Lesson 1 - Introduction to SPU programming

Summary

This lesson will teach you the basics of running code on one of the PlayStation 3's Synergistic Processing Units by getting an SPU to send the phrase 'Hello World' to the terminal output.

New Concepts

Loading SPU code, SPU initialisation, creating SPU and PPU threads, Event Queues

Hello World SPU Code

Create two Visual Studio projects: a PPU project called **HelloWorldPPU**, and an SPU project called **HelloWorldSPU**, according to the steps outlined in tutorial 0. Add a new source file to **HelloWorldSPU**, and add the following code to it.

```
1 #include <spu_printf.h>
2
3 int main(void)
4 {
5    for(int i = 0; i < 100; ++i) {
6       spu_printf("SPU: Hello world!\n");
7    }
8    return 0;
9 }
```

helloworld.spu.cpp

Looks pretty easy, yes? The code should be pretty familiar to you, if you've ever used the printf function to display output. Remember that when SPU programming, you do not have access to std::cout, you have to use the equivalent C function, so remember to include its header file. Additionally, the spu_printf function doesn't work quite how you might initially think it does; rather than sending text to the terminal output as you would expect, it sends an *event* to one of the SPU's ports, containing a pointer to our output text. Think of events as being triggered messages containing data.

SPU ports are conceptually similar to the ports used for communication over a network so we can send data to a any port we like, but if there's nothing listening to that port, nothing will happen! So that our events do something, we must set up an event queue in a PPU program to receive and process messages sent from an SPU. To make this tutorial a little more useful, we are going to put the code to handle the event queue in its own thread on the PPU, so that our program doesn't stall while sitting waiting for the SPU to send a message. Suddenly getting an SPU to say 'Hello World' doesn't sound so easy, does it!

Hello World PPU Code

Overview

In our PPU code, we are going to do the following:

1) Initialise the SPU
2) Load the SPU code we compiled in the previous section
3) Create an SPU thread to run our SPU "hello world" code
4) Link an event queue to our SPU thread to receive its output
5) Create a PPU thread to handle the output passed to the event queue
6) Wait for our SPU thread to finish, and then clean up and exit

Although you will have create threads before, doing so when the SPU is involved is a slightly different process. Also, message queues are likely to be new to you, so don't worry if you haven't heard of them before. It is worth noting that there are some functions in the Sony SDK for simplifying the process of outputting SPU printf statements, by automating and hiding the PPU thread and Event Queue creation steps. While this is nice, it is more beneficial to you to know how to create PPU threads as early as possible, so in this tutorial we will be doing everything manually.

Includes and definitions

Add a new source file called **HelloWorld.ppu.cpp** to your PPU project. The remaining code for this tutorial will all be PPU code, and will be added to this source file. The first code we'll add to this source file is some **#include** definitions and some macros.

```
1 #include <stdio.h>    //We want to be able to printf
2 #include <stdlib.h>   //And be able to call the exit function
3 #include <spu_printf.h> //And access to the SPU printf functions
4 #include <sys/spu_initialize.h>  //And initialise SPUs
5 #include <sys/spu_utility.h>     //And load in SPU images
6 #include <sys/paths.h>           //And know about SYS_APP_HOME
7 #include <sys/ppu_thread.h>      //And make PPU threads
8 #include <sys/spu_thread.h>      //And make SPU threads
9 #include <sys/event.h>           //And process events
```

helloworld.ppu.cpp

These are the headers we need to load and use SPU threads. You'll be seeing these a lot in these tutorials! You should be pretty used to the **#include** preprocessor directive by now, so let's move on. After the include directives, add the following code:

```
10 #define PPU_STACK_SIZE        4096
11 #define PPU_PRIORITY          200
12 #define SPU_PRINTF_PORT       0x1
13 #define MAX_PHYSICAL_SPU      1
14 #define MAX_RAW_SPU           0
15 #define SPU_SELF   "/HelloWorldSPU.self"
16 #define SPU_PROG   (SYS_APP_HOME SPU_SELF)
```

helloworld.ppu.cpp

These are the macros used in the PPU code. I'll explain each macro as it is used in the code. Firstly take a look at the *SPU_PROG* macro. Note how it is made up of two macros, the *SPU_SELF* macro defined on the line above it, and the *SYS_APP_HOME* macro. This macro is defined in the **paths.h** header file included on line 6, and is itself set to the value set in the ProDG VSI Project properties you were introduced to in Lesson 0. *SPU_SELF* should be set to the name of your SPU project if you're following this tutorial to the letter **HelloWorldSPU.self** will be correct, but double check! With the macros over with, it's time to look at our static variables and function declarations.

```
17 static sys_event_queue_t          print_queue;
18
19 void ppu_thread_entry(uint64_t arg);
20
21 SYS_PROCESS_PARAM(1001, 0x10000)
```

helloworld.ppu.cpp

sys_event_queue_t is a structure used by the PS3 API. It'll be described later, but for now just know that the output of our SPU program will be accessed through this struct. **ppu_thread_entry** is a function declaration this function will be passed on to the PPU thread we are about to create, and will handle the output of incoming SPU-side printf commands. Finally, we have the preprocessor directive **SYS_PROCESS_PARAM(1001, 0x10000)**. This is used by the PS3 compiler to set the priority and the stack size of the compiled program. These are the default values provided by Sony, and will set your programs priority to **1001**, and the stack size to **64kb**. You should know about the program stack by now, and how to read hex values, so lets move on to the main function of our 'hello world' program!

Main Function

The main function will contain the code required to perform each of the steps outlined in the overview section, starting with loading in the SPU program we created earlier.

Local variables

We begin the main function much as we would with any C++ program, and then define some local variables.

```
22 int main(void) {
23    sys_ppu_thread_t                 thread;
24    sys_spu_thread_t                 thread_spu;
25    sys_spu_thread_group_t           thread_group;
26    sys_event_queue_attribute_t      queue_attr;
27
28    int return_val;
29
30    printf("PPU: Hello world!\n");
```

helloworld.ppu.cpp

You probably won't have seen these variable types before, but their names should give you a pretty good idea what they do. **sys_ppu_thread_t** and **sys_spu_thread_t** are structs to represent a PPU thread and SPU thread, respectively, while **sys_spu_thread_group_t** is a pointer to an SPU thread group these will be explained shortly. **sys_event_queue_t** is the event queue mentioned earlier, while **sys_event_queue_attribute_t** holds the *attributes* attached to the queue. On line **28**, we define an integer, **return_val**. As the name suggests, this variable is used to store the value returned by various Sony API functions used throughout our main function. On line 29, we use printf to print out Hello world!, just to show how simple it is to print when not doing it through the SPU. Note that The phrase has been prefixed with "PPU: ", this is just to make it absolutely clear when examining the terminal output which processor outputted which print statement.

Initialising the SPU

To initialise our SPU we have our first Sony API function, **sys_spu_initialize**. This function basically 'switches on' the number of SPUs we require. Its first input variable determines the number of normal SPUs to enable, while the second determines the number of 'raw' SPUs. The difference between normal and raw SPUs is beyond the scope of this tutorial, so simply note the use of the macros we defined earlier, to enable 1 normal SPU and 0 'raw' SPUs.

See how we place the function's return value into **return_val**, and then use an **if** statement in conjunction with the SDK macro '**CELL_OK**' to determine what to do if a function 'fails', which for the purposes of this tutorial is to output an error message and then exit.

```
31    return_val = sys_spu_initialize(MAX_PHYSICAL_SPU, MAX_RAW_SPU);
32    if (return_val != CELL_OK) {
33       printf("PPU: Couldn't initialise SPUs!\n");
34       exit(return_val);
35    }
```

helloworld.ppu.cpp

Loading the SPU code

On line 36 we create a variable of type **sys_spu_image_t**. This is simply a struct that will hold pointers and management information related to our SPU image once we have loaded it into memory. Loading the SELF file is done on line 37, by passing a reference to our new struct to the SDK function '**sys_spu_image_open**', in addition to using our **SPU_PROG** macro, which defines the path and file name of the SELF file to load. As with our SPU initialisation code, we can check **return_val** to determine the success of our SELF image loading. Assuming the image open function has been correct, the SELF image that was created by our SPU project will now have been loaded into memory and pointed to by the **spu_img** variable, where it can be passed on to an SPU for execution.

```
36    sys_spu_image_t spu_img;
37    return_val = sys_spu_image_open(&spu_img, SPU_PROG);
38    if (return_val != CELL_OK) {
39       printf("PPU: sys_spu_image_open failed %x\n", return_val);
40       exit(return_val);
41    }
```

helloworld.ppu.cpp

Creating an SPU Thread Group

Before we can run any threads on the SPU, we must create a 'thread group'. These allow related SPU threads to be collected together for ease of management and context switching. For example you might have a thread group for rendering threads, and a thread group for AI threads, and use them to switch what your SPUs are processing as required. For the purposes of this tutorial we only need one thread group, so let's have a look at how to create one.

```
42    sys_spu_thread_group_attribute_t group_attr;
43    group_attr.type = SYS_SPU_THREAD_GROUP_TYPE_NORMAL;
44
45    return_val = sys_spu_thread_group_create(&thread_group,
46                    MAX_PHYSICAL_SPU, 100, &group_attr);
47    if (return_val != CELL_OK) {
48       printf("PPU: Thread group creation failed:  %i\n", return_val);
49       exit(return_val);
50    }
```

helloworld.ppu.cpp

You should be pretty familiar with this pattern by now - we call a Sony SDK function, then check its success. **sys_spu_thread_group_create** requires 4 parameters - a reference to the thread group to create, the number of SPUs in the group, a priority number, and a reference to an attribute structure. We create this structure on line 42, while our thread group structure we created way back on line 25.

We define the number of SPUs to use in a macro back on line 13, while the priority is set to 100 - the default value suggested by Sony. Raising this priority will give thread groups more processing time when using multiple thread groups, but as we are only using one group, it isn't important.

Creating the SPU Thread

Now that the thread group is created, we can initialise an SPU thread to run in it. Unlike normal threads which accept a function pointer, SPU thread initialisation requires references to our SPU code and thread group.

```
51    sys_spu_thread_attribute_t    thread_attr;
52    sys_spu_thread_argument_t     thread_args;
53
54    return_val = sys_spu_thread_initialize(&thread_spu, thread_group,
55                 0, &spu_img, &thread_attr,  &thread_args);
56
57    if (return_val != CELL_OK) {
58       printf("PPU: sys_spu_thread_initialize failed!");
59       switch(return_val) {
60          case(ESRCH):  printf("Invalid Thread Group ID\n")   ;break;
61          case(EINVAL): printf("spu_num out of range\n")      ;break;
62          case(EBUSY):  printf("Already initialised / used\n");break;
63          case(ENOMEM): printf("Memory allocation failed\n")  ;break;
64          case(EFAULT): printf("Invalid address access\n")    ;break;
65          default:printf("Error! %i\n", return_val            ;break;
66       }
67       exit(return_val);
68    }
```

helloworld.ppu.cpp

We're using the same pattern here again, this time with an additional switch statement - this is to demonstrate how to check against the value returned by an SDK function. To initialise an SPU thread, we use the **sys_spu_thread_initialize** SDK function, passing it a reference to the **sys_spu_thread_t** we defined on line 23, a reference to the SPU thread group we have just initialised, a reference to the SPU code we loaded earlier, and references to a couple of new variables we declare on lines 51 and 52, of types **sys_spu_thread_attribute_t** and **sys_spu_thread_argument_t**. The **sys_spu_thread_attribute_t** structure allows setting of attributes such as a name for a particular thread, while **sys_spu_thread_argument_t** sets the arguments that will be passed to the main function of the SPU code we send to the SPU. The second parameter is the number of the SPU in the thread group to assign the thread to. You might have a number of SPUs in a thread group, all requiring initialising with different code, so this parameter sets which SPU gets initialised. Just as with arrays, the index numbering begins at 0, so as we are initialising the first SPU in the thread group, we use 0 as the parameter.

You might have been wondering what **return_val** is set to if it isn't set to **CELL_OK**. This will depend on the function called, so look up **'sys_spu_thread_initialize**' in the SDK documentation. You'll find that it returns one of 6 values, either **CELL_OK** or 5 error values. We can check and output information relating to each error an SDK function can return by using a **switch** statement. Programming the PS3 can be tricky, so being able to output relevant debugging information such as error codes is a useful tool, and you should use it when possible. The **switch** statement defined on line 59 is to provide an example of this specific error outputting.

Creating and connecting the Event Queue

With the SPU thread created, we can move on to creating an Event Queue to handle its output. Every time the SPU thread calls spu_printf, an event is added to the Event Queue attached to the thread. This queue can then be polled in a PPU thread, and its data handled. The following code will create an Event Queue, and connect it to the SPU thread.

```
69    sys_event_queue_attribute_initialize(queue_attr);
70
71    return_val = sys_event_queue_create(&print_queue, &queue_attr,
72                        SYS_EVENT_PORT_LOCAL, 127);
73    if (return_val != CELL_OK) {
74       printf("PPU: sys_event_queue_create failed %i\n", return_val);
75       exit(return_val);
76    }
77
78    return_val=sys_spu_thread_connect_event(thread_spu, print_queue,
79                    SYS_SPU_THREAD_EVENT_USER,
80                    SPU_PRINTF_PORT);
81
82    if (return_val!=CELL_OK) {
83       printf("PPU:  Event Queue connect failed: %i\n", return_val);
84       exit(return_val);
85    }
```

helloworld.ppu.cpp

As with SPU thread initialisation, we require an *attributes* structure for our Event Queue, which in this case must additionally be initialised with default data by the **sys_event_queue_attribute_initialize** function. We then use the **sys_event_queue_create** SDK function to create our queue, passing references to the **sys_event_queue_t** we defined on line 17, and the **sys_event_queue_attribute_t** we defined on line 26, and initialised on line 69. Arguments 3 and 4 of the **sys_event_queue_create** require further explanation. Argument 3 is a unique 'key' value used to identify the queue, which for the purposes of this tutorial is set to the SDK defined key '**SYS_EVENT_PORT_LOCAL**'. This key sets the queue to be visible to only the process that created it. The last argument defines the maximum size of the event queue, and must be a value between 1 and 127. The larger the maximum queue size, the more memory it will take up, but as we don't have to worry about that for our simple program, we can safely set it to the maximum allowed.

After checking the SDK function has executed correctly, we can connect the Event Queue to our SPU thread, allowing its **printf** statements to be sent to the Event Queue. We do this on line 78, with the SDK function **sys_spu_thread_connect_event**. We pass to it our SPU thread and print queue, as well as two special values: **SYS_SPU_THREAD_EVENT_USER** and **SPU_PRINTF_PORT**. **SYS_SPU_THREAD_EVENT_USER** marks the event as a type that can be be sent arbitrarily in the SPU code, as opposed to an automatically generated event such as a DMA completion event notice. The last argument assigns a port number to the Event Queue. **spu_printf** sends events to port 1 (N.B see page 33 of the Lv2 Users Manual for a reference to this) so we set the argument to **SPU_PRINTF_PORT**, the macro we defined on line 12, which is set to a value of 1.

Creating the PPU thread

With the Event Queue initialised and attached, we can create the PPU thread that will handle the incoming **spu_printf** events. We do this using the **sys_ppu_thread_create** SDK function. This function requires 6 arguments to be passed to it, which requires some explanation. The first argument is a reference to a **sys_ppu_thread_t** , so we pass it out 'thread' variable we defined on line 23. The second argument is a pointer to the function we wish to run in the thread, so we pass it the function declaration we made on line 19. The third argument defines a variable to send as an argument to the function we will be using in our thread. As we don't require sending anything of any interest to our function, we can just leave it at 0. The next two arguments are a priority and stack size for the thread, so we use the PPU_PRIORITY and PPU_STACK_SIZE macros we defined on lines 10 and 11, which are set to Sony's default values of 200 and 4096, respectively. The fifth argument sets the flags used when creating the PPU thread. We want our thread to be joinable, so we use the value **SYS_PPU_THREAD_CREATE_JOINABLE**. The final argument is a name for the thread, which is used within the debugger to aid in the identification of threads, so we can pass any string to this argument, although it should be something unique if you want to be able to do any useful debugging!

```
86    return_val = sys_ppu_thread_create(&thread, ppu_thread_entry, 0,
87                 PPU_PRIORITY, PPU_STACK_SIZE,
88                 SYS_PPU_THREAD_CREATE_JOINABLE,
89                 (char*)"spu_printf_server");
90
91    if (return_val != CELL_OK) {
92       printf ("PPU:PPU thread creation failed: %i\n", return_val);
93       exit(return_val);
94    }
```

helloworld.ppu.cpp

Starting the SPU thread group

Now we have our PPU thread sitting waiting for incoming messages, we can start up our SPU group and start sending some 'Hello World' messages! On line 95, we use the **sys_spu_thread_group_start** SDK function, passing our thread group as an argument. As its name suggests, this function will start all of the threads in an SPU thread group.

```
95    return_val = sys_spu_thread_group_start(thread_group);
96
97    if (return_val != CELL_OK) {
98       printf("PPU:sys_spu_thread_group_start failed %i\n", return_val);
99       exit(return_val);
100   }
```

helloworld.ppu.cpp

Cleaning up

Our SPU thread will now happily spit out "Hello World", and our PPU will process it and output it to the terminal. Once the SPU thread has completed, we should destroy our unneeded resources so we can exit cleanly.

```
101   sys_spu_thread_group_join(thread_group, 0, 0);
102   sys_spu_thread_group_destroy(thread_group);
103   sys_event_queue_destroy(print_queue,SYS_EVENT_QUEUE_DESTROY_FORCE);
104
105   return_val = sys_spu_image_close(&spu_img);
106   if (return_val != CELL_OK) {
107      printf("PPU: sys_spu_image_close failed %x\n", return_val);
108      exit(return_val);
109   }
110
111   printf("PPU: Exiting...\n");
112
113   return 0;
```

helloworld.ppu.cpp

The **sys_spu_thread_group_join** function on line 101 joins our SPU thread group, and can also optionally store the exit cause and status in the second and third arguments. We aren't too concerned with how and why our SPU thread exited, so we don't use this feature, setting both output pointers to 0. Once our SPU thread has joined, we can destroy the thread group using the **sys_spu_thread_group_destroy** SDK function on line 102. With the SPU thread group destroyed, we can destroy the Event Queue, using the **sys_event_queue_destroy** SDK function on line 103. This will also cause our PPU thread to exit, as it no longer has an Event Queue to check. All that remains is to free up the memory used by our SPU code, which we do using the **sys_spu_image_close function** on line 105, before returning a value of 0.

You might initially be tempted to think that closing the image should be done straight after loading it and sending it to a SPU, but the **sys_spu_image_t** structure contains management information required by the SPUs that run the image code, so it cannot be safely destroyed until all SPU threads using the code have finished their execution.

The PPU thread code

The final piece in our SPU Hello World puzzle is our PPU thread function. This function waits for incoming events from the Event Queue attached to our SPU thread, and outputs them as text strings.

```
114 void ppu_thread_entry(uint64_t arg) {
115    int return_val;
116    sys_event_t event;
117    sys_spu_thread_t spu;
118
119    for(;;) {
120       return_val = sys_event_queue_receive(print_queue, &event,
121             SYS_NO_TIMEOUT);
122       if (return_val != CELL_OK) {
123          if(return_val == ECANCELED) {
124             printf("PPU thread: Event Queue destroyed! Exiting...\n");
125          }
126          else{
127             printf("PPU thread: Event Queue receive failed: %i\n",
128             return_val);
129          }
130          break;
131       }
132
133       spu = event.data1;
134
135       int sret = spu_thread_printf(spu, (std::uint32_t)event.data3);
136
137       return_val = sys_spu_thread_write_spu_mb(spu, sret);
138       if (return_val != CELL_OK) {
139          printf("PPU thread: SPU mailbox failed:  %i\n", return_val);
140          break;
141       }
142    }
143    sys_ppu_thread_exit(0);
144 }
```

helloworld.ppu.cpp

Like the main function, thread functions can have an argument passed to it, although in this example we don't use one. We define 3 local variables, an integer to store our SDK function return values (you should be well used to this concept by now!) and two SDK structs a variable of type **sys_event_t**, and a variable of type **sys_spu_thread_t**. We will use the **sys_event_t** variable as a reference to store our incoming event data, and the **sys_spu_thread_t** variable to act as a reference to the SPU thread that sent the message. On line 119 we enter into an infinite loop, which will process events for as long as events are successfully received. On line 120 we call the SDK function **sys_event_queue_receive**, which takes an Event Queue, a reference to a **sys_event_t** structure, and a microsecond counter structure as arguments. This counter defines how long the **sys_event_queue_receive** function should wait for an incoming event before failing. For our purposes, we want the function to wait forever, so we pass to it the special value **SYS_NO_TIMEOUT**, which as its name implies, will make the function sit and wait for an incoming event before returning.

As with other SDK functions, **sys_event_queue_receive** will return **CELL_OK** if successful. We should check the return value, and break out of our infinite loop if it fails, so that the thread exiting function on line 143 can be called. Note how on line 123 we check for a value of **ECANCELED** a look through the SDK documentation should tell you that this value is outputted if the event queue being checked is destroyed, which we do on line 103. By checking for this occurrence, we can output relevant information. In this example, Event Queue destruction is desired behaviour, so we shouldn't output an error string in the event of an **ECANCELED** return value. With our error cases handled, we can move on to what to do if we successfully receive an event - output it!

We update our **sys_spu_thread_t** variable on line 133 to contain a reference to the SPU thread that sent the event. We then call an SDK function, **spu_thread_printf**, to handle the actual terminal output of our incoming SPU **printf** Event. Sadly this function is undocumented, but it takes two arguments, the reference to the SPU thread that sent the event, and a pointer to some data, in this case the string to output. Notice how we cast this value from a 64 bit unsigned integer to a 32 bit unsigned integer. This is a peculiarity to the PlayStation 3 - It has a 64 bit address space, but SPU local store memory addresses are always expressed in 32 bits. This means that in general, PPU-side functions use 64 bit memory addresses, while SPU-side functions use 32 bit memory addresses. The addressing scheme means that both can be safely casted to the other (remember, the PS3 only has 512Mb of RAM in total including graphics memory, which can be safely expressed in 32 bits). We must also tell the SPU that its **spu_printf** function has successfully completed, which we do using the (again, sadly undocumented) function **sys_spu_thread_write_spu_mb**. The actual mechanism used for SPU communication in this function is the SPU 'mailbox' feature, which will be explained in a later tutorial. For now, it is enough to understand that you can send small packets of data to an SPU, and that is what this function does. Note how we pass it the SPU reference we assigned on line 133, and the return value of our call to **spu_thread_printf**. As usual, we check **return_val** for errors, and break out of our infinite loop if an error has occurred.

When the infinite loop is broken out of, either by an error or as the outcome of destroying our Event Queue, the **sys_ppu_thread_exit** function on line 143 will be called. This function takes one argument, which is simply an integer, that works in the same manner as the return value in a main function, informing the caller of the reason for exiting. For something as simple as a Hello World program, we don't really care why the thread has exited, so we just use the value 0.

Summary

That took a lot of work, didn't it? 144 lines of code, just to receive 'hello world' from one of the PS3's SPU processors! However, in this tutorial you have learnt pretty much everything you need to know to get started with programming on the Playstation 3. You've learnt how to create threads for execution on both the PPU and SPU, how SPUs are grouped together, how to write and compile SPU code as well as execute it, and even done a little bit of SPU-PPU communication! In a roundabout way, this has demonstrated an important concept of Playstation 3 programming - Synergistic Processing Units are not really designed to be general purpose. They don't excel at the type of I/O processing that printf statements require - they are much better at receiving a chunk of data, doing some processing on it, and spitting it back out again. In the next tutorial, we'll take a look at doing just that.

Further Work

1) Try getting two SPUs both outputting printf statements. How many PPU threads do we need? How many Event Queues do we need?

2) All this program does is receive printf statements from the SPU. Do we really need a separate PPU thread just to do that? Do we need all that error checking? Have a go at trying to make a smaller PPU program that successfully processes all of the SPU program's printf statements.

3) PPU thread functions can have an argument passed to them. How could we make use of this argument? Can we avoid having to create any static variables?