EXHIBIT MM
(VOL 3)

Programming high-performance applications on the Cell BE processor, Part 5: Programming the SPU in C/C++

Use the language extensions to power up your applications

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: In Part 5 of the Programming high-performance applications on the Cell BE processor series, apply your knowledge of the synergistic processing unit (SPU) to programming the Cell Broadband Engine™ (Cell BE) processor in C/C++. Learn how to use the vector extensions, direct the compiler to do branch prediction, and perform DMA transfers in C/C++.

Date: 20 Mar 2007
Level: Intermediate
Also available in: Chinese Russian Japanese

Activity: 13720 views
Comments: 0 (Add comments)

★★★★★ Average rating (based on 28 votes)

Previous discussions about the SPU have focused on the SPU's assembly language to help you get to know the processor intimately. Now I will switch to C/C++ so that you can see how to let the compiler do a large amount of the work for you. To utilize the SPU C/C++ language extensions, the header file `spu_intrinsics.h` must be included at the beginning of your code.

Vector basics on the SPU

The primary difference between vector processors and non-vector processors is that vector processors have large registers which allow them to store multiple values (called *elements*) of the same data type and process them with the same operation at once. On vector processors a register is treated both as a single unit and as multiple units. To represent this concept in C/C++, a `vector` keyword has been added to the language, which takes a primitive data type and uses it across a whole register. For instance, `vector unsigned int myvec;` creates a four-integer vector where the elements are to be loaded, processed, and stored altogether, and the variable `myvec` refers to all four of them simultaneously. The `signed/unsigned` keyword is required for non-floating point declarations. Vector constants are created by putting the type of vector in parentheses followed by the contents of the vector in curly braces. For instance, you can assign values to a vector named `myvec` like this:

```
vector unsigned int myvec = (vector unsigned int){1, 2, 3, 4};
```

In addition to direct assignment, there are four main primitives that are used to go between scalar and vector data: `spu_insert`, `spu_extract`, `spu_promote`, and `spu_splats`. `spu_insert` is used to put a scalar value into a specific element of a vector. `spu_insert(5, myvec, 0)` returns a copy of the

BAKER 0000123

vector `myvec` with the first element (element 0) of the new vector set to 5. `spu_extract` pulls out a specific element from a vector and returns it as a scalar. `spu_extract(myvec, 0)` returns the first element of `myvec` as a scalar. `spu_promote` converts a value to a vector, but only defines one element. The type of vector depends on the type of value promoted. `spu_promote((unsigned int)5, 1)` creates a vector of `unsigned ints` with 5 in the second element (element 1), and the remaining elements undefined. `spu_splats` works like `spu_promote`, except that it copies the value to *all* elements of the vector. `spu_splats((unsigned int)5)` creates a vector of `unsigned ints` with each element having the value 5.

It is tempting to think of vectors as short arrays, but in fact they act differently in several respects. Vectors are treated essentially as scalar values, while arrays are manipulated as references. For instance, `spu_insert` *does not modify the contents of the vector*. Instead, it returns a brand new copy of the vector with the inserted element. It is an expression that results in a value, not a modification to the value itself. For instance, just as `myvar + 1` gives back a new value instead of modifying `myvar`, `spu_insert(1, myvec, 0)` does not modify `myvec`, but instead returns a new vector value that is equivalent with `myvec` but has the first element set to 1.

Here is a short program using these ideas (enter as `vec_test.c`):

Listing 1. Program to introduce SPU C/C++ language extensions

```
#include <spu_intrinsics.h>

void print_vector(char *var, vector unsigned int val) {
        printf("Vector %s is: {%d, %d, %d, %d}\n", var, spu_extract(val, 0),
         spu_extract(val, 1), spu_extract(val, 2), spu_extract(val, 3));
}

int main() {
        /* Create four vectors */
        vector unsigned int a = (vector unsigned int){1, 2, 3, 4};
        vector unsigned int b;
        vector unsigned int c;
        vector unsigned int d;

        /* b is identical to a, but the last element is changed to 9 */
        b = spu_insert(9, a, 3);

        /* c has all four values set to 20 */
        c = spu_splats((unsigned int) 20);

        /* d has the second value set to to 5, and the others are garbage */
        /* (in this case they will all be set to 5, but that should not be relied u
        d = spu_promote((unsigned int)5, 1);

        /* Show Results */
        print_vector("a", a);
        print_vector("b", b);
        print_vector("c", c);
        print_vector("d", d);

        return 0;
}
```

BAKER 0000124
9/20/2010

To compile and run the program under elfspe, simply do:

```
spu-gcc vec_test.c -o vec_test
./vec_test
```

Vector intrinsics

The C/C++ language extensions include data types and intrinsics that give the programmer nearly full access to the SPU's assembly language instructions. However, many intrinsics are provided which greatly simplify the SPU's assembly language by coalescing many similar instructions into one intrinsic. Instructions that differ only on the type of operand (such as `a`, `ai`, `ah`, `ahi`, `fa`, and `dfa` for addition) are represented by a single C/C++ intrinsic which selects the proper instruction based on the type of the operand. For addition, `spu_add`, when given two `vector unsigned ints` as parameters, will generate the `a` (32-bit add) instruction. However, if given two `vector floats` as parameters, it will generate the `fa` (float add) instruction. Note that the intrinsics generally have the same limitations as their corresponding assembly language instructions. However, in cases where an immediate value is too large for the appropriate immediate-mode instruction, the compiler will promote the immediate value to a vector and do the corresponding vector/vector operation. For instance, `spu_add(myvec, 2)` generates an `ai` (add immediate) instruction, while `spu_add(myvec, 2000)` first loads the `2000` into its own vector using `il` and then performs the `a` (add) instruction.

The order of operands in the intrinsics is essentially the same as those of the assembly language instruction except that the first operand (which holds the destination register in assembly language) is not specified in C/C++, but instead is used as the return value for the function. The compiler supplies the appropriate operand in the assembly language code it generates.

Here are some of the more common SPU intrinsics (types are not given as most of them are polymorphic):

- `spu_add(val1, val2)`
  Adds each element of `val1` to the corresponding element of `val2`. If `val2` is a non-vector value, it adds the value to each element of `val1`.
- `spu_sub(val1, val2)`
  Subtracts each element of `val2` from the corresponding element of `val1`. If `val1` is a non-vector value, then `val1` is replicated across a vector, and then `val2` is subtracted from it.
- `spu_mul(val1, val2)`
  Because the multiplication instructions operate so differently, the SPU intrinsics do not coalesce them as much they do for other operations. `spu_mul` handles floating point multiplication (single and double precision). The result is a vector where each element is the result of multiplying the corresponding elements of `val1` and `val2` together.
- `spu_and(val1, val2)`, `spu_or(val1, val2)`, `spu_not(val)`, `spu_xor(val1, val2)`, `spu_nor(val1, val2)`, `spu_nand(val1, val2)`, `spu_eqv(val1, val2)`
  Boolean operations operate bit-by-bit, so the type of operands the boolean operations receive is not relevant except for determining the type of value they will return. `spu_eqv` is a bitwise equivalency operation, not a per-element equivalency operation.

- `spu_rl(val, count)`, `spu_sl(val, count)`
  `spu_rl` rotates each element of `val` left by the number of bits specified in the corresponding element of `count`. Bits rotated off the end are rotated back in on the right. If `count` is a scalar value, then it is used as the count for all elements of `val`. `spu_sl` operates the same way, but performs a shift instead of a rotate.
- `spu_rlmask(val, count)`, `spu_rlmaska`, `spu_rlmaskqw(val, count)`, `spu_rlmaskqwbyte (val, count)`
  These are very confusingly named operations. They are named "rotate left and mask," but they are actually performing *right shifts* (they are *implemented* by a combination of left shifts and masks, but the programming interface is for right shifts). `spu_rlmask` and `spu_rlmaska` shifts each element of `val` to the right by the number of bits in the corresponding element of `count` (or the value of `count` if `count` is a scalar). `spu_rlmaska` replicates the sign bit as bits are shifted in. `spu_rlmaskqw` operates on the whole quadword at a time, but only up to 7 bits (it performs a modulus on `count` to put it in the proper range). `spu_rlmaskqwbyte` works similarly, except that `count` is the number of bytes instead of bits, and `count` is modulus 16 instead of 8.
- `spu_cmpgt(val1, val2)`, `spu_cmpeq(val1, val2)`
  These instructions perform element-by-element comparisons of their two operands. The results are stored as all ones (for true) and all zeros (for false) in the resulting vector in the corresponding element. `spu_cmpgt` performs a greater-than comparison while `spu_cmpeq` performs an equality comparison.
- `spu_sel(val1, val2, conditional)`
  This corresponds to the `selb` assembly language instruction. The instruction itself is bit-based, so all types use the same underlying instruction. However, the intrinsic operation returns a value of the same type as the operands. As in assembly language, `spu_sel` looks at each bit in `conditional`. If the bit is zero, the corresponding bit in the result is selected from the corresponding bit in `val1`; otherwise it is selected from the corresponding bit in `val2`.
- `spu_shuffle(val1, val2, pattern)`
  This is an interesting instruction which allows you to rearrange the bytes in `val1` and `val2` according to a pattern, specified in `pattern`. The instruction goes through each byte in `pattern`, and if the byte starts with the bits `0b10`, the corresponding byte in the result is set to `0x00`; if the byte starts with the bits `0b110`, the corresponding byte in the result is set to `0xff`; if the byte starts with the bits `0b111`, the corresponding byte in the result is set to `0x80`; finally (and most importantly), if none of the previous are true, the last five bits of the pattern byte are used to choose which byte from `val1` or `val2` should be taken as the value for the current byte. The two values are concatenated, and the five-bit value is used as the byte index of the concatenated value. This is used for inserting elements into vectors as well as performing fast table lookups.

All of the instructions that are prefixed with `spu_` will try to find the best instruction match based on the types of operands. However, not all vector types are supported by all instructions -- it is based on the availability of assembly language instructions to handle it. In addition, if you want a specific instruction rather than having the compiler choose one, you can perform almost any non-branching instruction with the *specific intrinsics*. All specific intrinsics take the form `si_assemblyinstructionname` where `assemblyinstructionname` is the name of the assembly language instruction as defined in the SPU Assembly Language Specification. So, `si_a(a, b)` forces the instruction `a` to be used for addition. All operands to specific intrinsics are cast to a special type called `qword`, which is essentially an opaque register value type. The return value from specific intrinsics are also `qwords`, which can then be cast into whatever vector type you wish.

Using the intrinsics

Now let's look at how to do the uppercase conversion function using C/C++ rather than assembly language. The basic steps for converting a single vector are:

1. Convert all values using the uppercase conversion.
2. Do a vector comparison of all bytes to see if they are between `'a'` and `'z'`.
3. Use the comparison to choose between the converted and unconverted values using the select instruction.

In addition, to help better schedule instructions, the assembly language version performed several of these conversions simultaneously. In C/C++, you can call an inline function multiple times, and let the compiler take care of scheduling it appropriately. This doesn't mean that your knowledge of instruction scheduling is useless, but rather because you know how instruction scheduling works, you are able to give the compiler better raw material to work with. If you did not know that instruction scheduling improves your code, and that instruction scheduling can be helped by unrolling your loops, then you would not be able to help the compiler optimize your code.

So here is the C/C++ version of the `convert_buffer_to_upper` function (enter as `convert_buffer_c.c` in the same directory as the files from the previous articles -- you will need them to compile the full application):

Listing 2. Uppercase conversion in C/C++

```
#include <spu_intrinsics.h>

unsigned char conversion_value = 'a' - 'A';

inline vec_uchar16 convert_vec_to_upper(vec_uchar16 values) {
        /* Process all characters */
        vec_uchar16 processed_values = spu_absd(values, spu_splats(conversion_value
        /* Check to see which ones need processing (those between 'a' and 'z')*/
        vec_uchar16 should_be_processed = spu_xor(spu_cmpgt(values, 'a'-1),
        spu_cmpgt(values, 'z'));
        /* Use should_be_processed to select between the original and processed val
        return spu_sel(values, processed_values, should_be_processed);
}

void convert_buffer_to_upper(vec_uchar16 *buffer, int buffer_size) {
        /* Find end of buffer (must be casted first because size is bytes) */
        vec_uchar16 *buffer_end = (vec_uchar16 *)((char *)buffer + buffer_size);

        while(__builtin_expect(buffer < buffer_end, 1)) {
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
                *buffer = convert_vec_to_upper(*buffer);
                buffer++;
        }
}
```

To compile and run, simply do:

```
spu-gcc convert_buffer_c.c convert_driver.s dma_utils.s -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

As you probably noticed, this program uses slightly different notation for vector type names than used previously. The SPU intrinsics documentation (see Resources) defines simplified vector type names starting with `vec_`. For integer types, the next character is `u` or `s` for signed/unsigned types. After that is the name of the basic type being used (`char`, `int`, `float`, and so on). Finally, at the end is the number of elements of that type which are in the vector. `vec_uchar16`, for instance, is a 16-element vector of `unsigned chars`, and `vec_float4` is a 4-element vector of `floats`. This notation greatly simplifies the typing involved.

When computing the `buffer_end` the program did some casting gymnastics. Because `size` was in bytes, I had to convert the pointer to a `char *` so that when I added the size, it would move by bytes rather than by quadwords. Vector pointers, since the value they point to is 16-bytes long, move forward in increments of 16 bytes, while `char` pointers move forward in single-byte increments. That is why `buffer++` works -- it is incrementing by a single vector length, which is 16 bytes.

Another interesting feature of the C/C++ version is `__builtin_expect` which helps the compiler do branch hinting. You cannot do branch hinting directly in C/C++ because you have neither the address of the branch nor the target. Therefore, you instead provide hints to the compiler, which can then generate appropriate branch hints. `__builtin_expect(buffer < buffer_end, 1)` generates branching code based off of the first argument, `buffer < buffer_end`, but produces branch hints based off of the second argument, 1. It tells the compiler to generate hints that expect the value of `buffer < buffer_end` to be 1.

Now, there are two compilers currently available for SPU programming, and, as one might expect, they excel in different areas. GCC, for instance, does a fantastic job of interleaving the instructions between invocations of `convert_vec_to_upper` so that instruction latency is minimized. However, in this particular program, `__builtin_expect` gives us almost no help at all. The IBM XLC compiler, on the other hand, is the opposite. It does not interleave the instructions between invocations of `convert_vec_to_upper` at all, but structures the loop so that the branch hint has a maximum effect, and in fact was able to guess the branch hint without it being supplied. Unsurprisingly, neither compiler does nearly as well as the hand-coded assembly language version from the previous article, but for this program XLC outperformed GCC. Code that was compiled without any optimization flags resulted in code that was approximately *five times slower*, so be sure to always compile with `-O2` or `-O3`.

Composite intrinsics and MFC programming

The composite intrinsics are those that compile to multiple instructions. The composite intrinsics encapsulate common usage patterns on the SPE to simplify its programming. The two most important composite intrinsics are `spu_mfcdma64` and `spu_mfcstat`. `spu_mfcdma64` is almost exactly like the dma_transfer function I wrote and used in previous articles, except that the high and low parts of the effective address are split between two 32-bit parameters (dma_transfer used one 64-bit parameter for the effective address).

spu_mfcdma64 takes six parameters:

1. the local store address for the transfer
2. the high-order 32-bits of the effective address
3. the low-order 32-bits of the effective address
4. the size of the transfer
5. a "tag" to give the transfer
6. the DMA command to give

Often times you will have the effective address as a single 64-bit value. To separate it out into parts, use mfc_ea2h to extract the higher-order bits and mfc_ea2l to extract the lower-order bits. The tag is a number designated by the programmer between 0 and 31 used to identify a transfer or for a group of transfers for status queries and sequencing operations. The DMA command can take a range of values (see Resources for information on where to find the ones not listed here). DMA transfers are called PUTs if they transfer from the SPU local store to the system memory, and GETs if they go in the other direction. These DMA command names are prefixed with either MFC_PUT or MFC_GET, respectively. Then, MFC commands either operate individually or on a list. If the DMA command is a list command, the DMA command name has an L appended to it (see Resources for more information on DMA list commands). The DMA command can also have certain levels of synchronization applied to it. For barrier synchronization add a B, for fence synchronization add an F, and for no synchronization you do not need to add anything. Finally, all DMA command names have a _CMD suffix. So, the command name for a single transfer from the local store to system memory using fence synchronization would be MFC_PUTF_CMD.

By default DMA commands on the SPE's MFC are totally unordered -- the MFC may process them in any order that it wishes. However, tags, fences, and barriers can be used to force ordering constraints on MFC DMA transfers. A *fence* establishes the constraint that a given DMA transfer only execute *after* all previous commands using the same tag have completed. A *barrier* establishes the constraint that a given DMA transfer only execute *after* all previous commands using the same tag have completed (like a fence), but also that they must execute *before* all subsequent commands using the same tag.

Here are some examples of spu_mfcdma64:

Listing 3. Using spu_mfcdma64

```
typedef unsigned long long uint64;
typedef unsigned long uint32;
uint64 ea1, ea2, ea3, ea4, ea5; /* assume each of these have sensible values */
void *ls1, *ls2, *ls3, *ls4; /* assume each of these have sensible values */
uint32 sz1, sz2, sz3, sz4; /* assume each of these have sensible values */
int tag = 3; /* Arbitrary value, but needs to be the same for all
synchronized transfers */

/* Transfer 1: System Storage -> Local Store, no ordering specified */
spu_mfcdma64(ls1, mfc_ea2h(ea1), mfc_ea2l(ea1), sz1, tag, MFC_GET_CMD);
```

BAKER 0000129

```
/* Transfer 2: Local Storage -> System Storage, must perform after previous transfe
spu_mfcdma64(ls2, mfc_ea2h(ea2), mfc_ea2l(ea2), sz2, tag, MFC_PUTF_CMD);

/* Transfer 3: Local Storage -> System Storage, no ordering specified */
spu_mfcdma64(ls3, mfc_ea2h(ea3), mfc_ea2l(ea3), sz3, tag, MFC_PUT_CMD);

/* Transfer 4: Local Storage -> System Storage, must be synchronized */
spu_mfcdma64(ls4, mfc_ea2h(ea4), mfc_ea2l(ea4), sz4, tag, MFC_PUTB_CMD);

/* Transfer 5: System Storage -> Local Storage, no ordering specified */
spu_mfcdma64(ls4, mfc_ea2h(ea5), mfc_ea2l(ea5), sz4, tag, MFC_GET_CMD);
```

The above example has several possible orderings. All of the following are possibilities:

- 1, 2, 3, 4, 5
- 3, 1, 2, 4, 5
- 1, 3, 2, 4, 5

Because transfer 2 only uses a fence and transfer 3 doesn't specify any ordering at all, transfer 3 is free to float anywhere before the barrier (transfer 4). The only requirement for the first three transfers is that transfer 2 must be performed after transfer 1. Transfer 4, however, requires full synchronization of transfers before and after it.

Take a closer look at transfers 4 and 5. This is a useful idiom to take note of -- save and reload. If you are processing system memory data a piece at a time into local store and storing it back into system memory, you can queue up a save and a load at the same time, using a fence or barrier to order them. This puts all of the transferring logic into the MFC, and leaves your program free to do other computational tasks while the buffer waits for new data. We will make use of this in the next article when we talk about double buffering.

`spu_mfcdma64` is quite a handy tool, but it is a little tedious, especially when you have to keep on using `mfc_ea2h` and `mfc_ea2l` to convert your addresses. Therefore, the specification also provides a number of utility functions to lessen the amount of redundant typing necessary. The `mfc_` class of functions all take the same parameters as the `spu_mfcdma64` function, except that the effective address is a single 64-bit parameter, and the DMA command is encoded into the function name. It also takes two extra parameters, the *transfer class identifier* and the *replacement class identifier*. Both of these can be safely set to zero in non-realtime applications (see Resources for references to further information on these two fields). Therefore, transfer 2 above can be rewritten as:

```
mfc_putf(ls2, ea2, sz2, tag, 0, 0);
```

Tags are useful not just for synchronizing data transfers, but also for checking on the status of transfers. On the SPE, there is a tag mask channel which is used to specify which tags are currently used for status checks, a channel which is used to issue status requests, and another channel to read the channel status back. Although these are pretty simple operations anyway, the specification gives special methods for performing these operations as well. `mfc_write_tag_mask` takes a 32-bit integer and uses it as a channel mask for future status updates. In the mask, set the bit position of each tag that you want to check the status of to 1. So, to check the status of channel 2 and 4, you would use `mfc_write_tag_mask(20)`, or, to make it more readable, you can do `mfc_write_tag_mask(1<<2 |`

BAKER 0000130

`1<<4);`. To actually perform the status update, you have to pick a status command, and send it using `spu_mfcstat(unsigned int command)`. The commands are:

- `MFC_TAG_UPDATE_IMMEDIATE`
  This command causes the SPE to immediately return with the status of the DMA channels. Each channel which was specified in the channel mask will be set to 1 if there are no remaining commands in the queue with that tag (in other words, all operations that may have been previously active, are completed), and set to 0 if there are commands remaining in the queue.
- `MFC_TAG_UPDATE_ANY`
  This command causes the SPE to wait until at least one tag specified in the tag mask has no remaining commands before returning, then returns the status of the DMA channels that were specified in the tag mask.
- `MFC_TAG_UPDATE_ALL`
  This command causes the SPE to wait until all tags specified in the tag mask have no remaining commands before returning. The return value will be 0.

To use these constants, you need to include `spu_mfcio.h`.

Using `spu_mfcstat` allows you to both check on the status of DMA requests and wait for them. Using `MFC_TAG_UPDATE_ANY` allows you to issue multiple DMA requests, let the MFC process them in whatever order it thinks is best, and then your code can respond based on the order that the MFC processes them.

Example MFC program

Now I'll apply this knowledge of the MFC composite intrinsics to the uppercase conversion program. Earlier in the article I rewrote the main conversion function in C, and now I am going to rewrite the main loop in C. The new code is fairly straightforward (enter as `convert_driver_c.c`):

Listing 4. Uppercase conversion MFC transfer code

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
typedef unsigned long long uint64;

#define CONVERSION_BUFFER_SIZE 16384
#define DMA_TAG 0

void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);

char conversion_buffer[CONVERSION_BUFFER_SIZE];

typedef struct {
        int length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;

int main(uint64 spe_id, uint64 conversion_info_ea) {
        conversion_structure conversion_info; /* Information about the data from th

        /* We are only using one tag in this program */
        mfc_write_tag_mask(1<<DMA_TAG);
```

BAKER 0000131

```
        /* Grab the conversion information */
        mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), DMA_
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

        /* Get the actual data */
        mfc_get(conversion_buffer, conversion_info.data, conversion_info.length, DM
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        /* Perform the conversion */
        convert_buffer_to_upper(conversion_buffer, conversion_info.length);

        /* Put the data back into system storage */
        mfc_put(conversion_buffer, conversion_info.data, conversion_info.length, DM
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

To compile and run, simply do:

```
spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

This implementation in C follows the same basic structure as the original code, except that it's more readable to human beings, which, incidentally, makes it easier to revise and expand. For instance, one of the problems with the original code is that it is limited to the size of a DMA transfer. What if you wanted to remove that limitation? You could simply wrap the whole thing in a loop, and keep moving data a piece at a time until the whole string has been processed. Here's the revised code to do this:

Listing 5. Looping in the MFC transfer code

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h> /* constant declarations for the MFC */
typedef unsigned long long uint64;
typedef unsigned int uint32;

/* Renamed CONVERSION_BUFFER_SIZE to MAX_TRANSFER_SIZE because it is now
primarily used to limit the size of DMA transfers */
#define MAX_TRANSFER_SIZE 16384

void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);

char conversion_buffer[MAX_TRANSFER_SIZE];

typedef struct {
        uint32 length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;

int main(uint64 spe_id, uint64 conversion_info_ea) {
        conversion_structure conversion_info; /* Information about the data from th

        /* New variables to keep track of where we are in the data */
        uint32 remaining_data; /* How much data is left in the whole string */
        uint64 current_ea_pointer; /* Where we are in system memory */
        uint32 current_transfer_size; /* How big the current transfer is (may be sm
        than MAX_TRANSFER_SIZE) */

        /* We are only using one tag in this program */
        mfc_write_tag_mask(1<<0);

        /* Grab the conversion information */
        mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), 0, 0
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

        /* Setup the loop */
        remaining_data = conversion_info.length;
        current_ea_pointer = conversion_info.data;

        while(remaining_data > 0) {
                /* Determine how much data is left to transfer */
                if(remaining_data < MAX_TRANSFER_SIZE)
                        current_transfer_size = remaining_data;
                else
                        current_transfer_size = MAX_TRANSFER_SIZE;

                /* Get the actual data */
                mfc_getb(conversion_buffer, current_ea_pointer, current_transfer_si
                spu_mfcstat(MFC_TAG_UPDATE_ALL);

                /* Perform the conversion */
                convert_buffer_to_upper(conversion_buffer, current_transfer_size);

                /* Put the data back into system storage */
                mfc_putb(conversion_buffer, current_ea_pointer, current_transfer_si

                /* Advance to the next segment of data */
                remaining_data -= current_transfer_size;
                current_ea_pointer += current_transfer_size;
        }
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Compile and run using the same commands as you used in the previous example:

```
spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

So now you have just expanded the size of the data you can process to 4 gigabytes, though you could easily go beyond that by making the data size variables 64-bit instead of 32-bit. Notice that you don't explicitly code to ask the MFC to wait for your PUT to complete before you re-issue the GET. This is because you are using barriers with your transfers, and you are using the same DMA tag for them.

BAKER 0000133

This forces the transfers to be serialized by the MFC itself, so it will always wait until the current conversion is finished being PUT into system storage before GETting more data into the buffer. Just remember to wait for the completion at the end (notice the `spu_mfcstat` outside the loop), or else your last bit of data may not finish transferring before it is used in the program!

Another thing to be careful of when programming in C is to always make sure that you give function prototypes. It is real easy to accidentally mix up 32-bit and 64-bit values. On the PPE that isn't so bad, as the value is merely truncated or expanded. But in the SPE, if the prototype is wrong, the preferred slot for 32-bit and 64-bit values is offset in such a way that conversion between the two must be handled explicitly.

Helpful tips for C language SPE programming

Here are some tips to keep in mind when building SPE applications in C:

- Vectors *can* be cast between vectors of other types, and back-and-forth between the vector types and the special `quad` type, but *none of these casts perform any data conversion.* If you need to convert between types, use an appropriate SPU intrinsic.
- Vector and non-vector pointers *can* be cast between each other, but when converting from a scalar pointer to a vector pointer *it is the programmer's responsibility to be sure that the pointer is quadword-aligned.*
- Declared vectors are always quadword-aligned when allocated.
- Remember that DMA transfers of 16 bytes or more *must be in 16-byte multiples and aligned to 16-byte boundaries* on both the SPE and the PPE. Transfers smaller than that must be a power of two and be naturally aligned. Optimal transfers are multiples of 128 bytes that are on 128-byte boundaries.
- If you are not sure about the alignment of data on the PPE, use `memalign` or `posix_memalign` to allocate an aligned pointer from the heap, and use `memcpy` or an equivalent to move the data to the aligned area.
- Always compile with `-Wall` and *especially pay attention to missing prototype messages.* Incorrectly implied prototypes (especially between 32- and 64-bit types) can lead to bizarre error conditions.
- Always store effective addresses as `unsigned long long`s, on both the PPE and the SPE. This way they can be treated in a unified fashion on the SPE and on the PPE, whether the PPE code is compiled for 32-bit or 64-bit execution.
- Avoid integer multiplies (especially 32-bit multiplies) on the SPE. It takes five instructions to perform the multiply. If you must multiply, cast to an `unsigned short` before multiplying.
- In scalar code on the SPE, declaring scalar values as vectors and vector pointers (even if you aren't using them as vectors) can speed up code because it doesn't have to do unaligned loads and stores.
- Be aware that on the SPE, `float`s and `double`s are implemented differently, and round differently as well. `float`s in particular deviate from the C99 standard. The next article will cover these further.

Conclusion

The intrinsics available for C allow programmers to make the best mix of C and assembly language knowledge. The SPU intrinsics allow programs to freely switch among high- and low-level code, but all within the semantic framework of the C language.

BAKER 0000134

The next article applies this knowledge into a real-world numerical application.

Resources

Learn

- See the other parts in the Programming high-performance applications on the Cell BE processor series.
- The full set of intrinsics is documented in the PPU & SPU C/C++ Language Extension Specification.
- Another (more extended) tutorial resource for Cell BE programming on both the SPE and the PPE is the official Cell BE Programming Tutorial.
- For a complete list of available DMA commands on the MFC, see chapter 7 of the Cell BE Architecture Specification (1.01) and pages 508-510 of the Cell BE Programming Handbook (1.0).
- For more information on DMA list commands, see pages 51-62, 124-125, 129-130, and 157-158 of the Cell BE Architecture Specification (1.01) and pages 73, 459-460, 509-510, and 527-530 of the Cell BE Programming Handbook (1.0).
- The transfer class ID and replacement class ID fields for MFC operations is described on pages 78 and 114 of the Cell BE Architecture Specification (1.01) and pages 155-158, 455-456, and 513-515 of the Cell BE Programming Handbook (1.0).
- Find all Cell BE-related articles, discussion forums, downloads, and more at the IBM developerWorks Cell Broadband Engine resource center: your definitive resource for all things Cell BE.
- Stay abreast of all things Cell BE: subscribe to IBM microNews.

Get products and technologies

- Get Cell BE: Contact IBM about custom Cell-based or custom-processor based solutions.
- Get the alphaWorksCell Broadband Engine downloads -- including the IBM Full System Simulator, support libraries, toolchains, source code for libraries, and samples.

Discuss

- Take part in the IBM developerWorks Power Architecture Cell Broadband Engine discussion forum.

About the author

Jonathan Bartlett is the author of the book *Programming from the Ground Up* , an introduction to programming using Linux assembly language. He is the lead developer at New Media Worx, responsible for developing Web, video, kiosk, and desktop applications for clients.

PS3 fab-to-lab, Part 1: Build Linux lab equipment from a Sony PLAYSTATION 3

Introducing how to generate and analyze signals on your Cell/B.E.-based spectrum analyzer

Lewin Edwards (sysadm@zws.com), Design Engineer, Freelance

Summary: How do you take the Cell Broadband Engine™ (Cell/B.E.) processor from an off-the-shelf Sony® PLAYSTATION® 3 (PS3) and use it to construct a piece of Linux®-based laboratory equipment (in essence, taking the Cell/B.E. from fab to hab to lab)? In this series, Lewin Edwards shows you how to go from game console to simple audio-bandwidth spectrum analyzer and function generator. First up, uncover the design intent of the project and then make a close inspection of the details of the user interface implementation as you start a journey to generate and analyze signals on the Cell/B.E. processor.

Date: 15 May 2007
Level: Intermediate
Also available in: Japanese

Activity: 16964 views
Comments: 1 (View or add comments)

★★★★★ Average rating (based on 35 votes)

The Cell Broadband Engine (Cell/B.E.) processor has attracted a lot of fashionable attention for applications involving game playing and network data processing. However, there are many other, arguably more entertaining uses for this technology.

In this series of articles I will be using a Cell/B.E. processor — resident within an off-the-shelf PLAYSTATION 3 (PS3) — to build a Linux-hosted piece of laboratory equipment, namely a simple audio-bandwidth spectrum analyzer and function generator.

In this first article, I'll describe the design intent of the project and go into details of the user interface implementation.

The setup

The specific hardware and software combination I'm using is a 60GB PS3 running Yellow Dog Linux 5.0 (YDL). I am using a standard NTSC television set as my output device and I also added a vanilla USB keyboard and mouse and a Griffin iMic to the system (more on that peripheral later).

Everything I am doing here should be compatible with both the 20GB version of the PS3 and almost any other compatible Linux distribution (the only other distro known to work at the time of writing is Fedora Core 5 for PowerPC®). The reason I chose YDL is purely because it offers the path of least resistance — Terra Soft has assembled and certified it for use on the PS3, *and* it includes everything you'll need to get up and running with the development process and the hardware we intend to use.

If you're more comfortable with a different PowerPC Linux distribution, then feel free to use it — but you might need to download some additional components not mentioned explicitly in this text.

BAKER 0000137

Similarly, I chose the iMic because it is known to be well-supported by PowerPC Linux — again, you're free to use any USB audio input device you wish, but locating drivers is left as an exercise for the reader. As a sidebar, note that you do not need to buy a PS3 game controller; you can control the Sony operating system with a USB keyboard (at least enough to get Linux installed, after which you'll never need to interact with GameOS again).

Please refer to the Resources section for links to the products mentioned. If you plan to follow along and build and test the example code, you should also begin by reading and following the instructions in Jonathan Bartlett's article describing how to install Linux on the PS3 (see Resources). This is a relatively well-documented and fairly simple Linux install compared to some I could name, but it's not quite as simple as just inserting a DVD and clicking "Go." Some hand-holding is definitely necessary.

If you don't want to invest in a PS3, you can build most of the code inside the Cell simulator, but there probably isn't a lot of point unless you're also willing to write a front end to simulate the audio input/output devices (perhaps using .WAV files) and graphical display.

The rationale

Now, you might ask: What is the rationale behind using Cell/B.E. processors in an application of this type? More often than not these days, engineers need to be able to control test equipment from a PC and get all test data acquired back into the computer to be imported into analysis software such as Mathcad or Matlab. Given the increasing complexity and PC-centric nature of instrumentation, a general trend for practically all modern standalone laboratory instruments is that they are based around embedded PCs with some custom magic bolted onto the front end.

For example, a digital oscilloscope might contain a fairly low-end processor running a general-purpose operating system -- this processor will handle the user interface, networking, mass-storage and so forth. One or more digital signal processors (DSPs) coupled to fast analog-digital converters (ADCs) perform the signal acquisition and pre-processing, trigger generation, and so forth.

Functionality in a single package

With a Cell/B.E. processor, you can wrap most of that functionality up into a single chip -- in the base architecture, you already have a frisky main processor (the PPE) and eight DSP-like coprocessors (the SPEs). Moreover, the chip includes all the necessary plumbing to move bite-size chunks of DMA data around without any additional hardware design effort on your part.

The broad intent is that the software developer should use the PPE to herd data blocks from input streams to the SPEs, where the real computational magic takes place, and thence back to the output devices.

The tools are still familiar

Once you get your software development team over the surprisingly gentle learning curve of understanding the SPE programming interface, you can develop the whole system using familiar tools. You therefore end up with a laboratory instrument whose characteristics are defined almost entirely by software without any need to get involved with special-purpose DSP toolchains, tricky DMA architectures, ASICs, or FPGA programming.

Slight degree of customization

Note that while a from-scratch Cell/B.E. hardware design is definitely nontrivial, the degree of hardware customization required to develop a special application from a working Cell/B.E. reference design is comparatively small (since much of the device's important attributes can be implemented in software). Buried in this fact is the implication that significant functionality upgrades can be sold to end-users as simple software updates without any need to develop and certify new hardware.

This seems to be a compelling set of advantages and I would not be surprised to see Cell/B.E.-based spectrum analyzers, waveform synthesizers, and other complex test equipment such as base station simulators appearing in the near future.

The example

In this specific example, I'll start by looking at few things you need to know:

- Where's the Linux?
- Keeping the box from going up in flames.
- Working with the display
- Text-rendering code

Finding our inner Linux

When building an application around a PS3, you are constrained quite severely by the PS3's hardware and software design -- in particular, the fact that Sony has sandboxed Linux away from much of the hardware. A fully custom design — or even simply a generic Cell/B.E.-based mainboard with a custom PCI Express card containing your data acquisition/output hardware — would be much more flexible.

However, in line with the modest capabilities of the iMic, our target for this series is to work with two parallel (stereo) 44.1kHz 16-bit data streams, implying an audio bandwidth of 22.05kHz.

Ouch! That's hot!

Before you start, an important note: The PS3 hardware was designed to sit on a TV set or entertainment center, not on a workbench. It generates a lot of heat which is extracted by a fan that blows out the end of the unit closer to the Blu-Ray drive, as well as out the back.

When I first unpacked the unit, I ran it on my workbench next to my laptop with the PS3's fan blowing into the laptop's exhaust vents — the laptop eventually went into thermal shutdown. So I advise you to run the PS3 "standing up" (so the PLAYSTATION 3 text is right-side up; there are feet on the bottom of the unit to assist with this). In this configuration, it seems to run the coolest.

A picture's worth a thousand code lines

With that out of the way and Linux installed, the first task you need to tackle is working with the display. The default display configuration for your PS3's Linux install will depend on how exactly you installed it. If you're running with a normal NTSC or PAL TV set (as opposed to HDTV or a VGA converter connected to a monitor), it's not possible to use YDL's graphical install mode because it tries to set one of the progressive scan resolutions.

Therefore, the default Linux install will come up without X and with a TV-resolution screen. You can alter this behavior by editing /etc/kboot.conf — for the purposes of this article series, you'll want to run in one of the RGB modes, either 480i for NTSC users or 576i for PAL/SECAM users. Either of these will give you a screen size reported as 576x384 pixels by the framebuffer device; more on that a little later.

The specific framebuffer video mode at boot time is set by parameters passed to the kernel by the bootloader, kboot. (Note that this video mode setting is only examined after the ps3fb framebuffer device loads, since it is a kernel parameter. The video mode between the moment you power on and the moment the ps3fb device initializes is whatever you set in the Sony GameOS menus; by default, it's the interlaced SDTV resolution for the locale where your PS3 was purchased). The settings for kboot are stored in /etc/kboot.conf — here's how I've configured my system. Most of this is taken directly from the configuration generated by the YDL installer; I simply changed the video mode:

```
default=ydl
timeout=10
root=/dev/sda1
ydl='/dev/sda1:/vmlinux-2.6.16-20061110.ydl.2ps3
initrd=/dev/sda1:/initrd-2.6.16-20061110.ydl.2ps3.img root=/dev/sda3
init=/sbin/init video=ps3fb:mode:33 rhgb'
```

Changes made to kboot.conf take effect immediately from the next reboot; you don't need to do anything special to propagate the new configuration into the bootloader.

If you're using a TV set and you're in Europe, you will probably want to run in mode 38; simply change `mode:33` to `mode:38` in the previous listing. As a matter of interest, you can view a complete list of available modes by using the `ps3videomode -h` command. You can experiment with different modes on the fly by running `ps3videomode -v <number>`.

Now we'll do whatever it takes to write some pixels to that fresh new slate of blank video memory.

The PS3 video subsystem is heavily firewalled away from Linux by the Sony GameOS "hypervisor." It is not clear how much of this is due to a generalized terror that someone, somewhere may someday copy a PS3 game or see an unencrypted byte of HD video content, or how much is simply because Sony needed to develop a method of exposing a video interface to Linux without releasing any register-level documentation on the GPU (observe that this would potentially bring in all kinds of GPL side effects such as requiring public disclosure of code that is covered by a nondisclosure agreement with nVidia).

Whatever the motivations, the ps3fb video device works a little differently from other Linux framebuffers, which is both a curse and a blessing. An excellent, quite detailed Sony document on how it works is provided in the Resources section, although it neglects to mention at least one irritating bug. The points you need to observe can be summarized succinctly:

- Normally, framebuffer devices give you direct access to the video card's memory.
- Using the ps3fb device, your application writes to a main-memory (offscreen) buffer. Every vertical blank, the hypervisor DMAs this buffer to the GPU memory, and then flips the GPU's visible page to the new data synchronously with the vertical blank signal.
- The advantage to this system is that you never need to worry about "tearing" effects in animations caused by updating the screen contents partway through a frame.
- ps3fb also exposes ioctls that permit you to run the screen in a sort of single-buffered mode where you stop the periodic interrupts and explicitly dump new video data to the GPU when your application feels it's appropriate (still through the hypervisor, of course). X uses this mode.

BAKER 0000140

The irritating bug to which I referred is that the standard mode-info query ioctls don't quite work properly, at least for television resolutions. As I mentioned early on, the NTSC and PAL resolutions both report a virtual/physical size of 576x384 pixels. This isn't even vaguely correct; the actual width of the NTSC screen is 720 pixels and the height (as determined by mapping and poking progressively larger slices of RAM) is actually something like 480 lines, though the framebuffer console seems to stop at about 400 lines and the 480th line is somewhere off the bottom of the screen in the overscan area, at least with the default video configuration.

Therefore, generic framebuffer code that doesn't explicitly understand the PS3 will break spectacularly if you simply compile it and let it run. My workaround is to coerce the code into believing it's running on a 720x400-pixel screen, which seems to work nicely. You will want to change this if you're using something other than an NTSC TV set as your output device; though the code will still operate, it probably won't generate a legible display.

Another definite oddity of the PS3's output is that the overscan color is whatever was being output in the last pixel of the previous scanline. My theory on this is that the RAMDAC has a pixel latched into it on each DMA cycle and this latch freezes when the raster moves outside the framebuffer area up until the vertical blank interval, where it gets reloaded with a black pixel value. Another possibility is that this phenomenon is caused by some kind of subtle race condition with the hypervisor software.

The only reason I mention this is that if you draw a rectangle that touches the left edge of the screen, you'll see that the rectangle's color extends into the overscan area except in the upper-left corner of the rectangle (since that corner inherited the color from the end of the previous scanline). The phenomenon is easily noticeable even on a low-resolution television set; I don't want you reporting this as a bug in my code, because it isn't!

Make it legible

Another feature you'll need for your instrument is text-rendering code. Rather than draw out a character generator structure by hand, I've borrowed font_acorn_8x8.c from the Linux kernel (this is part of the framebuffer driver tree). Note: This is legal only because the sample code I'm providing here is GPL-licensed. If you need a character set that will allow you closed-source distribution, you'll need to search a bit. Red Hat's eCos operating system, for example, includes a character set you can use. The advantage of the Linux font is that it already includes all the hi-ASCII characters you might want.

At this point you can take a look at the sample code which includes the initialization I described earlier, as well as functions to plot single pixels, draw filled rectangles, and render text. The demo code in main.c draws a few colorbars onscreen, as well as a few lines of multicolor text, and prints some potentially interesting debugging information to the console. You'll find that all the graphics primitives I described above are in graphics.c and graphics.h, and are more or less self-documented.

A keyboard caveat

One final note which comes into focus when you run the sample code: I suggest you do not use a keyboard attached to the PS3 for your editing; do your development over ssh (the YDL install includes a fully-functioning ssh daemon by default; you don't need to configure anything). That way you can run the application on the ssh console and see useful debugging info on stdout without disturbing the graphics being displayed on the PS3's framebuffer.

Surprise! A functional graphics interface

BAKER 0000141

If you've been following along, you now have a functional graphics interface running on the PPE with some useful groundwork primitives ready to use in more advanced projects.

In the next article, I'll describe how to use the iMic to gather an analog data stream, and I'll show you how to use one of the SPEs to turn the system into a useful spectrum analyzer.

Downloads

The downloads for this article are being updated. Please try to download later.

Resources

Learn

- Jonathan Bartlett's "Programming high-performance applications on the Cell/B.E. processor" (developerWorks, January 2007) article on installing Linux on the PS3 is essential preliminary reading.
- Sony has released a very good document describing, inter alia, how the ps3fb device interacts with the GPU and your Linux programs. Note that this is a mirror; there doesn't appear to be an official copy of this document on Sony's sites.
- Some (very!) old information on using Linux framebuffer devices is available at the Linux Documentation Project. Frankly, you're much better off looking at some sample code snippets, linux/fb.h, for information on the various data structions, and the code and comments in the ps3fb.c driver (drivers/video/ps3fb.c in the linux-2.6.16-20061110 tree provided by Terra Soft Solutions).
- The developerWorks Cell Broadband Engine Resource Center provides new documentation, downloads, and community news for all things Cell/B.E.
- News, news, and more news can be found in the Thursday roundups of the Power Architecture blog.

Get products and technologies

- The Griffin iMic is our audio input device of choice. Note that the Web site shows a (newer) white model of the product. The model I have tested with PPC Linux is the older, translucent-and-silver version with a switch between the input and output jacks.
- You can get Yellow Dog Linux through free download from Terra Soft Solutions. My experience is that all the mirrors are quite slow; I got the install ISO much faster by searching a P2P network for the filename yellowdog-5.0-phoenix-20061208-PS3.iso.
- You can download the Cell/B.E. SDK (latest version 2.1) from alphaWorks.

Discuss

- Participate in the discussion forum..
- Got a question on how to leverage and program the processor? Pose your question in the Cell Broadband Engine Architecture forum.

About the author

Lewin A.R.W. Edwards works for a Fortune 50 company as a wireless security/fire safety device design engineer. Prior to that, he spent five years developing x86, ARM and PA-RISC-based networked multimedia appliances at Digi-Frame Inc. He has extensive experience in encryption and security software and is the author of two books on embedded systems development.

PS3 fab-to-lab, Part 2: Generating and analyzing signals

Try isochronous USB device audio data, extract spectrum information from a DFT library, and display the results

Lewin Edwards (sysadm@zws.com), Design Engineer, Freelance

Summary: How do you take the Cell Broadband Engine (Cell/B.E.) processor from an off-the-shelf Sony PLAYSTATION 3 (PS3) and use it to construct a piece of Linux®-based laboratory equipment (in essence, take the Cell/B.E. from fab to hab to lab)? In this series, Lewin Edwards shows you how to go from game console to simple audio-bandwidth spectrum analyzer and function generator. In this article, the author shows you how to build on the infrastructure from Part 1 to make the system into a fully operational, if primitive, spectrum analyzer.

Date: 02 Oct 2007
Level: Intermediate

Activity: 11113 views
Comments: 0 (Add comments)

★★★★★ Average rating (based on 3 votes)

Introduction

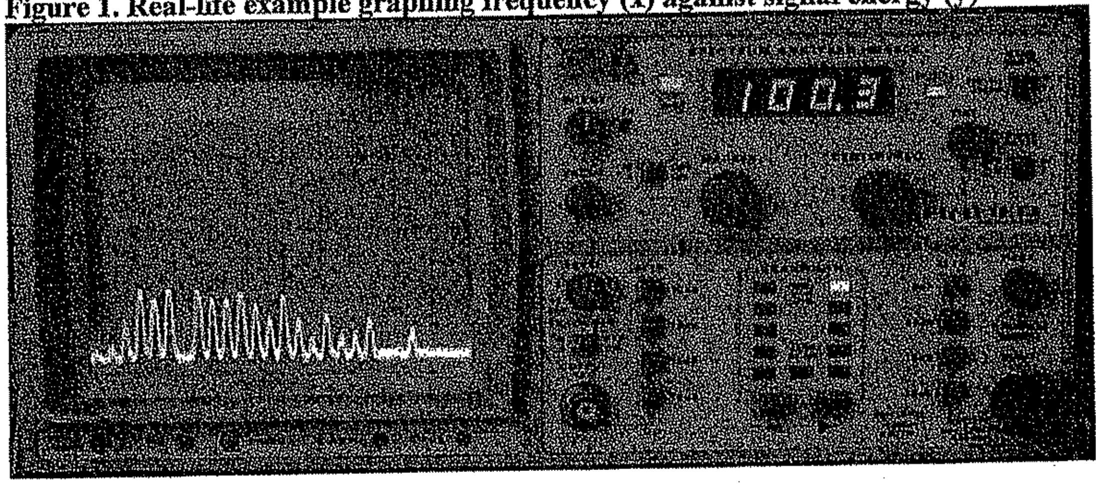
Part 1 mostly discussed infrastructure: support code needed in order to get something up on a PS3's screen and an explanation of various platform-specific oddities you'll encounter with this particular combination of hardware. In this article, you will see how to build on that infrastructure to make the system into a fully operational, if somewhat primitive, spectrum analyzer. To download the sample code referenced in this article, go to Part 1.

The target technology

The basic function of a spectrum analyzer is to decompose an input signal in the frequency domain and display a representation of the energy levels of different frequencies of interest. There are several approaches to building such a device depending on your needs and the acceptable project cost. The simplest type can be seen in stereo systems that have an LED or VFD bar chart display showing the output signal strength in a few discrete frequency bands (typically three to seven). The usual way of implementing such a display is to feed the input signal into a comb filter. Each *tooth* of the comb is fed to a circuit that is, in essence, a low-pass filter. The output of this second filter is a slow-moving average of the input signal level for one frequency band. This average is fed into a stack of comparators with progressively higher reference voltages, the outputs of which drive the display segments in one column of the bar chart.

A traditional analog spectrum analyzer is a considerably more complex beast, but the basic design principle is easy to understand. The front end is essentially a superheterodyne receiver with a wide tuning range. The center frequency of this receiver is voltage-controlled (typically by means of varactor diodes in the receiver's local oscillator). The control input is driven with an internally generated sawtooth waveform from an internal timebase. The same sawtooth drives the horizontal deflection of an oscilloscope trace; the output of the receiver drives the vertical deflection. What you actually see on the scope screen is therefore a graph of frequency (x) against signal energy (y). An example of such a display is shown in Figure 1.

Figure 1. Real-life example graphing frequency (x) against signal energy (y)



The display shows a 20 MHz slice of the broadcast radio spectrum from 90.3 MHz to 110.3 MHz as received by a rather badly mismatched antenna. WHTZ, 100.3 MHz, is the peak at the center of the trace. You can see various other FM radio stations at various signal strengths to either side of it.

The *non plus ultra* of spectrum analyzers is the (mostly) all-digital design. At the high end, this consists of an extremely high-speed, high-resolution analog-to-digital converter that acquires the input signal in the time domain. A fast digital signal processor then converts this to frequency domain data and displays the result on the screen, optionally performing various filtering or other processing. A high-end digital spectrum analyzer can also perform other intelligent tasks to help you look at a signal of interest. For example, the analyzer might *know* about frequency-hopping spread spectrum signaling systems and allow you to set up the hop list and protocol timing in the analyzer itself to track an ongoing communication session.

Complaints department

To prevent advanced readers from complaining: It would theoretically be possible to bring in signals of any arbitrary frequency (even up in the multiple gigahertz range) by using an external mixer to heterodyne the source down to the range of the iMic. In fact, some vendors of spectrum analyzers sell expansion boxes that do precisely this. However, because the iMic's bandwidth is severely limited, it would be irksome to scan across such high frequencies.

The system you are building is of the all-digital type. Unfortunately, the limiting performance factor for this is right at the front end: the PS3 hardware does note have a convenient method of acquiring high-speed signals. Therefore, this article helps you build something of a proof-of-concept device that is limited to the audible signal range of approximately 20 Hz to 20 kHz, using the Griffin iMic as the data acquisition device.

The four-step project list

It's time to get started. The subtasks in this project include:

1. Filter the input signal.
2. Acquire (digitize) the input waveform.
3. Convert the time domain data to the frequency domain.
4. Display the data attractively.

Step 2 (yes, 2). Digitizing the input waveform

The first thing to consider is how to funnel some data from the outside world into the PS3. You can use the Griffin iMic for this purpose, and there is really not much to say about the installation process. Recent Linux kernels include a compatible driver, so setting up the device on YDL is very much plug-and-play. You can verify that the device was mounted successfully by using `tail -10 /etc/dmesg` and checking for the appropriate USB messages, and then use the ALSA mixer to determine if you can tinker with the volume settings for the device.

For the programming side, there are a few different APIs you can use to access audio devices in Linux. The sample code presented with this article uses the Open Sound System API (OSS) mainly because it is uncomplicated to use and because its old age means it is well supported on various hardware and operating system flavors. The iMic is also supported by the Advanced Linux Sound Architecture (ALSA) API. It would not be unreasonably difficult to modify this sample code to work with ALSA.

The data stream will be sampled at 16 bits, 44.1 kHz. While this sampling frequency is not a particularly nice round number from a calculation point of view, it's a safe choice for hardware that was designed for audio recording. Because off-the-shelf hardware is often not tested rigorously against all possible API call parameters, it's prudent to choose *popular* sample formats and data rates when working with consumer hardware. This caveat basically restricts you to 11.025, 22.050, or the CD-quality sample rate of 44.100 kHz (though 48 kHz is also usually supported by modern audio hardware). I strongly advise you to stick to the upper end of this range. The reason for this is buried in the fact (which some of you might have noticed) that I glossed over step one in my project list. It's time to rectify that omission.

OK, now Step 1. Filtering the input signal

Normally, a digital data acquisition system starts with level-matching and isolation components followed by a sinc filter that rolls off, theoretically, to somewhere below the ADC's voltage resolution, across the frequency span, between the highest-frequency signal of interest and half the sample rate.

For the sample 16-bit ADC, that would theoretically mean 96 dB of attenuation (6 dB per bit) between the audible range of about 20 kHz and half the CD-quality sampling rate, which is 22.050 kHz. This requirement is an unfeasibly tall order for an analog circuit. It would involve an incomprehensibly high-order active filter network or a big compromise on performance parameters, such as passband ripple (or, more likely, both). Observably, the iMic does not contain such analog hardware. While it is *possible* that the device oversamples the input signal, filters it digitally, and downsamples it, this is very *unlikely*. Griffin does not publish specifications online, so you can make an educated guess that it is much more likely that the iMic compromises a bit by starting its rolloff earlier, quite probably not reaching the full 96 dB by the 22.050 kHz mark.

The net result of this discussion is that if you use a reduced sample rate, the iMic's front-end filter is not going to know about this, so it will continue passing through signals at frequencies that cannot be

BAKER 0000146

captured by your lower sample rate. This will cause aliasing artifacts to appear in your final output. As a result, the *best strategy is to capture at a known-good sample rate*. If you find that this generates too much data for the FFT engine to handle in a timely manner, then your next best plan is *still to capture at the higher rate, but to run a digital low-pass filter over the raw data then downsample it* before passing it on to the next stage.

Step 3. Converting data

When I planned this article, I was geared up to port an existing FFT algorithm to the Cell/B.E. platform and thereby impress you with my elite porting skills, but it seems that IBM already beat me to the punch. The latest alpha version of the FFTW library (see Resources) already includes explicit support for the Cell/B.E. processor. Basically the only thing you need to do is build and install the library, and then add `-lfftw3` and `-lm` to the linker flags in your Makefile. The tutorials included in the FFTW documentation are adequate to get you started. Note the caveat in the documentation regarding SPE usage. By default, the Cell/B.E. version chews up all available SPEs. Use the `fftw_cell_set_nspe(n)` call (it is in the documentation, but not right alongside the rest of the API description) to scale back FFTW's usage to `n` SPEs.

Initializing the FFTW library is simply a matter of completing the following two steps:

1. Allocate memory for the input and output buffer. It's best to use the `fftw_malloc()` function for this instead of the regular `malloc()` because the fftw-specific function optimizes data alignment. This is particularly important on platforms like the Cell/B.E. processor.
2. Develop and select a *plan*. There is actually a lot of arcane complexity in this step (well explained in the documentation). At its simplest, you call `fftw_plan_dft_1d()` and tell it the array size, pointers to the input and output arrays, whether you want to go forward or backward, and several flags that can be used to squeeze out optimal performance. For the example application, `FFTW_ESTIMATE` is perfectly acceptable as the flags' parameter.

The iMic delivers a stream of time-domain samples in the range 0 to 65535, 22.68 microseconds apart. You massage these (note that you are using only one channel of the stereo data stream) and place them in the FFT's input buffer. The example also plots a reduced-size, 128-pixel-high version of the input signal onscreen so you can see what it looks like in an oscilloscope-style format. That's not just a bit of eye candy, but rather to help you check that your input signal is properly connected and at an appropriate level. You also see a little snippet of code that you can uncomment if you don't have an iMic or a reference frequency source: it stuffs a sine wave directly into the sample buffer.

Now it's time to call `fftw_execute()`. This passes your sample data on to the SPEs, which crunch them into frequency spectrum data. The 512 real sample points are turned into two groups of complex spectrum data showing symmetry around the center. As the references are fond of saying, the *k*'th point in the output array represents the energy at a frequency of *k/n * Fs*, where *n* is the number of samples and *Fs* is the sample frequency.

The first entry in the table in memory (0 Hz) is a special case. It represents the *DC* level of the input signal, and hence is usually off the scale. The spectrum rendering code deliberately clamps the Y-coordinate to allow for this condition. Note that this probably is not representative of the actual DC level at the iMic's input pin. Most likely, the input is capacitively coupled, so there isn't any real DC at the ADC. Rather, this bogus spike represents the fact that your input signal does not vary equally positively and negatively about the 0 V line, but rather varies between 0 and +65535.

BAKER 0000147

Also note that, technically, you should use a logarithmic scale for the spectrum. The reason I don't do this in the sample code is that the resolution is rather low, and you can get a better idea of the signal shape from the linear plot.

Some technical details are in order here, particularly if you are now looking at the source code in puzzlement. Arbitrarily, I chose a 512-point FFT. FFTW does support arbitrary transform sizes, but you can realize much better performance with a size that is a power of 2. Speaking of performance, the generic complex one-dimensional transform I selected is not the optimal choice for your sort of input data. The fastest would be `fftw_plan_dft_r2c_1d()` (one-dimensional, real). The reason I went with the generic case is that is applicable to other sorts of data, and the additional computation load is really child's play to the PS3.

By the way, you shouldn't think of the FFT algorithm as being just a monolithic number-crunching magical black box. Plenty of research has gone into methods of computing FFTs and, among other things, how to factor a given FFT operation across multiple digital signal processors (DSPs). If, for some reason, you find the FFT itself is a bottleneck, there is a great deal of existing code (including, in this case, fftw) that can accelerate your application by splitting it up, if you throw more cores at the problem.

Step 4. Making a pretty display

Now that you have sorted the input data into buckets, all that remains to be done is to display it. To do this, calculate the magnitude of each complex output point using simple Pythagoras and plot that number. If you wanted an actual power reading in dB, you should instead plot 20 times the base-10 log of the output point. This is not terribly useful information without some kind of reference marker though.

Observe that if you were to plot all 512 output points, you would see a lot of irrelevant information. Everything to the right of the 22.050 kHz mark is aliased and might not actually exist in the input signal. Hence, the code accompanying this article only plots the first 256 points, and it doubles the horizontal size so the display fits neatly under the oscilloscope display.

At this point, experiment a bit with performance. In particular, try building the FFTW libraries without Cell/B.E. support so they use only the PPE. The improvement would be more noticeable if you were doing a larger transform because the SPEs are much better at this sort of thing than the PPE. The small size of your data set means the transaction overhead is a significant fraction of the execution time.

Surprise! A useful spectrum analyzer

So, you have now turned a PS3 into a useful spectrum analyzer. The next article in the series examines the other side of that equation and uses the same hardware as a function generator: the basis of an audio synthesizer, among many other things.

Resources

Learn

- Use an RSS feed to request notification for the upcoming articles in this series. (Find out more about RSS feeds of developerWorks content.)
- Check out all the articles in the series. Part 1 uncovers the design intent of the project and inspects the details of the user interface implementation.
- Refer to Advanced Linux Sound Architecture (ALSA) (which kinda sounds like a band from the 1980s) for audio and MIDI functionality to Linux with fully modularized sound drivers, SMP and thread-safe design, support for the older OSS API, binary compatibility for most OSS programs, and a user space library to simplify application programming and provide a higher level functionality.
- Read the Open Sound System (OSS) 4.0 Programmer's Guide for a wealth of well-chosen tiny demo applets that demonstrate recording, playback, and various mixer tweakage. Study this code for the fastest way to get up to speed on the required steps.
- Go to the home of the Poor Man's Spectrum Analyzer for just one of many sites for building garage-project lab equipment.
- Find the latest 3.2alpha2 prerelease version of the fftw Fast Fourier Transform library, including IBM-supplied Cell/B.E. optimizations.
- See "25 tips to optimal application performance" (developerWorks, June 2006) for how you can achieve near theoretical-maximum performance for real applications on the Cell/B.E. processor by learning about the processor's architectural characteristics.
- Review Jonathan Bartlett's essential preliminary article "Programming high-performance applications on the Cell/B.E. processor" (developerWorks, January 2007) about installing Linux on the PS3.
- Check out the document Sony released describing inter alia how the ps3fb device interacts with the GPU and your Linux programs. Note that this is a mirror document; there doesn't appear to be an official copy of this document on Sony's sites.
- Refer to the Cell Broadband Engine documentation section of the IBM Semiconductor Solutions Technical Library for a wealth of downloadable manuals, specifications, and more.
- Sign up for the developerWorks newsletter and get the latest developer news and Cell/B.E. happenings delivered to your inbox each week. Check *Power Architecture* when you sign up to receive Cell/B.E. news in your newsletter.

Get products and technologies

- Jump over to Part 1 if you need to find the sample code referenced in this article.
- Look for the Griffin iMic: my audio input device of choice. Note that the Web site shows a (newer) white model of the product. The model I have tested with PPC Linux is the older, translucent-and-silver version with a switch between the input and output jacks.

BAKER 0000149

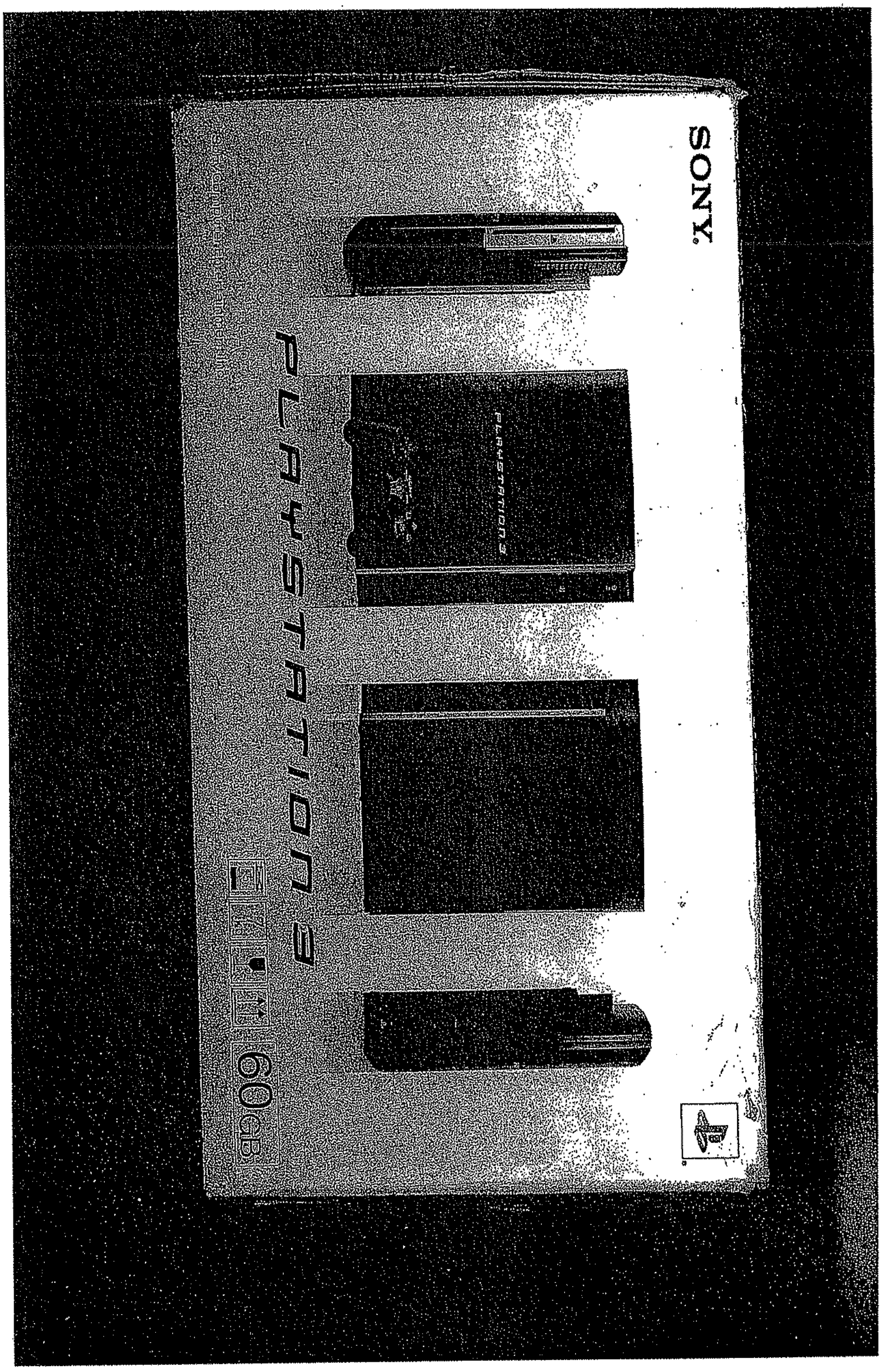
- Download Yellow Dog Linux through a free download from Terra Soft Solutions. My experience is that all the mirrors are quite slow. I got the install ISO much faster by searching a P2P network for the filename yellowdog-5.0-phoenix-20061208-PS3.iso.
- Find all Cell/B.E.-related articles, discussion forums, downloads, and more at the IBM developerWorks Cell Broadband Engine resource center: your definitive resource for all things Cell/B.E.
- Contact IBM about custom Cell/B.E.-based or custom-processor based solutions.

Discuss

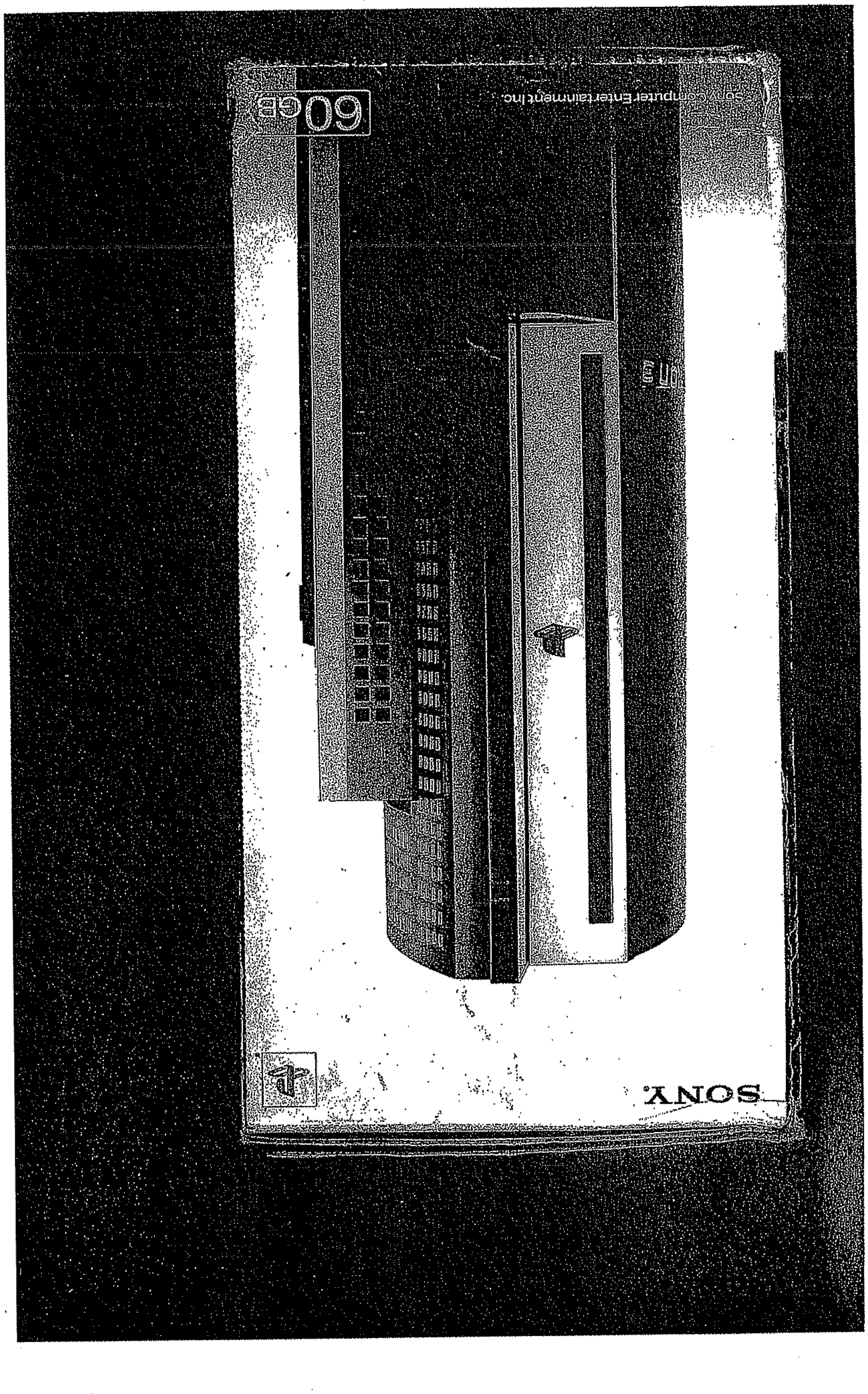
- Participate in the discussion forum.
- Check out the Cell Broadband Engine Architecture forum to get your technical questions about the processor answered. Juicy problems and answers from the forums are rounded up periodically and highlighted in the "Forum watch" blog series.
- Go to the Power Architecture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

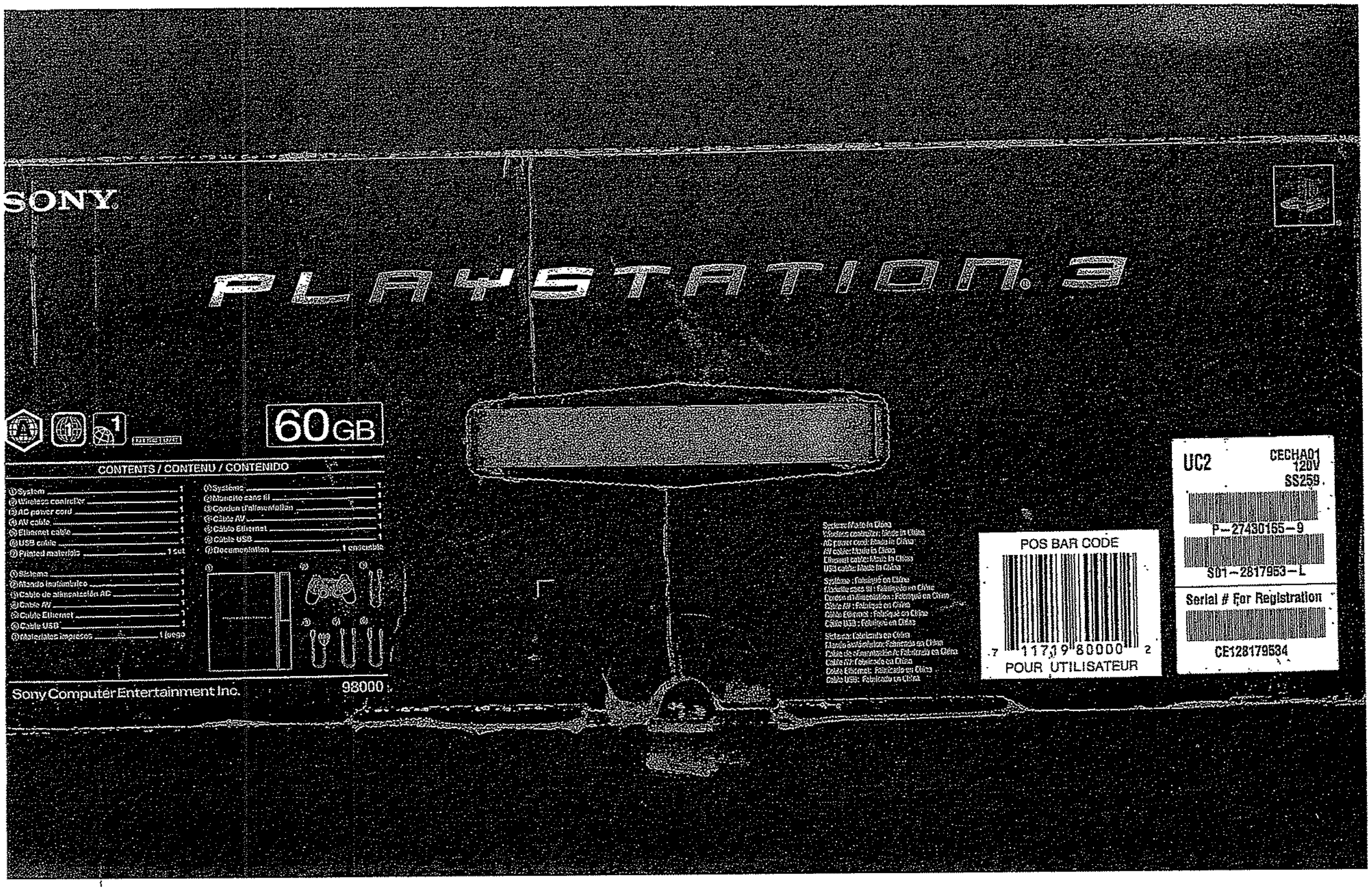
About the author

Lewin A.R.W. Edwards works for a Fortune 50 company as a wireless security/fire safety device design engineer. Prior to that, he spent five years developing x86, ARM and PA-RISC-based networked multimedia appliances at Digi-Frame Inc. He has extensive experience in encryption and security software and is the author of two books on embedded systems development.

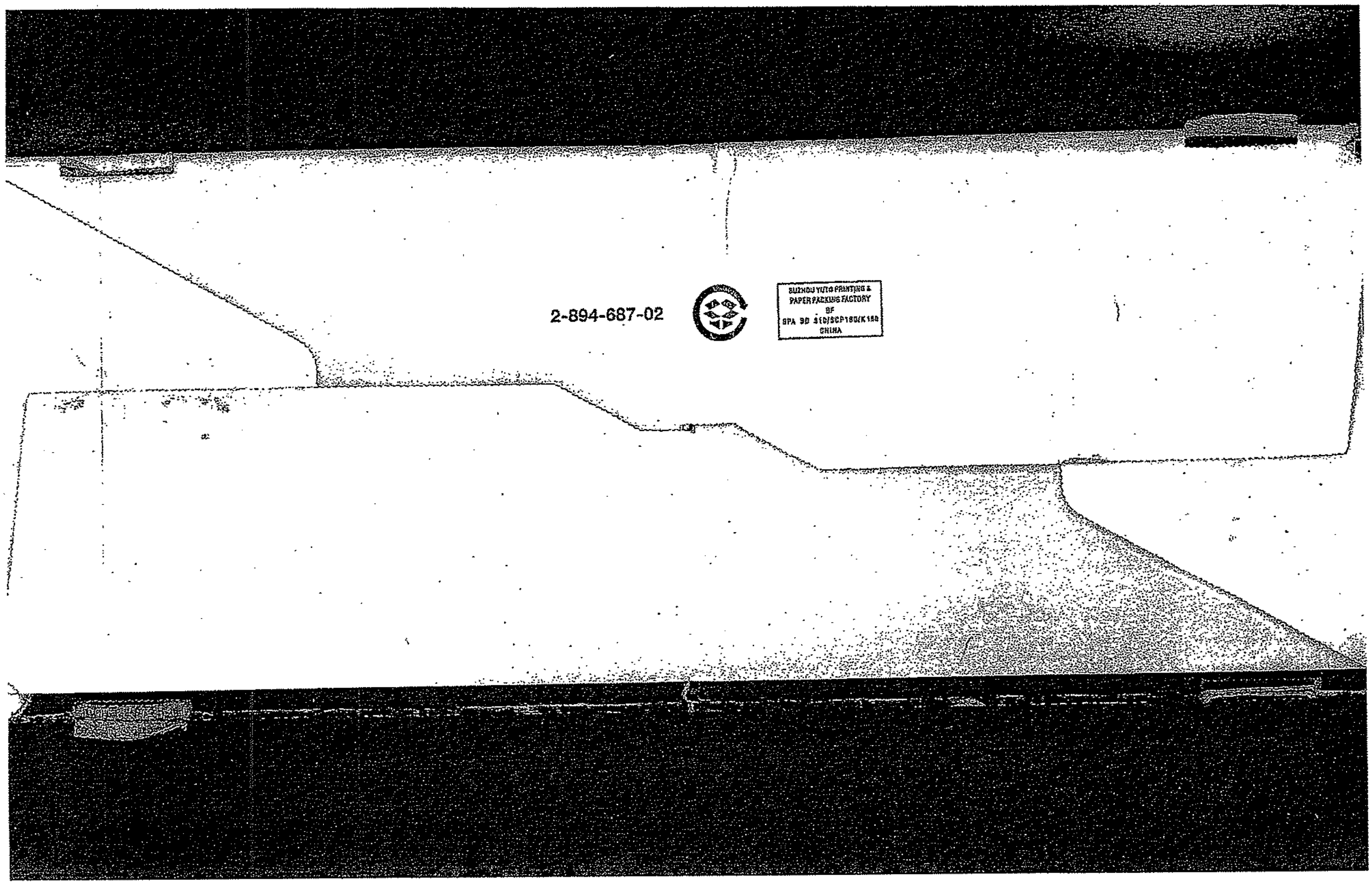


BAKER 0000151


60GB
Sony Computer Entertainment Inc.
SONY


SONY
PLAYSTATION® 3
60GB
CONTENTS / CONTENU / CONTENIDO
①System 1
②Wireless controller 1
③AC power cord 1
④AV cable 1
⑤Ethernet cable 1
⑥USB cable 1
⑦Printed materials 1 set
①Système 1
②Manette sans fil 1
③Cordon d'alimentation 1
④Câble AV 1
⑤Câble Ethernet 1
⑥Câble USB 1
⑦Documentation 1 ensemble
①Sistema 1
②Mando inalámbrico 1
③Cable de alimentación AC 1
④Cable AV 1
⑤Cable Ethernet 1
⑥Cable USB 1
⑦Materiales impresos 1 juego
Sony Computer Entertainment Inc.
98000
System: Made in China
Wireless controller: Made in China
AC power cord: Made in China
AV cable: Made in China
Ethernet cable: Made in China
USB cable: Made in China
Système : Fabriqué en Chine
Manette sans fil : Fabriqué en Chine
Cordon d'alimentation : Fabriqué en Chine
Câble AV : Fabriqué en Chine
Câble Ethernet : Fabriqué en Chine
Câble USB : Fabriqué en Chine
Sistema: Fabricado en China
Mando inalámbrico: Fabricado en China
Cable de alimentación AC: Fabricado en China
Cable AV: Fabricado en China
Cable Ethernet: Fabricado en China
Cable USB: Fabricado en China
POS BAR CODE
7 11719 80000 2
POUR UTILISATEUR
UC2
CECHA01
120V
SS259
P-27430155-9
S01-2817953-L
Serial # For Registration
CE128179534

BAKER 0000153


2-894-687-02
SUZHOU YUTO PRINTING &
PAPER PACKING FACTORY
SF
SPA 30 S(D/SCP180/K180
CHINA

PLAYSTATION 3

Lair

Heavenly Sword™

Resistance: Fall of Man™

Warhawk™

Genji: Days of the Blade™

MotorStorm™

RP Visit www.esrb.org for updated rating information.

60GB

7 kg

English

[illegible]

Français

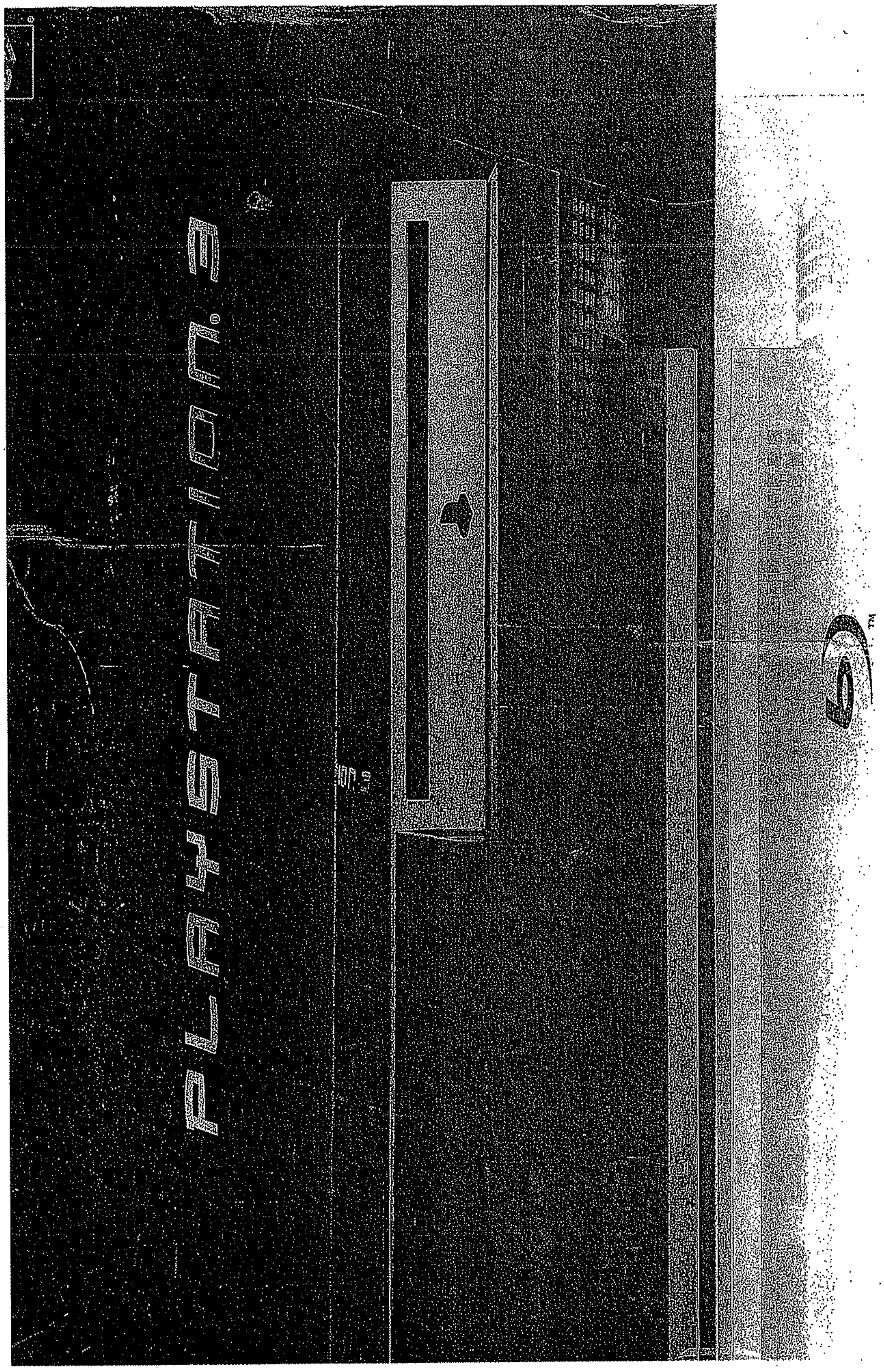
[illegible]

Español

[illegible]

SONY

BAKER 0000155



BAKER 0000156