Lesson 2
Playstation 3 Development

Introduction to Playstation 3 Programming

Sam Serrels and Benjamin Kenwright[1]

Abstract
An introduction to the system architecture of the Sony Playstation 3 (PS3). The features of the Cell Broadband Engine (the main processor used by the PS3) and the Nvidia RSX 'Reality Synthesizer' graphics processor will be explained. A starting guide to programming on the PS3 is also included, which details some essential knowledge needed when writing code for this architecture.

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 1
2 PS3 System Architecture 1
3 The Cell processor 2
3.1 The PPE . . . 2
3.2 The SPE . . . 3
4 The RSX 3
4.1 Vram . . . 3
4.2 GCM and PSGL . . . 3
5 Writing code for the Ps3 3
5.1 Memory alignment . . . 3
5.2 Debugging . . . 4
5.3 A simple PPU program . . . 6
6 Conclusion 6
References 7

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 [4].

This Tutorial The Playstation 3 is a specialised device, the potential computing power of its design could outperform anything else in it's time.
However this power relies on careful design decisions, flawless code and a deep knowledge of every internal system, earning it a reputation as a difficult beast to tame.
The toolset for working with the PS3 are well equipped and there is plenty of documentation, but with any new system, there is a steep learning curve and a large amount of information to absorb before becoming you can get up and running.
This document tries to condense most of the critical information about the architecture of the system into one place.

Starting point This tutorial assumes you have read the previous tutorial on compiling and deploying applications to the PS3. This tutorial will cover starting a PS3 program from scratch rather than opening a sample project.

Additional Reading In addition to the lesson tutorials, we would recommend reading a number of books on Playstation 3 development and cross-platform coding, such as, Cell Programming for the PS3 [3], Vector Maths and Optimisation for the PS3 [1], and Cross-Platform Development in C++ [2].

2. PS3 System Architecture

The PS3 The Cell microprocessor, designed by Sony, Toshiba and IBM, is the used as the CPU, which is made up of one 3.2 GHz PowerPC-based ”Power Processing Element” (PPE) and eight Synergistic Processing Elements (SPEs). The eighth SPE is disabled to improve chip yields. Only six of the seven SPEs are accessible to developers as the seventh SPE is reserved by the console's operating system.
Graphics processing is handled by the NVIDIA RSX 'Reality

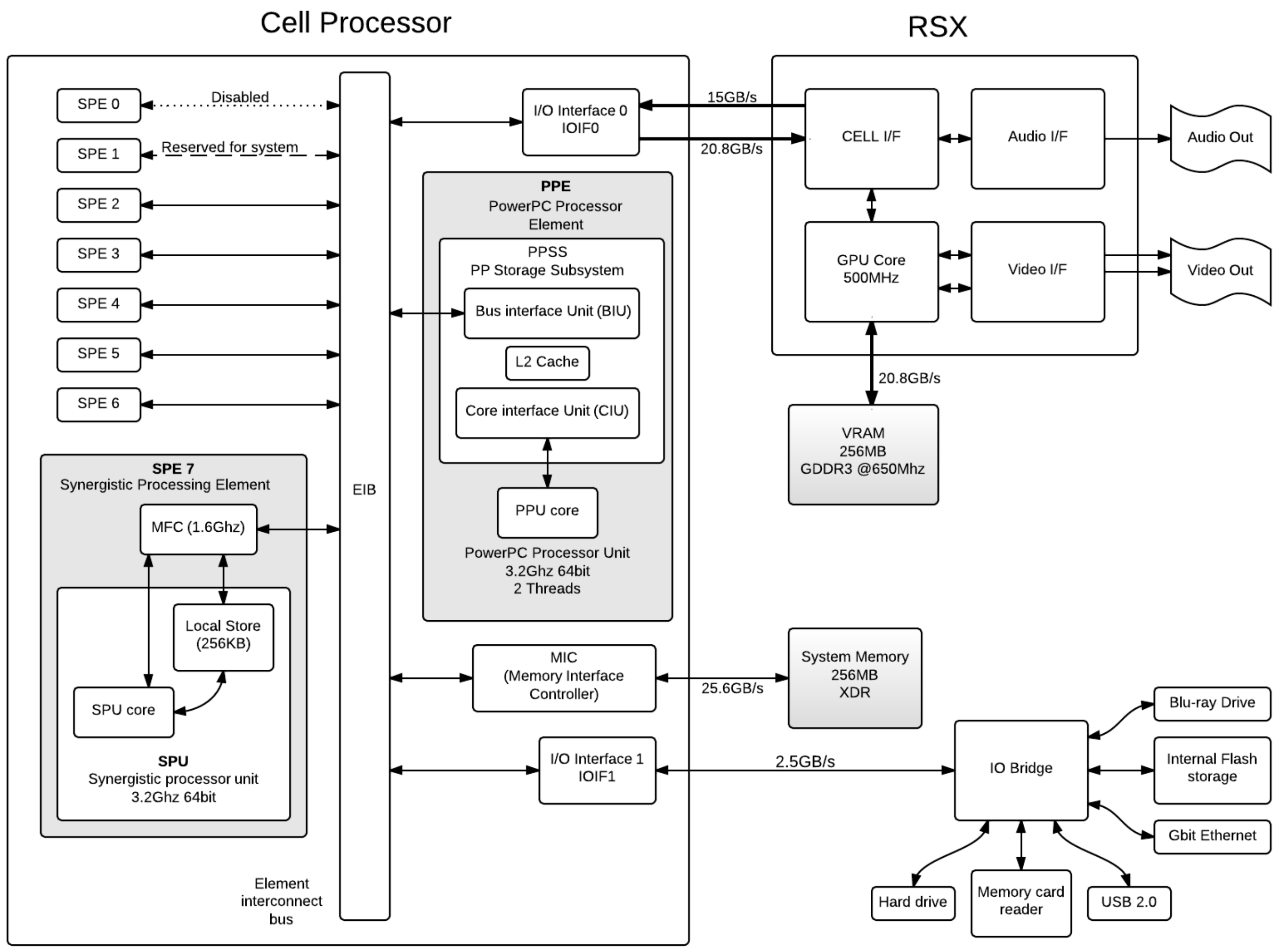

Figure 1. PS3 System Diagram - A high level overview of the main system components

Synthesizer', which can produce resolutions from 480i/576i SD up to 1080p HD.
The PlayStation 3 has 256 MB of XDR DRAM main memory and 256 MB of GDDR3 video memory for the RSX.
All PS3 models have user-upgradeable 2.5" SATA hard drives and come installed with drives of various sizes up to 500 GB.
The system has Bluetooth 2.0 (with support for up to 7 bluetooth devices), gigabit Ethernet, 2x speed Blu-ray Disc drive, USB 2.0 and HDMI 1.4 built in on all currently shipping models.
Wi-Fi networking and a flash card reader (compatible with Memory Stick, SD/MMC and CompactFlash/Microdrive media) is built-in on most models.

3. The Cell processor

The PS3 uses the Cell microprocessor, which is made up of one 3.2 GHz PowerPC-based "Power Processing Element" (PPE) and six accessible Synergistic Processing Elements (SPEs). A seventh runs in a special mode and is dedicated to aspects of the OS and security, and an eighth is a spare to improve production yields. PlayStation 3's Cell CPU achieves a theoretical maximum of 230.4 GFLOPS in single precision floating point operations and up to 100 GFLOPS double precision using iterative refinement for the solution of linear equations. The PS3 has 256 MB of Rambus XDR DRAM, clocked at CPU die speed.
Cell is a multi-core microprocessor microarchitecture which can have a number of different configurations, the basic configuration is a multi-core chip composed of one "Power Processor Element" ("PPE") (sometimes called "Processing Element", or "PE"), and multiple "Synergistic Processing Elements" ("SPE").The PPE and SPEs are linked together by an internal high speed bus dubbed "Element Interconnect Bus" ("EIB").

3.1 The PPE

The PPE is the Power Architecture based, two-way multithreaded core acting as the controller for the eight SPEs, which handle most of the computational workload. The PPE will work with conventional operating systems due to its similarity to other 64-bit PowerPC processors, while the SPEs are designed for vectorized floating point code execution. The PPE contains a 64 KiB level 1 cache (32 KiB instruction and a 32 KiB data) and a 512 KiB Level 2 cache.

3.2 The SPE

Each SPE is composed of a ”Synergistic Processing Unit”, SPU, and a ”Memory Flow Controller”, MFC.
The SPU runs a specially developed instruction set (ISA) with 128-bit SIMD organization for single and double precision instructions. Each SPE contains a 256 KB embedded SRAM for instruction and data, called ”Local Storage”which is visible to the PPE and can be addressed directly by software. (Not to be mistaken for ”Local Memory”, which is VRAM on the RSX) The local store does not operate like a conventional CPU cache since it is neither transparent to software nor does it contain hardware structures that predict which data to load. Note that the SPU cannot directly access system memory; the 64-bit virtual memory addresses formed by the SPU must be passed from the SPU to the SPE memory flow controller (MFC) to set up a DMA operation within the system address space. In one typical usage scenario, the system will load the SPEs with small programs (similar to threads), chaining the SPEs together to handle each step in a complex operation.

An SPE can operate on sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single-precision floating-point numbers in a single clock cycle, as well as a memory operation. At 3.2 GHz, each SPE gives a theoretical 25.6 GFLOPS of single precision performance. For double-precision floating point operations, as sometimes used in personal computers and often used in scientific computing, Cell performance drops by an order of magnitude, but still reaches 20.8 GFLOPS (1.8 GFLOPS per SPE, 6.4 GFLOPS per PPE). Compared to Desktop processors at the time of release, the relatively high overall floating point performance of a Cell processor seemingly dwarfs the abilities of the SIMD unit in CPUs like the Pentium 4 and the Athlon 64.
However, comparing only floating point abilities of a system is a one-dimensional and application-specific metric. Unlike a Cell processor, such desktop CPUs are more suited to the general purpose software usually run on personal computers. As to be expected, modern day desktop processors have caught up and overtaken the PS3 Cell processor in almost all of it’s strengths due to advances in multi-core and multi-threaded optimisations and software design.
A further difference to desktop processors is that the SPU has no branch prediction, features in the compiler are used to compensate for this. Code analysis at compile time is used to add in prepare-to -branch ’hints’ into the code.

4. The RSX

The RSX ’Reality Synthesizer’ is a proprietary graphics processing unit (GPU) co-developed by Nvidia and Sony for the PlayStation 3 game console. It is a GPU based on the Nvidia 7800GTX graphics processor and, according to Nvidia, is a G70/G71 hybrid architecture with some modifications. The RSX has separate vertex and pixel shader pipelines. The GPU makes use of 256 MB GDDR3 RAM clocked at 650 MHz, this is referred to as ”Local Memory” in the Sony documentation.

Specifications
- 500 MHz on 90 nm process (shrunk to 65 nm in 2008 and to 40 nm in 2010)
- 256 MB of GDDR3 memory running at 700MHZ.
- Multi-way parallel FP shader pipelines.
- Independent Vertex/Pixel shaders.
- Programmable shading processors – 136 shader operations per cycle.
- 128-bit pixel precision.
- Support for PSGL (OpenGL ES 1.1 + Nvidia Cg)
- Support for S3TC texture compression

Comparisons Here is the RSX up against some other graphics chips.

4.1 Vram

Although the RSX has 256MB of GDDR3 RAM, not all of it is usable. The last 4MB is reserved for keeping track of the RSX internal state and issued commands.

Because of the VERY slow Cell Read speed from VRAM, it is more efficient for the Cell to work in XDR and then have the RSX pull data from XDR and write to GDDR3 for output to the HDMI display. This is why extra texture lookup instructions were included in the RSX to allow loading data from XDR memory (as opposed to just the local memory).

4.2 GCM and PSGL

Developing with the official SDK leaves you with two APIs to choose from in terms of rendering. GCM and PSGL (Playstation OpengGL). GCM is specific to the hardware and is as low level as it gets. As a result what you make with it will (or should) preform somewhat better.
However, it should be noted, the PSGL is also popular due to using the OpenGL convention(OpenGL ES 1.0) - hence simple to understand and implement. The sample engine framework developed by Sony, PhyreEngine, uses PSGL as it’s rendering framework for simplicity reasons.
This is covered in greater detail in ’Tutorial 1-4 Basic Graphics’.

5. Writing code for the Ps3

stdio and stdlib All of the standard C/C++ libraries have been ported across to the PS3 - hence, it’s very easy to port across basic C/C++ code to the PS3 (e.g, sprint, fopen, write, puts).

5.1 Memory alignment

When transfering data to and from SPUs/RSX, the data being transferred has certain restrictions placed upon it. The Primary restriction is the size of the data, the other is the alignment.

Attribute	RSX	XBOX 360 Xenos	7800GTX	GTX 780	PS4 APU	Xbox One APU
Date	2005	2005	2005	2013	2013	2013
Core clock	500 MHz	500MHz	550MHz	863MHz	800MHz	853MHz
Mem Bus	128bit	128bit	256bit	384bit	256bit	256bit
Mem Clock	700 MHZ	1400 MHz	850 MHz	6000 MHz	5000 MHz	2132 MHz
Mem Bandwidth	22.4 GB/s	22.4 GB/s	54.4 GB/s	288.4 GB/s	176 GB/s	68.2 GB/s
RAM	256MB	10MB + 512MB(shared)	512MB	3GB	8GB(shared)	5GB(shared)
ROPs^1	8	8	16	48	32	16
TMUs^2	24	16	24	192	80	48
Technology	40nm	45nm	110nm	28nm	28nm	28nm

^{1}Raster Operation Units ^{2}Texture mapping units

Processor	XDR ”Main Memory”	GDDR3 RSX ”VRAM / Local Memory”
Cell Read	16.8GB/s	15.6MB
Cell Write	24.9GB/s	4GB/s
RSX Read	15.5GB/s	20.8GB/s
RSX Write	10.6GB/s	20.8GB/s

For example when transferring data to an SPU via DMA, data must be 16-byte aligned. This means that that total size AND the start address of the data, must be divisible by 16. So if you need to transfer 24 bytes of data, you must pad it with an extra 8 bytes to push it upto 32 bytes, which is divisible by 16.
Almost all of the standard datatypes are evenly aligned (1,4,8,16 bytes), but when you join them up in structs or arrays you can get odd sizes which need to be padded. Do not forget that it isn't just the size, but the starting address also, which makes things much more complicated and can lead to memory fragmentation.

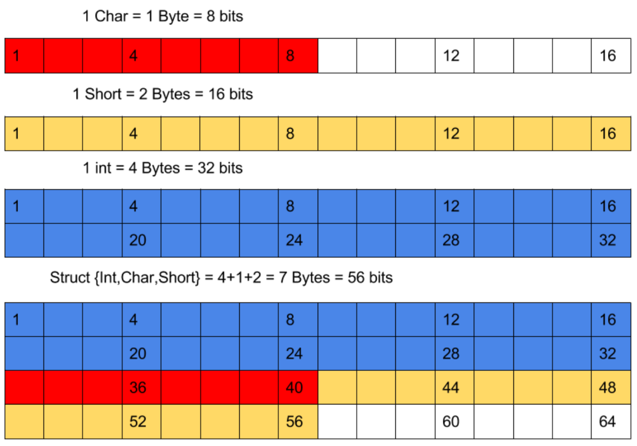
1 Char = 1 Byte = 8 bits
1
4
8
12
16
1 Short = 2 Bytes = 16 bits
1 int = 4 Bytes = 32 bits
20
24
28
32
Struct {Int,Char,Short} = 4+1+2 = 7 Bytes = 56 bits
36
40
44
48
52
56
60
64

Figure 2. Memory structure -

Malloc Malloc(N) is an old C function that allocates a block of N bytes of memory, returning a pointer to the beginning of the block. In modern C++ code, malloc() is almost never used. It was replaced by the C++ New() method, which allocates memory for a specified class, instantiates it and calls the constructor. Classes created with New() are placed on the heap and have to be Deleted() manually, Malloc is similar in this regard, memory blocks reserved by malloc have to the function free() called to release the memory.

Memalign When we need a piece of data aligned to a specific boundary (16 bytes etc..) we need to call this ancient malloc() function to give us a chunk of memory to do the alignment in. Fortunately, in the Sony stdlib library there is a function that does this and more for us.
void *memalign(size_t *boundary*, size_t *size*)
The function allocates *size* bytes and returns a pointer to the allocated memory. The memory block returned will be aligned on a multiple of *boundary*.

Memory alignment applications The standard Memalign functions will be used in the future when communicating with the SPUs. When communicating with the RSX things get a little trickier as we have to manage our own virtual memory space inside main memory. This will be covered more in future tutorials, but always keep in mind that allocated heap memory must be freed or else you will run out of the already minuscule amount of ram available to you. In the case that you do run out of ram, unlike a pc which will start spooling to the harddrive, the PS3 will just crash, and Sony will certainly not certify your game.

5.2 Debugging

Sony provide a large set of tools and libraries for debugging applications and measuring performance. With specialized hardware like the Playstation3 optimisation plays a huge part in game development, getting code to run efficiently as possible split across 6 SPUs, 1 PPU and a GPU while using the minimum possible amount or ram takes a massive amount of work. Measuring everything, literally every operation, is the key to performance, without doing so will not allow unexpected bottlenecks to be found, which is why good debugging libraries are paramount in this type of development.

Of course this only applies once the code is actually working. Debugging in it’s literal meaning and traditional sense is finding and removing bugs, and doing that on a weird and wonderful device over the network is a large step up from debugging local win32 applications in Visual studio.
With the tools provided, and the knowledge of how to use them, debugging PS3 applications is not as daunting as it would seem. The local debugger is well featured, and the libraries that run on the console side are robust and battle-tested. Debugging on the PS3 is not hard, it just has a steeper learning curve, and you will be a better software engineer at the end of it as the skills are transferable to any software project.

Break-Points If your only experience with breakpoints is clicking on a line of code in visual studio and letting it do all the work, then this segment will introduce you to some low-level assembly magic. Break points can be manually inserted into code via special assembly commands, as assembly is specific to a platform, the commands differ for different hardware/compilers and debuggers.

Listing 1. Halts on different platforms

```
//IA−32 (Intel Architecture, 32−bit)
__asm {  int 3  }
//x86/XBOX/Win32(basically a robust wrapper for int 3)
//Only supported in visual studio
__debugbreak() ;
// Halts a program running on PPC32 or PPC64 (e.g. PS3).
//Also works for ARM and in GCC/XCode
__asm volatile( ”trap” ) ;
```

Break-Point Macros If you need to stop the code at one specific location to quickly take a peak at the internal workings of code, then manually inserting a breakpoint there is an O.K solution. As with almost code design, this becomes infeasible as it scales. Wrapping a breakpoint in an IF statement is quick way of having conditional breakpoints that only fire when something goes wrong, but now you could have breakpoints sprinkled all through your code. What if you need to disable them all for a release build? They are embedded into the code so it’s not just a case of telling the debugger to not pay attention. You could wrap them all in an additional IF, or comment them all out, but doing something in code more than once means there is almost certainly a better and quicker way. There is, **Macros**

#define MYCOOLMACRO ”my cool macro”
When the preprocessor encounters this directive, it replaces any occurrence of MYCOOLMACRO in the rest of the code with ”my cool macro”. This replacement can be an expression, a statement, a block or simply anything, e.g a breakpoint command

```
// In your implementation you would do something like this:
#if PS3
  #define HALT __asm volatile( ”trap” )
#elif XBOX
  #define HALT __debugbreak();
#elif PC
  #define HALT __asm {int 3}
#else
  #error ”unknown platform’
#endif
```

So now, assuming Either PS3, XBOX or PC is defined before this somewhere (an easy thing to do), we can call HALT anywhere in the code and it will call the correct version for the platform. Now what about disabling all halts? Easy:

```
//There are better ways to do this, but this is super simple:
#if DEBUG
  #if PS3
    #define HALT......
  #endif
#endif
```

Great, we can change platform by defining one variable, and toggle breakpoints with #define DEBUG = TRUE/FALSE. Is there anything else macros can do for us here? What about conditional breakpoints, can we simplify them? Yes:

```
// call DBG_HALT on assert fail
#define ASSERT(exp){ if ( !(exp) ) {HALT;}}
// Prints the suplied string on assert fail, then call HALT
#define ASSERT_M(exp,msg){if(!(exp)){puts(msg);HALT;}}
// Calls the suplied function on assert fail, then call HALT
#define ASSERT_F(exp,func){if(!(exp)){func; HALT;}}
//Now we can do
int a = 0;
ASSERT(a > 1);
ASSERT_M ((a > 1), ”A is less than 1!”);
ASSERT_F ((a > 1 , printf(”Error : %i\n”, a) );
```

So what would all this look like in full?

Listing 2. Over engineered Macro sample

```
// −− Take a guess at the current platform

//The PS3 compiler defines either of these
#if defined(__PPU__) || defined(__SPU__)
  #define PS3

//This probably doesn’t exist, but it’s here for completness
#elif defined(_XBOX360_)
  #define XBOX360

//Windows has a lot of variants.
//In production you would want to split this up,
// as win64 might have issues with win32 stuff
#elif defined(_WIN32) || defined(_WIN64) || defined(WIN32) || ↩
        defined(__CYGWIN__) || defined(__MINGW32__)
  #define WINDOWS

//This works, but it could be either an OSX or iOS device
#elif defined(__APPLE__)
  #define MAC

//This works, but isn’t a complete solution, google __unix__
#elif defined(__linux__)
  #define LINUX

#else
  #error ”unknown platform’

#endif

#define DEBUG true

// −− Platform specific halts
#if DEBUG
  #if defined(PS3)
  #define HALT __asm volatile( ”trap” )

 #elif defined(XBOX360) || defined(WINDOWS)
  #define HALT __debugbreak();

 #elif defined(__APPLE__)
  #define HALT __builtin_trap();

 #elif defined(__linux__)
  #define HALT raise(SIGINT);

 #else
  #error ”HALT: unknown platform’

 #endif
#endif

// −− Asserts derived from HALT
#define ASSERT(exp){ if ( !(exp) ) {HALT;}}

// Prints the suplied string on assert fail, then call HALT
#define ASSERT_M(exp,msg){if(!(exp)){puts(msg);HALT;}}

// Calls the suplied function on assert fail, then call HALT
#define ASSERT_F(exp,func){if(!(exp)){func; HALT;}}

int main(void)
{
  //Try running an assert
 int a = 0;
 DBG_ASSERT(a > 1);
}
```

In reality you will probably just do something like this if you are only ever going to target one platform.

```
#define HALT { __asm volatile( ”trap” );}
```

But where is the fun in that?

5.3 A simple PPU program

So after the largest preamble ever, let’s get down to writing some real code. Outputting something to a screen is something that will happen in another tutorial as this one has already gone on too long. Instead of outputting video, let’s do something much more fun: blink some LEDS.
The devkit has 8 LEDS(GPO) and 8 Input switches(GPI) on the front of the machine. Only 4 [0,1,2,3] of each are usable to us.

Listing 3. LED.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/return_code.h>
4 #include <sys/gpio.h>
5 #include <sys/timer.h>
6
7 #define HALT { __asm volatile( ”trap” ); }
8 #define ASSERT(exp) { if ( !(exp) ) {HALT;} }
9 #define ASSERT_MSG( exp, smsg ) {{ ASSERT(exp); } }
10
11 #define LED_DELAY_TIME  1
12
13 void change_led(uint64_t led_value)
14 {
15  uint64_t err =
16   sys_gpio_set (SYS_GPIO_LED_DEVICE_ID,
17        SYS_GPIO_LED_USER_AVAILABLE_BITS,
18        led_value);
19  ASSERT_MSG((err == CELL_OK),
20      ”sys_gpio_set error”
21  );
22 }
23
24 int main(void)
25 {
26  puts(”Program Starting!\n”);
27
28  //Do a binary count 10 times
29  int a = 0;
30  int rounds = 0;
31  while(rounds < 10)
32  {
33   a++;
34   rounds++;
35   if (a > 15){
36    a = 0;
37   }
38   change_led(a);
39   sys_timer_sleep(LED_DELAY_TIME);
40  }
41
42  uint64_t dip_switch;
43  int err;
44  while(true)
45  {
46   //Read dip switches
47   err = sys_gpio_get(
48        SYS_GPIO_DIP_SWITCH_DEVICE_ID,
49        &dip_switch
50      );
51   ASSERT_MSG((err == CELL_OK), ”Can’t read dip switches”);
52   dip_switch = dip_switch &
53    SYS_GPIO_DIP_SWITCH_USER_AVAILABLE_BITS;
54   //Control LEDs
55   change_led(dip_switch);
56   sys_timer_sleep(LED_DELAY_TIME);
57  }
58
59  puts(”Program Quitting!\n”);
60  return EXIT_SUCCESS;
61 }
```

The Leds are set with one command, sys_gpio_set() on line 16. The first parameter is always the same, and is stored as SYS_GPIO_LED_DEVICE_ID. The second parameter is a mask specifying which bits to change, and the last parameter is the important one, which LEDS are on and off. The 4 LEDs are in are set-up like binary, 1111(15) = all on, 0000(0) = all off, 1010(5) Lights 0 and 2 on. The input switches are exactly the same so the switch input values can be directly mapped to the LED output values.

6. Conclusion

The LED program is simple and doesn’t really serve much of a purpose other than to show that programming on the PS3 can be really simple. Obviously it will get very complex later on, but it’s important not to be intimidated and to remember that it’s just a simple computer. The limitations of the hardware is why understanding the architecture is really useful, on a normal pc you rarely need to care about anything other than your code and simple factors like filesize and load times.
Learning where the boundaries of the PS3 are before you hit them results in better designed code from the start, makes your life easier in the future, and makes you a better programmer.

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] James Leiterman. Vector games math processors (wordware game math library) (isbn:978-1556229213), 2011. 1

[2] Syd Logan. Cross-platform development in c++: Building mac os x, linux, and windows applications (isbn:978-0321246424), 2007. 1

[3] Matthew Scarpino. Programming the cell processor: For games, graphics, and computational proccessing (isbn: 978-0136008866), 2011. 1

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