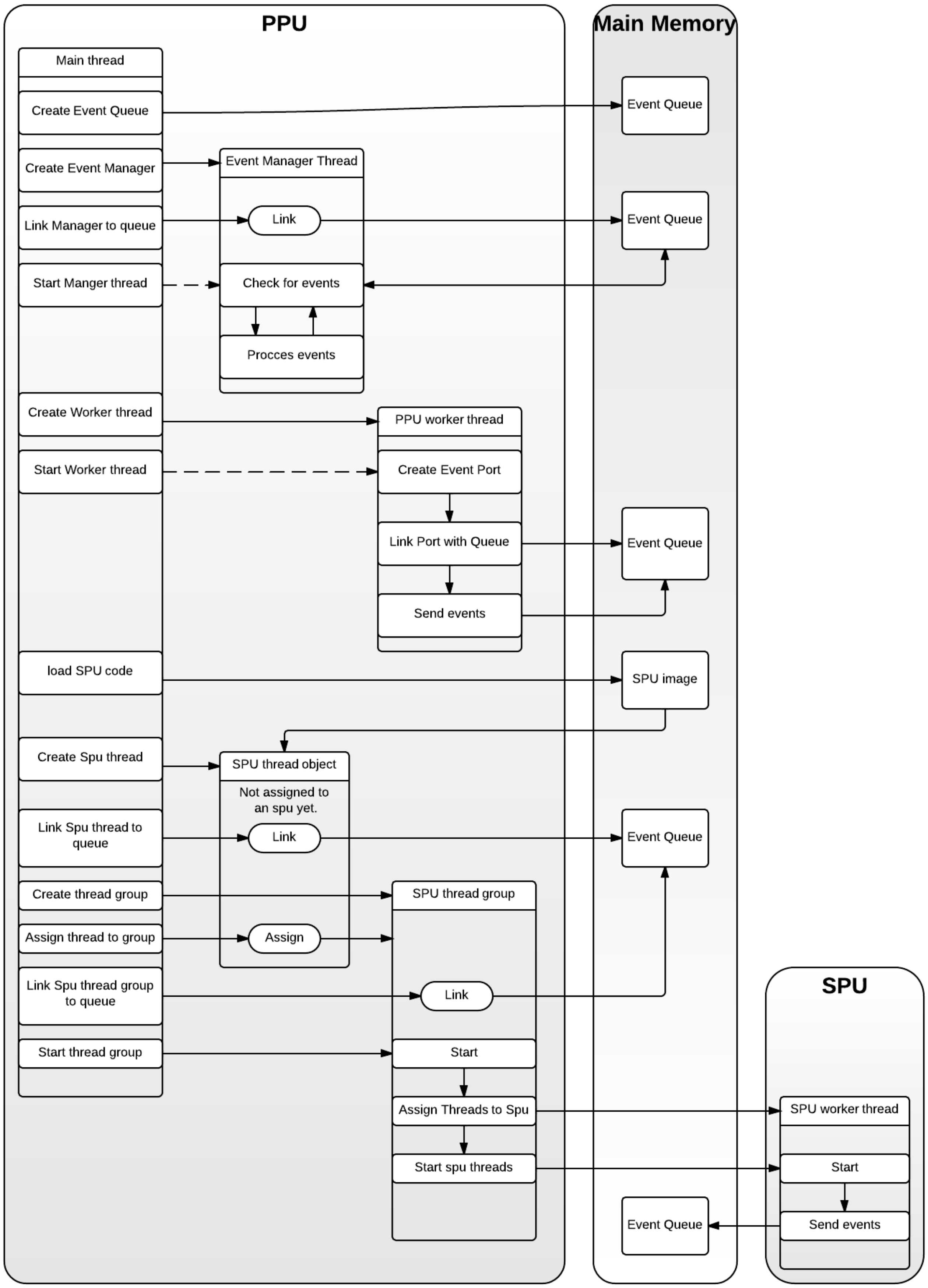


Figure 2. **Event Queue Setup, Rough Sequence of Operations -**

```c
{
 spu_printf("SPU: Hello, world!\n");
 return 0;
}
```

Listing 4. SPU threading, PPU code

```c
/*
 Tutorial 1 - 7 - 3
 SPU THREADS - PPU CODE
 Creates an SPU thread, waits for it to finish, then exits.

 www.napier.ac.uk/games
 Sam Serrels and Benjamin Kenwright (b.kenwright@napier.ac.uk)
*/
#include <stdio.h>    // printf
#include <stdlib.h>   // exit function
#include <spu_printf.h>   // SPU printf functions
#include <sys/spu_initialize.h> // A initialise SPUs
#include <sys/spu_utility.h>  // SPU images
#include <sys/paths.h>    // SYS_APP_HOME
#include <sys/spu_thread.h> // make SPU threads
#include <sys/event.h>    // process events

//#define DBG_HALT { __asm__ volatile( "trap" ); }
#define DBG_HALT { exit (1); }
// call DBG_HALT on assert fail
#define DBG_ASSERT(exp) { if ( !(exp) ) {DBG_HALT;} }
// Prints the suplied string on assert fail, then call DBG_HALT
#define DBG_ASSERT_MSG( exp, smsg ) { if ( !(exp) ) {puts (smsg); DBG_HALT;} }
// Calls the suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC( exp, func) { if ( !(exp) ) {func; DBG_HALT;} }

#define PPU_STACK_SIZE 4096
#define PPU_PRIORITY 200
#define MAX_PHYSICAL_SPU 1
#define MAX_RAW_SPU 0

// ~~~@@~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~@@~~~
#define SPU_SELF "/ps3_tut_1_7_3_spu_threads_SPU.self"
// ~~~@@~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~@@~~~

#define SPU_PROG (SYS_APP_HOME SPU_SELF)

//Priority and stack size of the primary PPU thread of the game process
SYS_PROCESS_PARAM (1001, 0x10000);

int main(void)
{
 sys_spu_thread_t spu_worker_thread;
 sys_spu_thread_group_t spu_thread_group;
 int return_val;

 printf ("\n\nPPU: Hello world !\n");

 // Init the required number of SPUs ------------------------------------------

 return_val = sys_spu_initialize ( MAX_PHYSICAL_SPU , MAX_RAW_SPU );
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU : Couldn 't initialise SPUs ! %i\n", return_val) );

 // Load SPU program code/image ------------------------------------------

 sys_spu_image_t spu_img;

 printf ("PPU: loading spu code from: %s\n",SPU_PROG);
 return_val = sys_spu_image_open (& spu_img , SPU_PROG );
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU : sys_spu_image_open failed %x\n", return_val) );

 printf ("PPU: SPU program load success\n");

 // Create a Thread group ------------------------------------------

 sys_spu_thread_group_attribute_t group_attr ;
 sys_spu_thread_group_attribute_initialize(group_attr);
 sys_spu_thread_group_attribute_name(group_attr, "My Group");
 group_attr.type = SYS_SPU_THREAD_GROUP_TYPE_NORMAL;

 return_val = sys_spu_thread_group_create (&spu_thread_group, MAX_PHYSICAL_SPU, 100, &group_attr);
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: Thread group creation failed : %i\n", return_val) );

 // Create an SPU Thread, within the thread group --------------------------

 sys_spu_thread_attribute_t thread_attr ;
 sys_spu_thread_argument_t thread_args ;
 return_val = sys_spu_thread_initialize (&spu_worker_thread, spu_thread_group, 0, &spu_img, &thread_attr, &thread_args);
 if ( return_val != CELL_OK )
 {
  printf ("PPU: sys_spu_thread_initialize failed !\n");
  switch ( return_val )
  {
  case ( ESRCH ):
   printf (" Invalid Thread Group ID\n");
   break;
  case ( EINVAL ):
   printf (" spu_num out of range \n");
   break;
  case ( EBUSY ):
   printf (" Already initialised / used \n");
   break;
  case ( ENOMEM ):
   printf (" Memory allocation failed \n");
   break;
  case ( EFAULT ):
   printf (" Invalid address access \n");
   break;
  default :
   printf (" Error ! %i\n", return_val );
   break;
  }
  DBG_HALT;
 }

 // Enable Spu Print handling --------------------

 //priority 1000, NULL = no output re-routing
 return_val = spu_printf_initialize(1000, NULL);
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: spu_printf_initialize failed %i\n", return_val) );
 //Attach print handler to all threads in the group
 return_val = spu_printf_attach_group(spu_thread_group);
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: spu_printf_attach_group %i\n", return_val) );

 // Start the SPU thread group ----------------

 return_val = sys_spu_thread_group_start ( spu_thread_group );
 DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: sys_spu_thread_group_start failed %i\n", return_val) );

 // Wait for spu worker to finish ------------
 sys_spu_thread_group_join ( spu_thread_group , 0, 0);
 printf ("PPU: SPU worker finished\n");

 // Shutdown ------------------------------------------

 sys_spu_thread_group_destroy ( spu_thread_group );

 //Close the spu image
 return_val = sys_spu_image_close (& spu_img );
```

```
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: sys_spu_image_close failed %x\n", return_val) );

  printf ("PPU: Exiting ...\n");

  return 0;
}
```

4.2 SPU Threading with Events code

SPU events Sending events from an SPU and receiving them is very similar to sending PPU thread events. The difference is that instead of the thread creating a port and linking itself to the queue, a PPU thread must do it on behalf of the SPU thread. Before the SPU thread is started, the PPU sets up any links to any event queues. Creating an SPU event port is even simpler than on a PPU thread, instead of creating a port object, you just have to specify a unique number between 0x1(1) and 3F (63). [Remember that port 0x1 is used for the SPU printF protocol]

Listing 5. SPU threading + Events, SPU code

```
#include <spu_printf.h>
#include <sys/spu_event.h>

int main(void)
{
  spu_printf("SPU: Hello, world!\n");

  //send some events
  sys_spu_thread_send_event(0x10, 100, 200);
  sys_spu_thread_send_event(0x10, 300, 400);
  //The ppu will send an event when data.2 is 100
  sys_spu_thread_send_event(0x10, 100, 100);

  //Check our own queue for events and wait
  uint32_t data1, data2, data3;
  sys_spu_thread_receive_event(1337, &data1, &data2, &data3);

  spu_printf("SPU: Event received, shutting down\n");
  return 0;
}
```

SPU receiving events Notice that we are actually *receiving* events in this spu code, but receiving from what queue? A special SPU event queue is created in the PPU thread. Then an spu thread is bound to it, with a special identifier number, in this case '1337'. An Spu could be bound to many spu event queues, with different identifiers. This queue can be accessed by the spu code, just like the PPU queue manager thread does. With this system we can send events to the SPU, there are more efficient ways of communicating small data (flags and signal registers) to SPUs, so this code just here is to show how extensive the events system is.

Listing 6. SPU threading + Events, PPU code

```
#include <stdio.h>   // printf
#include <stdlib.h>  // exit function
#include <spu_printf.h>  // SPU printf functions
#include <sys/spu_initialize.h> // A initialise SPUs
#include <sys/spu_utility.h>  // SPU images
#include <sys/paths.h>    // SYS_APP_HOME
#include <sys/ppu_thread.h> // make PPU threads
#include <sys/spu_thread.h> // make SPU threads
#include <sys/event.h>    // process events
#include <sys/timer.h>  // Sleeps

//#define DBG_HALT { __asm__ volatile( "trap" ); }
#define DBG_HALT { exit (1); }
// call DBG_HALT on assert fail
#define DBG_ASSERT(exp) { if ( !(exp) ) {DBG_HALT;} }
// Prints the suplied string on assert fail, then call DBG_HALT
#define DBG_ASSERT_MSG( exp, smsg ) { if ( !(exp) ) {puts (smsg); DBG_HALT;} }
// Calls the suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC( exp, func) { if ( !(exp) ) {func; DBG_HALT;} }

#define PPU_STACK_SIZE 4096
#define PPU_PRIORITY 200
#define SPU_EVENT_PORT 0x10 //(16),  must be between 0x1(1) and 3F (63)
#define MAX_PHYSICAL_SPU 1
#define MAX_RAW_SPU 0

// ~~~@@~~~~~~~~~~~~~~~~~@@~~~
#define SPU_SELF "/ps3_tut_1_7_4_spu_threads_events_SPU.self"
// ~~~@@~~~~~~~~~~~~~~~~~@@~~~

#define SPU_PROG (SYS_APP_HOME SPU_SELF)

static sys_event_queue_t event_queue;
static sys_event_queue_t spu_queue;
void ppu_queue_manager_thread_entry(uint64_t arg);
void send_event_to_spu();

//Priority and stack size of the primary PPU thread of the game process
SYS_PROCESS_PARAM (1001, 0x10000);

int main(void)
{
  sys_ppu_thread_t qmgr_thread;
  sys_spu_thread_t spu_worker_thread;
  sys_spu_thread_group_t spu_thread_group;
  sys_event_queue_attribute_t queue_attr;

  int return_val;

  printf ("\n\nPPU: Hello world !\n");

  // Create and init and Event queue --------------------

  sys_event_queue_attribute_initialize ( queue_attr );
  return_val = sys_event_queue_create (&event_queue, &queue_attr, SYS_EVENT_PORT_LOCAL, 127);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU : sys_event_queue_create failed %i\n", return_val) );

  printf ("PPU: starting quene manager\n");

  // Create a PPU thread to watch the event queue --------------

  return_val = sys_ppu_thread_create (&qmgr_thread, ppu_queue_manager_thread_entry, 0, PPU_PRIORITY, PPU_STACK_SIZE,
                     SYS_PPU_THREAD_CREATE_JOINABLE, (char*)" spu_printf_server ");
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU :PPU thread creation failed : %i\n", return_val) );


  //Create an SPU event queue ----------------------
  // NOTE THIS HAS TO BE DONE BEFORE ANY SPU GROUPS ARE MADE, FOR SOME REASON
  sys_event_queue_attribute_t spu_queue_attr = {SYS_SYNC_FIFO, SYS_SPU_QUEUE};
  return_val = sys_event_queue_create(&spu_queue, &spu_queue_attr, SYS_EVENT_QUEUE_LOCAL, 64);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: sys_event_queue_create %i\n", return_val) );
```

```

// Init the required number of SPUs ↩
    −−−−−−−−−−−−−−−−−−
return_val = sys_spu_initialize ( MAX_PHYSICAL_SPU , ↩
    MAX_RAW_SPU );
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU : ↩
    Couldn 't initialise SPUs ! %i\n", return_val) );

// Load SPU program code/image −−−−−−−−−−−−−−−
sys_spu_image_t spu_img;

printf ("PPU: loading spu code from: %s\n",SPU_PROG);
return_val = sys_spu_image_open (& spu_img , SPU_PROG );
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU : ↩
    sys_spu_image_open failed %x\n", return_val) );

printf ("PPU: SPU program load success\n");

// Create a Thread group −−−−−−−−−−−−−−−−−
sys_spu_thread_group_attribute_t group_attr ;
sys_spu_thread_group_attribute_initialize(group_attr);
sys_spu_thread_group_attribute_name(group_attr, "My Group");
group_attr.type = ↩
    SYS_SPU_THREAD_GROUP_TYPE_NORMAL;

return_val = sys_spu_thread_group_create (&spu_thread_group, ↩
    MAX_PHYSICAL_SPU, 100, &group_attr);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    Thread group creation failed : %i\n", return_val) );


// Create an SPU Thread, within the thread group ↩
    −−−−−−−−−−−−
sys_spu_thread_attribute_t thread_attr ;
sys_spu_thread_argument_t thread_args ;
return_val = sys_spu_thread_initialize (&spu_worker_thread, ↩
    spu_thread_group, 0, &spu_img, &thread_attr, &thread_args);
if ( return_val != CELL_OK )
{
 printf ("PPU: sys_spu_thread_initialize failed !\n");
 switch ( return_val )
 {
 case ( ESRCH ):
  printf (" Invalid Thread Group ID\n");
  break;
 case ( EINVAL ):
  printf (" spu_num out of range \n");
  break;
 case ( EBUSY ):
  printf (" Already initialised / used \n");
  break;
 case ( ENOMEM ):
  printf (" Memory allocation failed \n");
  break;
 case ( EFAULT ):
  printf (" Invalid address access \n");
  break;
 default :
  printf (" Error ! %i\n", return_val );
  break;
 }
 DBG_HALT;
}

// Connect SPU thread to event queue ↩
    −−−−−−−−−−−−−−−−−−−−−

return_val = sys_spu_thread_connect_event ( spu_worker_thread, ↩
    event_queue,
                SYS_SPU_THREAD_EVENT_USER, ↩
    SPU_EVENT_PORT );
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    Event Queue connect failed : %i\n", return_val));

// Connect SPU thread group to event queue ↩
    −−−−−−−−−−−−−−

sys_spu_thread_group_connect_event(spu_thread_group, ↩
    event_queue, SYS_SPU_THREAD_GROUP_EVENT_RUN);
 sys_spu_thread_group_connect_event(spu_thread_group, ↩
    event_queue, ↩
    SYS_SPU_THREAD_GROUP_EVENT_EXCEPTION);
printf ("PPU: starting SPU worker\n");

// Enable Spu Print handling −−−−−−−−−−−−−−−−−−

//priority 1000, NULL = no output re−routing
return_val = spu_printf_initialize(1000, NULL);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    spu_printf_initialize failed %i\n", return_val) );
//attach print handeler to all threads in the group
return_val = spu_printf_attach_group(spu_thread_group);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    spu_printf_attach_group %i\n", return_val) );


//Connect SPU thread to it's own event queue ↩
    −−−−−−−−−−−−−−
//1337 is an abirtray queue number
//Each SPU thread identifies the bound event queue by this ↩
    number, instead of the event queue ID.
//Therefore, the SPU queue number must be unique within the ↩
    SPU thread.
return_val = sys_spu_thread_bind_queue(spu_worker_thread, ↩
    spu_queue, 1337);
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    sys_spu_thread_bind_queue %i\n", return_val) );


// Start the SPU thread group −−−−−−−−−−−−−−−−

return_val = sys_spu_thread_group_start ( spu_thread_group );
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    sys_spu_thread_group_start failed %i\n", return_val) );

// Wait for spu worker to finish −−−−−−−−−−−−−−−
sys_spu_thread_group_join ( spu_thread_group , 0, 0);
printf ("PPU: SPU worker finished\n");

// Shutdown −−−−−−−−−−−−−−−−−−−

sys_spu_thread_group_destroy ( spu_thread_group );
//destroy the Queue
sys_event_queue_destroy ( event_queue , ↩
    SYS_EVENT_QUEUE_DESTROY_FORCE );
//Using the Force mode, causes sys_event_queue_receive() in the ↩
    manager thread to return ECANCELED.

//Close the spu image
return_val = sys_spu_image_close (& spu_img );
DBG_ASSERT_FUNC((return_val == CELL_OK), printf("PPU: ↩
    sys_spu_image_close failed %x\n", return_val) );

//Let's wait for the manager thread to finish, just to be polite.
sys_ppu_thread_join(qmgr_thread,0);

printf ("PPU: Exiting ...\n");

return 0;
}

void ppu_queue_manager_thread_entry ( uint64_t arg ) {
int return_val;
sys_event_t event;
sys_spu_thread_t spu;

//keep looping, and checking for new events
while (1) {
 return_val = sys_event_queue_receive (event_queue, &event, ↩
    SYS_NO_TIMEOUT);
 if ( return_val != CELL_OK ) {
  if( return_val == ECANCELED ) {
   printf ("PPU:Q MGR: Event Queue destroyed ! Exiting ... \n"↩
    );
```

```
      } else {
        printf ("PPU:Q MGR: Event Queue receive failed : %i\n", return_val);
      }
      break ;
    }

    //New event!
    switch (event.source)
    {
    case (SYS_SPU_THREAD_GROUP_EVENT_RUN_KEY):
      //It's an SPU Group Run event!
      printf ("PPU:Q MGR: SPU Group Run Event!, SPU thread group ID: %i \n", (sys_spu_thread_group_t)event.data1);
      //could do a free(spu_elf_img); if you wish
      break;
    case (SYS_SPU_THREAD_GROUP_EVENT_EXCEPTION_KEY):
      //It's an SPU Group exeption event, Oh no!
      printf ("PPU:Q MGR: SPU Group exeption Event!, Group ID: %i, SPU thread ID:%i, SPU_NPC: %i, Cause: %i\n",
        (sys_spu_thread_group_t)event.data1), (uint8_t)(event.data1 >> 32),event.data2,(uint8_t)(event.data2 >> 32);
      break;
    case (SYS_SPU_THREAD_EVENT_USER_KEY):
      //It's an SPU user event!
      printf ("PPU:Q MGR: SPU USER Event!, Thread ID: %i, Port: %i, Data: %i, %i\n",
        (sys_spu_thread_t)event.data1,(uint8_t)(event.data2 >> 32), event.data2, event.data3);

      if (event.data3 == 100){
        printf ("PPU: spu has requested an event\n");
        //send an event back to the spu
        send_event_to_spu();
      }
      break;
    default :
      //It's some other type of event
      printf ("PPU:Q MGR: New Event!, Port %i, Data: %i, %i, %i\n", event.source,event.data1,event.data2,event.data3);
      break;
    }
  }

  printf ("PPU:Q MGR: Event Queue receive failed : %i\n", return_val);
  sys_ppu_thread_exit (0);
}

void send_event_to_spu(){

  //create an Event Port
  sys_event_port_t port;
  int return_val = sys_event_port_create(&port, SYS_EVENT_PORT_LOCAL, SYS_EVENT_PORT_NO_NAME);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("sys_event_port_create failed %x\n", return_val ));

  //link port to Main event queue (which is global and static so we can use it in this scope)
  return_val = sys_event_port_connect_local(port, spu_queue);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("sys_event_port_connect_local %x\n", return_val ));

  //sleep for 2 seconds
  sys_timer_sleep(2);

  //Let's send an event!
  //we can send 3x 64-bit data (e.g 3 ints), wow such data!
  printf ("PPU: Sending event from PPU to SPU\n");
  return_val = sys_event_port_send(port, 0, 0, 0);
  DBG_ASSERT_FUNC((return_val == CELL_OK), printf ("PPU: sys_event_port_send failed %x\n", return_val ));
}
```

5. Conclusion

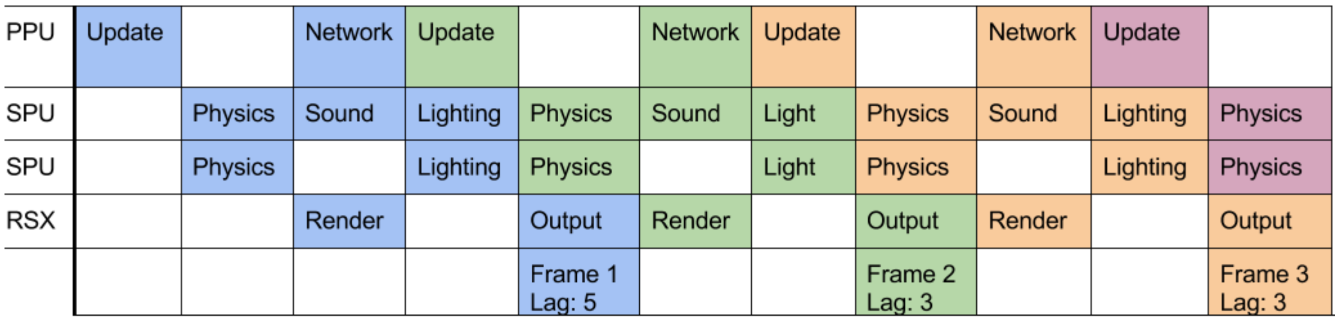
This tutorial was an introduction to delegating work out to separate threads, and separate processing elements. The SPUs are extremely powerful if their specialised instruction set is utilised correctly. Almost all of the 'Triple A' Playstation 3 titles are build on top of a robust 'Task managing system', that abstracts most of the content covered in this tutorial into a higher level system. As you can see in figure 5, having multiple operations running in parallel is a must for good performance.

Recommended Reading

Programming the Cell Processor: For Games, Graphics, and Computation, Matthew Scarpino, ISBN: 978-0136008866
Vector Games Math Processors (Wordware Game Math Library), James Leiterman, ISBN: 978-1556229213
Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

References

[1] Edinburgh Napier Game Technology Website. www.napier.ac.uk/games/. *Accessed: Feb 2014*, 2014. 1



PPU	Update		Network	Update		Network	Update		Network	Update	
SPU		Physics	Sound	Lighting	Physics	Sound	Light	Physics	Sound	Lighting	Physics
SPU		Physics		Lighting	Physics		Light	Physics		Lighting	Physics
RSX			Render		Output	Render		Output	Render		Output
					Frame 1 Lag: 5			Frame 2 Lag: 3			Frame 3 Lag: 3

Figure 5. Example Render Pipeline - Heavy use of threading