==========
T H E " U N - O F F I C I A L"
PLAYSTATION DEVELOPMENT FAQ

OS
CONFERENCE
==========

Release v1.2
Last Updated: February 29, 1996

DISCLAIMER

This FAQ is to aid in informing the licensed game developer about the development environment provided by Sony Computer Entertainment.

The Development System Tool to which this manual relates is supplied pursuant to and subject to the terms of the Sony Playstation Licensed Developer Agreement.

This FAQ is intended for distribution to and use only by Sony PlayStation Licensed Developers in accordance with the Sony Playstation Licensed Developer Agreement. The information in thismanual is subject to change without notice.

The content of this manual is Confidential Information of Sony for the purposes of the Sony PlayStation Licensed Developer Agreement and otherwise.

TRADEMARK INFORMATION

COPYRIGHT NOTICE

[1.] OS

[1.1.]: How do I initialize the GP?

Properly accounting for the (GP) register within GNU C is as follows:

```
static int MyVarNonGp __attribute__ ((section("data"))) = 0;
```

This will cause all optimized and non-opimized code to use direct addressing when referencing this function. Other methods will work, however, this is clean and should work properly in all cases.

[1.2.]: I would like to know about the relation between physical space and logic space.

The physical space represents the main memory space on PlayStation, and the address is from 00000000 to 0x001fffff.(in the case of 2 MB) The logic space refers to the area mapped logically, and the following is a logic mapping example.

```
  0x80000000 to 0x801fffff
  0xa0000000 to 0xa01fffff
```

This 2 MB will be the logic space. If the mapping starts from 0x80000000, I-cache will be effective, and instructions will be executed via cache. If the mapping starts from 0xa00000000, I-cache will be ineffective.

In other words, the logic space is determined by how to allocate the physical 2 MB space (area) in 32-bit address apace (4 GB).

The execution space including the global variables and the heap area for malloc() is set by "-Xo" option designated in linking ccpsx. For example, when it is specified as -Xo$80100000, the execution space is allocated in the logic space within 80000000 to 0x801fffff(= segment B). The value designated in the InitHeap() function can be specified in this area.

A segment name depends on where the first address is set in the logic space. Separate segments are not prepared for each data type like Intel x86 series. Since a RISC chip is adopted for PlayStation, effective cache is preferable. Thus, as seen in the sample, it is better to use the area from 0x80000000 basically.

[1.3.]: Is it possible to nest the child process activated by Exec()?

Yes, but keep in mind that the memory management is difficult.

[1.4.]: I would like to know about I-cache strategy and a wait at the time of the cache-miss in PlayStation.

A direct mapping strategy is adopted for I-cache. It takes 4 cycles to read the first word. Thus, the wait at the cache-miss will be 5 cycles in total, summing the 4-cycle stalls and 1-cycle FixUp.

[1.5.]: What should be mentioned in order to speed up the program processing?

The ways to achieve it are as below:

* Effective use of the scratch-pad By providing often-used arrays, global variables, or functions, much speeding-up can be expected.

* Selection of the compiler option (use of I-cache)
Select the option, which achieves the higher cache- hit rate, from either -O3 or -O2. .

* Control of the number of arguments of a function created by C (4 or less)With R3000, up to 4 arguments are passed via register, and more via memory.

* Use of texture cache/CLUT cache Arrange the drawing method to raise the hit rate.

[1.6.]: How long does it take to access to the memory once, compared with the transmission between registers?

It depends on the cache-hit rate, but basically it is as below. It is better to keep in mind passing via register as possible.

Transmission between registers	1 cycle
Access to the data on the memory	5 cycles
Access to the data on the scratch-pad	1 cycle

[1.7.]: For a module which 2mbyte.obj is linked.

In starting up, the following processing takes place:
The bss area is cleared by zero
A stack pointer value is set at 0x80200000
The InitHeap() is called.

[1.8.]: For a module which none2.obj is linked.

In starting up, no processing takes place.
The InitHeap() is not called.
Thus, one has to call InitHeap() to use the malloc().

[1.9.]:What size is my executable in memory ?

The size of your executable in total in PSX memory is:

text size + data size + bss size

Your program has 3 parts:
- The program 'text', which is the actual code for your game;
- The data, which is global variables which you have initialised in you code;
- The bss segment, which is the space your program needs for uninitialised data (or globals or statics)

So to provide a handy example, consider the following silly code:

```
#include <stdio.h>

int numbers[5] =  { 0, 1, 2, 3, 4 };
int myNumber;

int Add(int a, int b)
{
  int temp;
  int max = 0xffffffff;
  static int odd;

  temp = a + b;

  return temp;
}
```

- The text section will contain the assembler version of the function (but with no variables or anything).

- The array 'numbers' is stored in the data segment (because it must be initialised with the values you specified, which are stored as part of your executable).

- The integer 'myNumber' is stored in the bss section (because it is not initialised).
- The variable 'temp' is created on the stack at run time, and so is not part of the executable segments.

- The variable 'odd' is stored in the bss section too, because its static, and must keep its value across function calls.

- The variable 'max' is actually created on the stack at run time too, and initialised with the value you give to it at function entry timw (so this is actually more expensive in terms of execution time than using a global, which is initialised at program start up, or the best case of all, using a #defined value).

- The heap addresses will be in kseg0 (cacheable memory), even though the data cache is not automatically used.

So the size of the heap at program start is this:

```
  2 Mb (Main RAM size) -
  32k (standard stack) -
  64k (kernel RAM) -
  (text length + bss length + data length (your program))
```

Libsn automatically initialises the heap for you at program start. Its possible to change the stack size of you want to, but don't mess with the
kernel RAM.

AJM
22/02/95

[1.10.]:Question on the speed of the D-cache in relation to data reading and writing.

[1.10.1.]:What speed is the D-cache in comparison to main RAM at reading data

Five times faster. It takes one CPU cycle to read from D-cache, and it takes five CPU cycles to read from main RAM.

[1.10.2.]:What speed is the D-cache in comparison to main RAM at writing data ?

It takes one CPU cycle to write to D-cache from general registers. Writing to main RAM is the moust complicated part of R3000 CPU. I should say that only GOD knows the exact number of cycles for writing data to main RAM!

R3000 CPU has "Write-buffer" between registers and main RAM. W-buffer is four step 32bit length fifo. It takes one cycle to write to W-buffer. But if there are no free register on W-buffer, CPU must flush W-buffer, write ALL the data on W-buffer to main RAM.

It takes one or four cycles to write a data on W-buffer to main RAM. If two or more write operations are done continuously, the first operation takes four cycles and the second and later operations take only one cycle.

And main RAM has 1KByte "pages". Any write operation on new page takes four cycles. And programmer cannot control or detect the W-buffer flush timing, and cannot know the status of it. R3000 has Bus-Snoop-Mechanism and I-cache. You cannot predict the start of flushing of W-buffer, even if you know the complete assembler codes.

So if you have good luck, it takes only one cycle to finish one store instruction. For the worst case, I can say nothing.

As the result, I can only say that writing to main RAM is VERY slow comparted to writing to D-cache, and probably writing to main RAM is faster than reading from main RAM.

Best Regards,

Okamoto

PS Each step of W-buffer is assigned to one store instruction. So if four Store-Byte-Instructions are executed, W-buffer becomes full.