SPS2

A Development Library for
Linux (for PlayStation 2)

Steven “Sauce” Osman
Terratron® Technologies Inc.

Table of Contents

Introduction

SPS2 combines a library of inline functions with a Linux kernel module to facilitate the development of high-performance applications on PlayStation 2 systems running Linux (for PlayStation 2). The goal of SPS2 is to reduce the difference between developing within the Linux environment and directly to the PlayStation 2 while allowing the developer to leverage the Linux services and tools. A future version of SPS2 will allow developers to compile applications to run either directly within the Runtime Environment or within the Linux environment with no modifications to the source code.

SPS2 enables high-performance applications by allowing applications full access to the PlayStation 2 DMA controller. It enables developers to allocate non-swappable memory segments, allows the developers to obtain the physical addresses of the memory segments, and enables access to the DMA controller registers in order to configure and initiate the DMA transfers. In addition, SPS2 gives developers access to all of the memory-mapped Emotion Engine and Graphics Synthesizer registers as well as the Vertex Unit memories and the Scratch Pad memory. Finally, SPS2 enables developers to access memory allocated through SPS2 in both a cached and uncached manner.

Current Linux (for PlayStation 2) development utilities are either too restrictive; requiring that the kernel be modified and recompiled so that a predefined portion of memory be permanently put aside for use in DMA transfer, or don't provide satisfactory performance; by requiring the use of system calls to change register values or to initiate DMA transfers that could perform a number of memory allocations and data shuffles before invoking the transfer. SPS2 seeks to address both of these issues by allowing programmers to allocate DMA friendly memory during runtime and access the appropriate registers directly.

However, SPS2 is not without its faults. First, by exposing the DMA controller to non-privileged users, improper use of SPS can compromise the stability and security of a system. Whereas future versions of SPS2 will provide some tools to minimize the likelihood of crashing the system during debugging, allowing users full access to the DMA controller will remain a security concern; this is, unfortunately, a price that must be paid in favor of performance. Users are cautioned not to allow access to their PlayStation 2 to people they do not trust. Another problem specific to SPS2 is that whereas large amounts of memory can be allocated, SPS2 cannot guarantee that the entire memory region is physically consecutive. SPS2 can only guarantee that individual pages – that is, 4096 bytes chunks – are physically consecutive. This problem can be easily worked around by creating a reference DMA chain for data that extends beyond 4096 bytes, pointing to the sequence of 4096 byte chunks of the data.

SPS2 delivers on its promise of providing high performance for Linux-based PlayStation 2 applications. For example, one of the provided sample applications, *vspeed*, is capable of generating 460,000 textured, shaded polygons per frame, yielding about 34.2 million vertices per second.

For the remainder of this document, Linux (for PlayStation 2) and PlayStation 2 will be referred to as PS2 Linux and PS2, respectively.

License Agreement

SPS2 IS DISTRIBUTED "AS IS". NO WARRANTY OF ANY KIND IS EXPRESSED OR IMPLIED. YOU USE AT YOUR OWN RISK. TERRATRON TECHNOLOGIES INC. WILL NOT BE LIABLE FOR DATA LOSS, DAMAGES, LOSS OF PROFITS, OR ANY OTHER KIND OF LOSS WHILE USING OR MISUSING THIS SOFTWARE OR ANY DERIVATIVE WORKS THEREOF.

In terms of your right to distribute applications that use the SPS2 libraries or kernel module, you may:

1. Include the library files (sps2*.h) with your application in source or binary form, provided that you do not modify them
2. Include any or all of the sample and framework code, modified or unmodified, in source or binary form with your application

You may not:

1. Distribute a modified version of the library (sps2*.h) files, either with your application or separately.
2. Distribute a source or binary version of the SPS2 kernel module, whether modified or unmodified
3. Distribute any derivative work of the SPS2 kernel module

This license is not intended to be overly restrictive; rather, it is designed to ensure maximum compatibility across different kernel versions and across different SPS2 versions. If you would like to suggest any changes to the module or the libraries, please contact the authors who will, at their sole discretion, choose to incorporate the changes in future releases.

Most importantly, this license is designed for the benefit of the community at large; it discourages multiple “SPS2-like” modules from being released, causing severe fragmentation in the Linux (for PlayStation2) community while giving developers motivation for getting their updates included in the official SPS2 distribution.

Acknowledgements

This package would have never been completed had it not been for the support and encouragement of two brilliant people, Lionel Lemarié and Morten Mikkelsen (aka Hikey and Sparky). Hikey and Sparky patiently answered many of my (often stupid) questions and provided a lot of input into what functionality this library should provide. They also put together the sample applications that are provided in this package and tested the software for both stability and functionality.

Not only have I gained a lot of knowledge about the workings of the PS2 from Hikey and Sparky, but also, and more importantly, I have gained two good friends.

Steven Osman

Programming with SPS2

SPS2 is a kernel module which is accessed through a special device file, typically `/dev/sps2`. SPS2 provides two API's: the first, a set of `ioctl()` commands; and the second, a number of inline functions. Use of the `ioctl` commands is heavily discouraged simply because the inline functions are more convenient and safer to use. In terms of performance, the inline functions provide only minimal error checking before invoking the `ioctl` commands, so they do not impact performance noticeably. Another reason that use of the `ioctl` commands is discouraged is that future versions of SPS2 may no longer support that function set. For example, if a version of SPS2 were developed that allowed applications to run directly within the Runtime Environment without the Linux kernel loaded, there might be no support for invoking file operations such as `ioctl()`.

This library of inline functions is further divided into two groups. The first group provides a full set of functionality with a good set of defaults. Most developers should use this set of functions. All functions in this group begin with the letters `sps2`. The second set of functions provides a slightly more granular level of control at the cost of a few minor inconveniences to the developer. These functions, which have names beginning with `_sps2`, will be of interest to developers of middleware and other libraries that leverage off of SPS2. These two sets of functions will be referred to as the *SPS2 Core Function Set* and the *SPS2 Extended Function Set*, respectively, and the `ioctl` commands will be referred to as the *SPS2 ioctl Command Set*.

Developers should be forewarned: SPS2 enables the developer to do much more than Linux would ordinarily allow. This includes crashing the system. Developers are encouraged to take a number of precautions to minimize data corruption and loss:

- Back up all files regularly
- Carefully read through code that builds DMA commands before executing it
- Sync the file system before executing an untested application (use `man sync` for more information)
- If possible, mount some or all of the file system as read-only before executing an untested application (use `man mount`)

Whereas syncing the file system will greatly reduce the chance of data corruption should the PS2 crash, mounting partitions as read-only (in addition to the sync) will greatly reduce the boot time of Linux by not requiring a file system check (`fsck`) of the read-only partitions. Future versions of SPS2 will include tools to help predict the likelihood of a DMA transfer causing the PS2 to crash, allowing the developer to circumvent the transfer. Ideas for additional debugging tools are welcomed and encouraged.

SPS2 Files and Directories

Included in the SPS2 archive are a number of files and directories. The following table outlines the roles that these items play:

Name	Type	Role
sps2interface.h	File	Declares the *SPS2 ioctl Command Set* and the format of their parameters and return values
sps2lib.h	File	Defines the *SPS2 Core Function Set* and the *SPS2 Extended Function Set* functions. All functions are inline; there is no linking required.
sps2registers.h	File	Defines macros for the Emotion Engine registers and the Graphics Synthesizer registers. Also includes structures for some of the Emotion Engine registers.
sps2scratchpad.h	File	Defines macros to access the scratch pad memory
sps2vumemory.h	File	Defines macros to access the vertex unit memories
Tests	Directory	Contains a small set of test applications. These are more useful for developing the kernel module than as sample applications.
samples	Directory	Contains a number of sample applications that use SPS2. Also provide a framework by which other applications can be developed

The following table describes the directories within the samples directory:

Name	Type	Role
readme.txt	File	A description of the samples with comments about the techniques they illustrate
Makefile	File	This Makefile builds all of the samples
bumpmap	Directory	This sample uses the GS blendmodes to perform per-pixel bumpmapping
common	Directory	Contains a number of common files used by most of the samples, including the framework
dyntexs	Directory	Illustrates how to synchronize the texture upload with the geometry upload. Most closely approximates a “real” PS2 application
int_lock	Directory	Creates an interlock loop to perform a number of operations in parallel
ps2lframework	Directory	Defines a framework by which other applications can be developed
vspeed	Directory	Illustrates the speed capabilities of the PS2, even within the Linux kernel by rendering 560,000 transformed, textured and shaded polygons per frame

Installing and Loading SPS2

The SPS2 Kernel Module is shipped separately. Programmers should download and install the kernel module before using this library.

Building the Kernel Module

In order for applications that use SPS2 to work, the sps2_mod kernel module must be built and loaded. To achieve this, starting from the root directory of the SPS2 kernel module distribution perform the following steps:

```
make depend
make
```

If you're not running as root at this point:

```
su
```

and finally:

```
make install
```

This will build the kernel module for the current kernel version loaded, and install it in `/lib/modules/<version>/misc`. It also installs two scripts, `sps2_load` and `sps2_unload` in `/usr/sbin`. Finally, it configures the module to load automatically when the system is booted up in runlevels 2, 3, 4, or 5, and unload when the system is switched to runlevels 0, 1 and 6.

Loading the Kernel Module

To load the SPS2 kernel module, as root, invoke:

```
/usr/sbin/sps2_load
```

Unloading the Kernel Module

To unload the SPS2 kernel module, as root, invoke:

```
/usr/sbin/sps2_unload
```

Removing SPS2 From Your System

SPS2 can be permanently removed from your system by changing to the root directory of the SPS2 kernel module distribution and, as root, performing the following step:

```
make uninstall
```


Building and Running the Sample Applications

In order to build and run the samples, first follow the instructions in the section *Building and Loading the Kernel Module*. Then, starting from the root directory of the SPS2 distribution:

```
cd samples
make depend
make
```

To run the *bumpmap* sample, starting from the samples directory:

```
cd bumpmap
./spky_bumpmap
```

You can use Control+C to exit the application.

To run the *dyntexs* sample, starting from the samples directory:

```
cd dyntexs
./mskpath3app
```

You can use Control+C to exit the application.

To run the *int_lock* sample, starting from the samples directory:

```
cd int_lock
./myapp
```

To run the *ps2lframework* sample, starting from the samples directory:

```
cd ps2lframework
./defapp
```

You can use Control+C to exit the application.

To run the *vspeed* sample, starting from the samples directory:

```
cd vspeed
./vspeed
```

You can use Control+C to exit the application.

A Sample SPS2 Program

This file can be found as `tests/hello.c`. It illustrates many of the key concepts in the SPS2 library. The following code shows how to:

- Initialize the SPS2 library
- Allocate memory using the SPS2 library
- Execute a simple, normal-mode DMA transfer to the scratch pad memory
- Flush the cache to ensure that all data to be transferred is written to memory
- Access the DMA controller registers directly to initiate a DMA transfer
- Access the scratch pad memory directly to display some data that was just transferred
- Access the Graphics Synthesizer registers directly to change the background color
- Shut down the SPS2 library

```
#include <stdio.h>
#include <sps2lib.h>

// This is the string we will be outputting
#define OUTPUT_STRING "Hello SPS2 world!\n"

// This is the number of Q-Words (i.e. 16 byte increments) to copy
// We want to round up to the next Q-Word
#define OUTPUT_STRING_QWC ((strlen(OUTPUT_STRING)+15) >> 4)

int main(int iArgC, const char **ppcArgV) {
  int iSPS2Device;                 // Handle to the SPS2 device
  char *pcMemory;                  // DMA memory for transfer
  sps2Memory_t *pSPS2Memory;       // DMA memory descriptor
  Dn_CHCR_t chcrValue;             // Value sent to DMA controller to
                                   // initiate transfer
  Dn_SADR_t sadrValue;             // Destination scratch pad address
  Dn_MADR_t madrValue;             // Source memory address

  iSPS2Device=sps2Init();          // Initialize the SPS2 device

  if (iSPS2Device<0) {
    fprintf(stderr,"Error initializing SPS2 library\n");
    exit(-1);
  }

  // Allocate 4K of memory.  We're allocating in 4K chunks, we want this
  // memory to be cached this could improve performance if we did a lot
  // of work on this memory before transferring.  We have to remember to
  // flush the cache before the transfer
  pSPS2Memory=sps2Allocate(4096, SPS2_MAP_BLOCK_4K | SPS2_MAP_CACHED,
                           iSPS2Device);

  if (!pSPS2Memory) {
    fprintf(stderr, "Error allocating memory\n");
    exit(-2);
  }

  // Get the actual pointer to the memory
  pcMemory=(char *) pSPS2Memory->pvStart;

  // Copy our string to the memory
  strcpy(pcMemory, OUTPUT_STRING);

  // Flush the cache
```



```
  sps2FlushCache(iSPS2Device);

  // Set the memory address of the DMA transfer.  We're using channel
  // 9, which is a transfer to the scratch pad memory

  madrValue.i32=0;                  // Make sure all bits are zero
  // We're not copying from the scratch pad
  madrValue.s.SPR=0;
  // Get the physical address for the memory we allocated
  madrValue.s.ADDR=sps2GetPhysicalAddress(pcMemory, pSPS2Memory);
  // Set the memory address register in the DMA controller
  *EE_D9_MADR=madrValue.i32;

  sadrValue.i32=0;                  // Make sure all bits are zero
  // We're copying to the beginning of the scratch pad
  sadrValue.s.ADDR=0;
  // Set the scratch pad memory address in the DMA controller
  *EE_D9_SADR=sadrValue.i32;

  // Set the number of q-words to transfer.
  *EE_D9_QWC=OUTPUT_STRING_QWC;

  chcrValue.i32=0;                  // Set all CHCR bits to zero
  chcrValue.s.MOD=CHCR_MOD_NORMAL;  // Normal DMA transfer
  chcrValue.s.STR=1;                // Start DMA transfer

  // Set the Dn_CHCR register.  This starts the DMA transfer since we set STR=1
  *EE_D9_CHCR=chcrValue.i32;

  sps2WaitForDMA(9, iSPS2Device);  // Wait for DMA transfer to finish

  // Display the string now stored in the scratch pad
  printf((char *) SCRATCH_PAD);

  DPUT_GS_BGCOLOR(0xff0000);      // BG color is BBGGRR, set it to all blue

  sps2Release(iSPS2Device);       // Close the SPS2 library

  return 0;
}
```


SPS2 Programmer's Guide

The following sections outline the basic steps needed to perform common tasks with the SPS2 library.

Performing a DMA Transfer

One of the most important reasons to use SPS2 is because it grants you complete access to the DMA controller. In order to perform a DMA transfer, a developer needs to perform several steps:

1. Include `sps2lib.h`
2. Initialize the SPS2 device with `sps2Init` or `_sps2Open`
3. Allocate unswappable memory with `sps2Allocate`
4. Load data to be transferred into the memory
5. (optional) build DMA chains also within the unswappable memory
6. (optional) if using cacheable memory, flush the cache with `sps2FlushCache`
7. Setup the DMA controller registers (see *Accessing the Emotion Engine Registers* below)
8. Start the transfer by setting the `STR` bit on the `Dn_CHCR` register to 1 (see *Accessing the Emotion Engine Registers* below)
9. (optional) wait for the transfer to complete with `sps2WaitForDMA`
10. Free the memory with `sps2Free`
11. Shut down the SPS2 device with `sps2Release` or `_sps2Close`

The example above performs all of these steps except for #5 because it performs a normal mode transfer.

Also, note that there are a number of unions defined for some of the registers is `sps2registers.h` for your convenience. As an example, this is the union for the `Dn_CHCR` registers.

```
typedef union Dn_CHCR {
  sps2uint32 i32;

  struct {
    unsigned int DIR     :  1;
    unsigned int _PAD1   :  1;
    unsigned int MOD     :  2;
    unsigned int ASP     :  2;
    unsigned int TTE     :  1;
    unsigned int TIE     :  1;
    unsigned int STR     :  1;
    unsigned int _PAD2   : 10;
    unsigned int TAG_PCE :  2;
    unsigned int TAG_ID  :  3;
    unsigned int TAG_IRQ :  1;
  } s;
} Dn_CHCR_t;
```


Accessing the Emotion Engine Registers

In order to access the Emotion Engine registers directly using SPS2, a developer may use one of the two methods outlined below. Note that because the FIFO registers are 128 bits in length and they must be read from/written to all at once, SPS2 provides access functions instead of 128 bit pointers. This is similar to the 64 bit Graphics Synthesizer registers.
The Emotion Engine register macros are defined in the file `sps2registers.h`

Method 1 Using the *SPS2 Core Function Set*:

1. Include `sps2lib.h`
2. Initialize the SPS2 device with `sps2Init`.
3. Access the Emotion Engine registers by using the pointers listed below.
4. Release the SPS2 device with `sps2Release` (or just exit the application).

Timer

```
EE_T0_COUNT
EE_T0_MODE
EE_T0_COMP
EE_T0_HOLD

EE_T1_COUNT
EE_T1_MODE
EE_T1_COMP
EE_T1_HOLD

EE_T2_COUNT
EE_T2_MODE
EE_T2_COMP

EE_T3_COUNT
EE_T3_MODE
EE_T3_COMP
```

IPU

```
EE_IPU_CMD
EE_IPU_CTRL
EE_IPU_BP
EE_IPU_TOP
```

GIF

```
EE_GIF_CTRL
EE_GIF_MODE
EE_GIF_STAT
EE_GIF_TAG0
EE_GIF_TAG1
EE_GIF_TAG2
EE_GIF_TAG3
EE_GIF_CNT
EE_GIF_P3CNT
EE_GIF_P3TAG
```

VIF0

```
EE_VIF0_STAT
EE_VIF0_FBRST
EE_VIF0_ERR
EE_VIF0_MARK
EE_VIF0_CYCLE
EE_VIF0_MODE
EE_VIF0_NUM
EE_VIF0_MASK
EE_VIF0_CODE
EE_VIF0_ITOPS
EE_VIF0_ITOP
EE_VIF0_R0
EE_VIF0_R1
EE_VIF0_R2
EE_VIF0_R3
EE_VIF0_C0
EE_VIF0_C1
EE_VIF0_C2
EE_VIF0_C3
```

VIF1

```
EE_VIF1_STAT
EE_VIF1_FBRST
EE_VIF1_ERR
EE_VIF1_MARK
EE_VIF1_CYCLE
EE_VIF1_MODE
EE_VIF1_NUM
EE_VIF1_MASK
EE_VIF1_CODE
EE_VIF1_ITOPS
EE_VIF1_BASE
EE_VIF1_OFST
EE_VIF1_TOPS
EE_VIF1_ITOP
EE_VIF1_TOP
EE_VIF1_R0
EE_VIF1_R1
EE_VIF1_R2
EE_VIF1_R3
EE_VIF1_C0
EE_VIF1_C1
EE_VIF1_C2
EE_VIF1_C3
```

FIFO

```
DPUT_EE_VIF0_FIFO(val)
DPUT_EE_VIF1_FIFO(val)
DGET_EE_VIF1_FIFO(val)
DPUT_EE_GIF_FIFO(val)
DGET_EE_IPU_out_FIFO
DPUT_EE_IPU_in_FIFO(val)
```

DMAC

```
EE_D0_CHCR
EE_D0_MADR
EE_D0_QWC
EE_D0_TADR
EE_D0_ASR0
EE_D0_ASR1

EE_D1_CHCR
EE_D1_MADR
EE_D1_QWC
EE_D1_TADR
EE_D1_ASR0
EE_D1_ASR1

EE_D2_CHCR
EE_D2_MADR
EE_D2_QWC
EE_D2_TADR
EE_D2_ASR0
EE_D2_ASR1

EE_D3_CHCR
EE_D3_MADR
EE_D3_QWC

EE_D4_CHCR
EE_D4_MADR
EE_D4_QWC
EE_D4_TADR

EE_D5_CHCR
EE_D5_MADR
EE_D5_QWC

EE_D6_CHCR
EE_D6_MADR
EE_D6_QWC
EE_D6_TADR

EE_D7_CHCR
EE_D7_MADR
EE_D7_QWC

EE_D8_CHCR
EE_D8_MADR
```



```
EE_D8_QWC
EE_D8_SADR

EE_D9_CHCR
EE_D9_MADR
EE_D9_QWC
EE_D9_TADR
EE_D9_SADR
```

```
EE_D_CTRL
EE_D_STAT
EE_D_PCR
EE_D_SQWC
EE_D_RBSR
EE_D_RBOR
EE_D_STADR
EE_D_ENABLER
EE_D_ENABLEW
```

INTC

```
EE_I_STAT
EE_I_MASK
```

SIF

```
EE_SB_SMFLG
```

Method 2 Using the *SPS2 Extended Function Set*:

1. Include `sps2lib.h`
2. Open the SPS2 device with `_sps2Open`
3. Obtain a base pointer to the Emotion Engine registers with `_sps2MapEERegisters`
4. Access the Emotion Engine registers by using the functions below with the base pointer:
5. Close the SPS2 device with `_sps2Close` (or just exit the application).

Timer

```
EE_T0_COUNT_OFF(base)
EE_T0_MODE_OFF(base)
EE_T0_COMP_OFF(base)
EE_T0_HOLD_OFF(base)

EE_T1_COUNT_OFF(base)
EE_T1_MODE_OFF(base)
EE_T1_COMP_OFF(base)
EE_T1_HOLD_OFF(base)

EE_T2_COUNT_OFF(base)
EE_T2_MODE_OFF(base)
EE_T2_COMP_OFF(base)

EE_T3_COUNT_OFF(base)
EE_T3_MODE_OFF(base)
EE_T3_COMP_OFF(base)
```

IPU

```
EE_IPU_CMD_OFF(base)
EE_IPU_CTRL_OFF(base)
EE_IPU_BP_OFF(base)
EE_IPU_TOP_OFF(base)
```

GIF

```
EE_GIF_CTRL_OFF(base)
EE_GIF_MODE_OFF(base)
EE_GIF_STAT_OFF(base)
EE_GIF_TAG0_OFF(base)
EE_GIF_TAG1_OFF(base)
EE_GIF_TAG2_OFF(base)
EE_GIF_TAG3_OFF(base)
EE_GIF_CNT_OFF(base)
EE_GIF_P3CNT_OFF(base)
EE_GIF_P3TAG_OFF(base)
```

VIF0

```
EE_VIF0_STAT_OFF(base)
EE_VIF0_FBRST_OFF(base)
EE_VIF0_ERR_OFF(base)
EE_VIF0_MARK_OFF(base)
EE_VIF0_CYCLE_OFF(base)
EE_VIF0_MODE_OFF(base)
EE_VIF0_NUM_OFF(base)
EE_VIF0_MASK_OFF(base)
EE_VIF0_CODE_OFF(base)
EE_VIF0_ITOPS_OFF(base)
EE_VIF0_ITOP_OFF(base)
EE_VIF0_R0_OFF(base)
EE_VIF0_R1_OFF(base)
EE_VIF0_R2_OFF(base)
EE_VIF0_R3_OFF(base)
EE_VIF0_C0_OFF(base)
EE_VIF0_C1_OFF(base)
EE_VIF0_C2_OFF(base)
EE_VIF0_C3_OFF(base)
```

VIF1

```
EE_VIF1_STAT_OFF(base)
EE_VIF1_FBRST_OFF(base)
EE_VIF1_ERR_OFF(base)
EE_VIF1_MARK_OFF(base)
EE_VIF1_CYCLE_OFF(base)
EE_VIF1_MODE_OFF(base)
EE_VIF1_NUM_OFF(base)
EE_VIF1_MASK_OFF(base)
EE_VIF1_CODE_OFF(base)
EE_VIF1_ITOPS_OFF(base)
EE_VIF1_BASE_OFF(base)
EE_VIF1_OFST_OFF(base)
EE_VIF1_TOPS_OFF(base)
EE_VIF1_ITOP_OFF(base)
EE_VIF1_TOP_OFF(base)
EE_VIF1_R0_OFF(base)
EE_VIF1_R1_OFF(base)
EE_VIF1_R2_OFF(base)
EE_VIF1_R3_OFF(base)
EE_VIF1_C0_OFF(base)
EE_VIF1_C1_OFF(base)
EE_VIF1_C2_OFF(base)
EE_VIF1_C3_OFF(base)
```

FIFO

```
DPUT_EE_VIF0_FIFO_OFF(
base, val)
DPUT_EE_VIF1_FIFO_OFF(
base, val)
DGET_EE_VIF1_FIFO_OFF(
base, val)
DPUT_EE_GIF_FIFO_OFF(b
ase, val)
DGET_EE_IPU_out_FIFO_O
FF(base)
DPUT_EE_IPU_in_FIFO_OF
F(base, val)
```

DMAC

```
EE_D0_CHCR_OFF(base)
EE_D0_MADR_OFF(base)
EE_D0_QWC_OFF(base)
EE_D0_TADR_OFF(base)
EE_D0_ASR0_OFF(base)
EE_D0_ASR1_OFF(base)

EE_D1_CHCR_OFF(base)
EE_D1_MADR_OFF(base)
EE_D1_QWC_OFF(base)
EE_D1_TADR_OFF(base)
EE_D1_ASR0_OFF(base)
EE_D1_ASR1_OFF(base)

EE_D2_CHCR_OFF(base)
EE_D2_MADR_OFF(base)
EE_D2_QWC_OFF(base)
EE_D2_TADR_OFF(base)
EE_D2_ASR0_OFF(base)
EE_D2_ASR1_OFF(base)
```



```
EE_D3_CHCR_OFF(base)
EE_D3_MADR_OFF(base)
EE_D3_QWC_OFF(base)

EE_D4_CHCR_OFF(base)
EE_D4_MADR_OFF(base)
EE_D4_QWC_OFF(base)
EE_D4_TADR_OFF(base)

EE_D5_CHCR_OFF(base)
EE_D5_MADR_OFF(base)
EE_D5_QWC_OFF(base)

EE_D6_CHCR_OFF(base)
EE_D6_MADR_OFF(base)
EE_D6_QWC_OFF(base)
EE_D6_TADR_OFF(base)
```

```
EE_D7_CHCR_OFF(base)
EE_D7_MADR_OFF(base)
EE_D7_QWC_OFF(base)

EE_D8_CHCR_OFF(base)
EE_D8_MADR_OFF(base)
EE_D8_QWC_OFF(base)
EE_D8_SADR_OFF(base)

EE_D9_CHCR_OFF(base)
EE_D9_MADR_OFF(base)
EE_D9_QWC_OFF(base)
EE_D9_TADR_OFF(base)
EE_D9_SADR_OFF(base)

EE_D_CTRL_OFF(base)
```

```
EE_D_STAT_OFF(base)
EE_D_PCR_OFF(base)
EE_D_SQWC_OFF(base)
EE_D_RBSR_OFF(base)
EE_D_RBOR_OFF(base)
EE_D_STADR_OFF(base)
EE_D_ENABLER_OFF(base)
EE_D_ENABLEW_OFF(base)
```

INTC

```
EE_I_STAT_OFF(base)
EE_I_MASK_OFF(base)
```

SIF

```
EE_SB_SMFLG_OFF(base)
```

Accessing the Graphics Synthesizer Registers

In order to access the Graphics Synthesizer registers directly using SPS2, a developer may use one of the two methods outlined below. Note that unlike the Emotion Engine registers and the Scratch Pad and Vertex Unit memories, the Graphics Synthesizer registers cannot be accessed simply by assigning a value to the appropriate pointer.

For example, one would expect to set the background color in the following manner:

```
*GS_BGCOLOR=0xbbggrr;
```

but instead one must set the color in the following manner:

```
DPUT_GS_BGCOLOR(0xbbggrr);
```

The reason for this is that all Graphics Synthesizer registers are 64 bits in length. Unfortunately, regardless of the pointer prototype, issuing a `*GS_BGCOLOR` results in two separate store functions to store 32 bits at a time. This is a problem because with each store the Emotion Engine will copy the value to the Graphics Synthesizer. This means that when the lower 32 bits are stored they are sign extended to 64 bits and transferred regardless of the intended upper 32 bits. If the application is compiled to MIPS 3 standards so that the store produced is a single 64 bit store, the code ends up being incompatible with the other libraries on the PS2 Linux system. The DPUT macros use some inline assembly to ensure that all 64 bits are store correctly in one write.

The DPUT macros for the Graphics Synthesizer registers are all defined in `sps2registers.h`.

Method 1 Using the *SPS2 Core Function Set*:

5. Include `sps2lib.h`
6. Initialize the SPS2 device with `sps2Init`.
7. Set the Graphics Synthesizer registers with the following macros:
   ```
   DPUT_GS_PMODE(value)
   DPUT_GS_SMODE1(value)
   DPUT_GS_SMODE2(value)
   DPUT_GS_SRFSH(value)
   DPUT_GS_SYNCH1(value)
   DPUT_GS_SYNCH2(value)
   DPUT_GS_SYNCV(value)
   DPUT_GS_DISPFB1(value)
   DPUT_GS_DISPLAY1(value)
   DPUT_GS_DISPFB2(value)
   DPUT_GS_DISPLAY2(value)
   DPUT_GS_EXTBUF(value)
   DPUT_GS_EXTDATA(value)
   DPUT_GS_EXTWRITE(value)
   DPUT_GS_BGCOLOR(value)
   DPUT_GS_CSR(value)
   DPUT_GS_IMR(value)
   DPUT_GS_BUSDIR(value)
   DPUT_GS_SIGBLID(value)
   ```
8. Release the SPS2 device with `sps2Release` (or just exit the application).

Method 2 Using the *SPS2 Extended Function Set*:

1. Include `sps2lib.h`
2. Open the SPS2 device with `_sps2Open`
3. Obtain a base pointer to the Graphics Synthesizer registers with `_sps2MapGSRegisters`
4. Set the Graphics Synthesizer registers by using the following macros with the base pointer:

```
DPUT_GS_PMODE_OFF(base pointer, value)
DPUT_GS_SMODE1_OFF(base pointer, value)
DPUT_GS_SMODE2_OFF(base pointer, value)
DPUT_GS_SRFSH_OFF(base pointer, value)
DPUT_GS_SYNCH1_OFF(base pointer, value)
DPUT_GS_SYNCH2_OFF(base pointer, value)
DPUT_GS_SYNCV_OFF(base pointer, value)
DPUT_GS_DISPFB1_OFF(base pointer, value)
DPUT_GS_DISPLAY1_OFF(base pointer, value)
DPUT_GS_DISPFB2_OFF(base pointer, value)
DPUT_GS_DISPLAY2_OFF(base pointer, value)
DPUT_GS_EXTBUF_OFF(base pointer, value)
DPUT_GS_EXTDATA_OFF(base pointer, value)
DPUT_GS_EXTWRITE_OFF(base pointer, value)
DPUT_GS_BGCOLOR_OFF(base pointer, value)
DPUT_GS_CSR_OFF(base pointer, value)
DPUT_GS_IMR_OFF(base pointer, value)
DPUT_GS_BUSDIR_OFF(base pointer, value)
DPUT_GS_SIGBLID_OFF(base pointer, value)
```

5. Close the SPS2 device with `_sps2Close` (or just exit the application).

Accessing the Scratch Pad Memory

In order to access the Scratch Pad memory directly using SPS2, a developer may use one of the two methods outlined below.

`SCRATCH_PAD` and `SCRATCH_PAD_OFF` are defined in `sps2scratchpad.h`.

Method 1 Using the *SPS2 Core Function Set*:

1. Include `sps2lib.h`
2. Initialize the SPS2 device with `sps2Init`.
3. Access the Scratch Pad memory by using the `SCRATCH_PAD` pointer.
4. Release the SPS2 device with `sps2Release` (or just exit the application).

Method 2 Using the *SPS2 Extended Function Set*:

1. Include `sps2lib.h`
2. Open the SPS2 device with `_sps2Open`
3. Obtain a base pointer to the Scratch Pad memory with `_sps2MapScratchPad`
4. Access the Scratch Pad memory by using the `SCRATCH_PAD_OFF(base pointer)` function with the base pointer
5. Close the SPS2 device with `_sps2Close` (or just exit the application).

Accessing the Vertex Unit Memories

In order to access the Vertex Unit memories directly using SPS2, a developer may use one of the two methods outlined below.

The Vertex Unit functions and pointers are defined in `sps2vumemory.h`.

Method 1 Using the *SPS2 Core Function Set*:

1. Include `sps2lib.h`
2. Initialize the SPS2 device with `sps2Init`.
3. Access the Vertex Unit memories by using the following pointers:
   `VU0_MEM`
   `VU0_MICRO_MEM`
   `VU1_MEM`
   `VU1_MICRO_MEM`
4. Release the SPS2 device with `sps2Release` (or just exit the application).

Method 2 Using the *SPS2 Extended Function Set*:

1. Include `sps2lib.h`
2. Open the SPS2 device with `_sps2Open`
3. Obtain a base pointer to the Vertex Unit memories with `_sps2MapVUMemory`
4. Access the Vertex Unit memories by using the following functions with the base pointer:
   `VU0_MEM_OFF(base pointer)`
   `VU0_MICRO_MEM_OFF(base pointer)`
   `VU1_MEM_OFF(base pointer)`
   `VU1_MICRO_MEM_OFF(base pointer)`
5. Close the SPS2 device with `_sps2Close` (or just exit the application).

SPS2 Core Function Set Reference

In the next pages, the instructions that constitute the *SPS2 Core Function Set* will be outlined. Developers are encouraged to use only the functions in the core function set as much as possible. They provide a good set of default actions for extreme convenience and ease of development while minimizing the amount of overhead they introduce to an application.

sps2Init

Prototype:

```
static inline int sps2Init();
```

Parameters:

None

Return Value:

- If successful, a descriptor to the SPS2 device (>=0)
- If unsuccessful, an error number <0

See Also:

`_sps2Open, sps2Release`

Comments:

This function gains access to the SPS2 kernel module. For developers using the *SPS2 Core Function Set* only, it should be the first function invoked. It performs a number of functions:

1. It connects to the SPS2 device
2. It ensures that the SPS2 device supports the current version and hasn't been already opened. If it has been already opened, it merely duplicates the descriptor from the previous open. This has important ramifications which are outlined below.
3. It maps the Emotion Engine (EE) registers to `SPS2_EE_REGISTERS_START` and aborts the application if this is not possible. This allows programmers to be sure that, if `sps2Init` returns, the EE registers ***will be*** mapped starting at `SPS2_EE_REGISTERS_START`. This allows programmers to write applications that use the fixed pointers to the EE registers defined in `sps2registers.h` such as `EE_D9_CHCR`.
4. It maps the Graphics Synthesizer (GS) registers to `SPS2_GS_REGISTERS_START` allowing, like in #3, developers to use fixed functions such as `DPUT_GS_BGCOLOR`.
5. It maps the scratch pad memory to `SPS2_SCRATCH_PAD_START` allowing, like in #3, developers to use fixed pointers such as `SCRATCH_PAD`.
6. It maps the Vertex Unit (VU) memory to `SPS2_VU_MEMORY_START` allowing, once again as in #3, developers to access fixed pointers such as `VU0_MICRO_MEM`.

In order for #3-#6 to succeed, the virtual memory area 0x00010000-0x0004ffff must be free once your application is loaded. For most normal applications, this should not be a problem, however, developers with special link scripts may need to adjust their scripts to ensure this memory area is free. In the unlikely event that this is not possible, `sps2Init` cannot be used and developers are directed to `_sps2Open` in the *SPS2 Extended Function Set*.

Because of step #2, multiple instances of sps2Open end up sharing the same resources, even though they are assigned different descriptors. This means:

- Memory allocated by `sps2Allocate` will not be released until the all of the descriptors have been closed up using `sps2Release` *unless* the memory is explicitly released with `sps2Free`. Basically this means that you shouldn't assume that `sps2Release` will free up all your memory, you should explicitly free up all your allocations instead.
- The Emotion Engine and Graphics Synthesizer registers as well as the Scratch Pad and Vertex Unit memories won't go away just because one descriptor is closed. They will only go away once all of the descriptors have been closed. This is good because a module that chooses to issue `sps2Init` then `sps2Release` need not worry that by releasing it's descriptor it will cause the application to stop working by releasing the fixed pointers.

sps2Release

Prototype:

```
static inline int sps2Release(int iSPS2Device);
```

Parameters:

iSPS2Device – An SPS2 device descriptor returned by `sps2Init`

Return Value:

None

See Also:

`sps2Init`

Comments:

This function releases an SPS2 device descriptor. If it is the last SPS2 device descriptor being released, then it frees up all memory that hasn't been explicitly freed, and unmaps the Emotion Engine and Graphics Synthesizer registers as well as the Scratch Pad and Vertex Unit memories from `SPS2_EE_REGISTERS_START`, `SPS2_GS_REGISTERS_START`, `SPS2_SCRATCH_PAD_START` and `SPS2_VU_MEMORY_START` respectively.

Caution

All of the `sps2Memory_t` structures, however are not freed (only the actual memory they describe) which could cause a memory leak unless the developer explicitly frees up the memory by using `sps2Free`.

sps2Allocate

Prototype:

```
static inline sps2Memory_t *sps2Allocate(unsigned long ulSize,
                                         int iMapOptions,
                                         int iDeviceHandle);
```

Parameters:

ulSize -- The number of bytes to allocate, which will be rounded up by the block size

iMapOptions – One of:

`SPS2_MAP_BLOCK_4K` to map memory in 4K increments,
`SPS2_MAP_BLOCK_8K` to map memory in 8K increments,
`SPS2_MAP_BLOCK_16K` to map memory in 16K increments,
`SPS2_MAP_BLOCK_32K` to map memory in 32K increments,
`SPS2_MAP_BLOCK_64K` to map memory in 64K increments,
`SPS2_MAP_BLOCK_128K` to map memory in 128K increments

bitwise-ORed with one of:

`SPS2_MAP_CACHED` to allow memory be cached (the default value),
`SPS2_MAP_UNCACHED` to allocate this memory as uncached

iDeviceHandle – An SPS2 device descriptor returned by `sps2Init or _sps2Open`

Return Value:

- On success, an sps2Memory_t structure
- On failure, null, most likely due to insufficient memory

See Also:

`sps2Init, _sps2Open, sps2Free, sps2FlushCache, sps2GetPhysicalAddress`

Comments:

THE ONLY BLOCK SIZE SUPPORTED WITH THIS RELEASE IS 4K.

Memory returned by `sps2Allocate` is only physically contiguous in increments of the block size. This means that if you allocate 8K in 4K blocks, bytes 0-4095 will be physically contiguous and bytes 4096-8191 will be physically contiguous. **You should NOT treat it as a single 8K chunk**! Whereas it is okay to do so while populating the data (e.g. reading in a 8K texture from a file into the memory), when you use it to perform DMA transfers you will have to treat it as a sequence of 2 consecutive 4K chunks.

Remember, the whole allocation is contiguous in virtual space, but only individual chunks are contiguous in physical space. Your application understands virtual space, the DMA controller understands physical space. Also, for future versions, when you try to allocate larger increments, remember that there is less of a chance that your allocation will succeed.

Memory allocated with `sps2Allocate` is freed with `sps2Free`.

To get the virtual address of your memory, use the `pvStart` field in the `sps2Memory_t` structure that is returned. ***`pvStart` IS THE ONLY FIELD OF INTEREST TO DEVELOPERS.***

To get the physical address of any offset within your memory, use the `sps2GetPhysicalAddress` function.

If you are using cacheable memory and would like to flush it in order to start a DMA transfer, use `sps2FlushCache`.

sps2Free

Prototype:

```
static inline void sps2Remap(sps2Memory_t *pMapping);
```

Parameters:

pMapping – An sps2Memory_t structure returned by `sps2Allocate` or `sps2Remap`

Return Value:

None

See Also:

`sps2Init, _sps2Open, sps2Allocate, sps2Remap`

Comments:

This function releases memory allocated through `sps2Allocate`. If the memory has been remapped one or more times by `sps2Remap` then this will only release the memory once all mappings (including the original one) have been freed.

It also releases the memory associated with the `sps2Memory_t` structure.

sps2Remap

Prototype:

```
static inline sps2Memory_t *sps2Remap(sps2Memory_t *pOriginalArea,
                                      int iMapOptions,
                                      int iDeviceHandle);
```

Parameters:

pOriginalArea – An sps2Memory_t structure returned by `sps2Allocate` or `sps2Remap`

iMapOptions – One of:

`SPS2_MAP_BLOCK_4K` to map memory in 4K increments,
`SPS2_MAP_BLOCK_8K` to map memory in 8K increments,
`SPS2_MAP_BLOCK_16K` to map memory in 16K increments,
`SPS2_MAP_BLOCK_32K` to map memory in 32K increments,
`SPS2_MAP_BLOCK_64K` to map memory in 64K increments,
`SPS2_MAP_BLOCK_128K` to map memory in 128K increments

bitwise-ORed with one of:

`SPS2_MAP_CACHED` to allow memory be cached (the default value),
`SPS2_MAP_UNCACHED` to allocate this memory as uncached

iDeviceHandle – An SPS2 device descriptor returned by `sps2Init or _sps2Open`

Return Value:

- On success, an sps2Memory_t structure
- On failure, null, most likely due to insufficient memory

See Also:

`sps2Init, _sps2Open, sps2Allocate, sps2Free`

Comments:

This function allows you to "remap" an area allocated through `sps2Allocate`. The primary reason for doing this is to allow the programmer to have both cached and uncached pointers to the same area of memory (i.e. use `sps2Allocate` with `SPS2_MAP_CACHED` then use `sps2Remap` with `SPS2_MAP_UNCACHED`). The block size should be the same as the one used with `sps2Allocate`.

`sps2Free` will only release the memory once all mappings of an area have been freed.

Caution:

Developers are cautioned to be very careful when using both cached and uncached pointers to the same area of memory. If within a single cache frame the memory is accessed both cached and uncached without an intermediate `sps2FlushCache`, the system can crash.

sps2GetPhysicalAddress

Prototype:

```
static inline unsigned long sps2GetPhysicalAddress(void *pvAddress,
                                                   sps2Memory_t *pDescriptor);
```

Parameters:

pvAddress – The virtual address for which to retrieve a physical address

pDescriptor – The sps2Memory_t structure returned by `sps2Allocate` or `sps2Remap` that corresponds to this pointer.

Return Value:

- On success, returns the physical address of `pvAddress`
- On failure, the application is terminated.

See Also:

`sps2Allocate, sps2Remap`

Comments:

This function gives the physical address for a virtual address. This is important because in order to perform a DMA transfer, the DMA controller needs to be given a physical address.

The reason this function terminates the application if a bad pointer or descriptor is passed in is that it is very likely that the function is called immediately before a DMA transfer. This will prevent the developer from accidentally passing an error return value to the DMA controller as the address and causing the system to crash. Basically, if this function were to return an error code, the program is already sufficiently broken to warrant an exit.

sps2FlushCache

Prototype:

```
static inline int sps2FlushCache(int iDeviceHandle);
```

Parameters:

iDeviceHandle – An SPS2 device descriptor returned by `sps2Init or _sps2Open`

Return Value:

- zero on success
- non-zero if an invalid device handle was specified

See Also:

`sps2Init, _sps2Open`

Comments:

This function flushes all caches. This is beneficial if the memory returned by `sps2Allocate` or `sps2Remap` was being cached. Flushing the cache allows DMA transfers to properly transfer all the contents of the memory.

sps2WaitForDMA

Prototype:

```
static inline int sps2WaitForDMA(int iChannel,
                                 int iDeviceHandle);
```

Parameters:

iChannel - the DMA channel to wait for.
iDeviceHandle – An SPS2 device descriptor returned by `sps2Init or _sps2Open`

Return Value:

- zero on success
- non-zero if an invalid device handle or DMA channel was specified

See Also:

`sps2Init, _sps2Open`

Comments:

This function allows the scheduler to run other applications on the system while a transfer is in progress. If your application is time critical and must continue the **instant** that the DMA transfer has ended, then you should consider creating a spinlock loop instead.

This function returns after the CHCR register corresponding to the channel has the STR bit cleared.

SPS2 Extended Function Set Reference

The *SPS2 Extended Function Set* is a superset of the *SPS2 Core Function Set*. It includes all of the functions in the core function set as well as a few more. There are two major differences between the core and extended function sets.

First, the core function set contains a function called `sps2Init`. This function opens up the SPS2 Kernel Module and prepares a number of preset mappings for the Emotion Engine and Graphics Synthesizer registers as well as the Scratch Pad and Vertex Unit memories. Because these preset mappings use a fixed location, applications using these registers or memories can use predetermined pointers to achieve maximum efficiency while minimizing the difference between application development within the Linux environment and development within the native PS2 environment.

The extended function set defines a simpler function called `_sps2Open` that performs less initialization than `sps2Init`. Specifically, `_sps2Open` does not map the registers and memories as `sps2Init` does, but instead, the *SPS2 Extended Function Set* provides a number of additional functions to map these registers and memories to any location the developer desires. Whereas this provides slightly more flexibility to the developer, there is a tradeoff. Because the developer does not know in advance exactly where these registers and memories will be mapped to, they must use offset functions to access the individual registers within the memory mapping (as opposed to predefined pointers that are made available in the *SPS2 Core Function Set*).

To illustrate the example, consider the following two code segments that are intended to set the value of the `EE_D0_QWC` register to 12.

First, using the *SPS2 Core Function Set*:

```
int iSPS2Device=sps2Init();
*EE_D0_QWC=0;
sps2Release(iSPS2Device);
```

Now, using the *SPS2 Extended Function Set*

```
int iSPS2Device=_sps2Open();
void *pvEERegisters=_sps2MapEERegisters(0,iSPS2Device);
*EE_D0_QWC_OFF  (pvEERegisters)=0;
_sps2Close(iSPS2Device);
```

In some obscure situations (where custom linking is used for the application), it is possible for the first example to fail. Typically the developer would be able to modify their link script so that that would not become an issue. On the other hand, the second example would work even with the most obscure link scripts, but the developer would now need to distribute the "base pointer" `pvEERegisters` throughout the application, possibly by defining it as a global variable.

The second difference between the function sets is that the device handles in the *SPS2 Extended Function Set* are not shared. This means that if a process opens the device, allocates some memory (or maps some of the registers) and then closes the devices, the allocations and mappings are freed. This is true even if the device was opened multiple times before being closed. On the other hand, with the core set, if the device is initialized multiple times with `sps2Init`, none of the resources are released until all instances are closed. The prior situation makes sense for a developer of an independent module who wishes to create a library of

functions or classes that work independently of the rest of the system. The latter example makes more sense for most developers because they can open and close the module within their application repeatedly without having to worry about accidentally unmapping some registers being used by another function.

_sps2Open

Prototype:

```
static inline int sps2Open();
```

Parameters:

None

Return Value:

- If successful, a descriptor to the SPS2 device (>=0)
- If unsuccessful, an error number <0

See Also:

`sps2Init, _sps2Close`

Comments:

This function gains access to the SPS2 kernel module.

1. It connects to the SPS2 device
2. It ensures that the SPS2 device supports the current version.

_sps2Close

Prototype:

```
static inline int _sps2Close(int iSPS2Device);
```

Parameters:

iSPS2Device – An SPS2 device descriptor returned by `_sps2Open`

Return Value:

None

See Also:

`_sps2Open`

Comments:

This function releases an SPS2 device descriptor. This will release the memory that has been allocated and unmap any of the Emotion Engine and Graphics Synthesizer registers as well as Scratch Pad and Vertex Unit memories that may have been mapped.

Caution

All of the `sps2Memory_t` structures, however are not freed (only the actual memory they describe) which could cause a memory leak unless the developer explicitly frees up the memory by using `sps2Free`.

_sps2MapEERegisters

Prototype:

```
static inline int _sps2MapEERegisters(void *pvWhere,
                                      int iSPS2Device);
```

Parameters:

pvWhere – The desired location at which to map the registers (can be zero). If this location is not suitable, the library will map the registers at an appropriate location

iSPS2Device – An SPS2 device descriptor returned by `_sps2Open` or by `sps2Init`

Return Value:

- On success, the base pointer of the address at which the registers have been mapped
- On failure, MAP_FAILED (-1)

See Also:

`_sps2Open`

Comments:

This function attempts to map the Emotion Engine registers at a specified location. If the location is zero or unsuitable for the mapping, the function will map the registers elsewhere and indicate, through its return value, where they have been mapped.

There are a number of preprocessor macros defined in `sps2registers.h` that can be used to determine the exact location of a specific Emotion Engine register relative to the base pointer.

These macros are in the form of `EE_*_OFF(baase pointer)`, such as `EE_D0_CHCR_OFF`(`base pointer`). The value of `base pointer` that should be passed in is the return value of this function.

These registers will be unmapped once the device handle has been closed with `_sps2Close`.

_sps2MapGSRegisters

Prototype:

```
static inline int _sps2MapGSRegisters(void *pvWhere,
                                      int iSPS2Device);
```

Parameters:

pvWhere – The desired location at which to map the registers (can be zero). If this location is not suitable, the library will map the registers at an appropriate location

iSPS2Device – An SPS2 device descriptor returned by `_sps2Open` or by `sps2Init`

Return Value:

- On success, the base pointer of the address at which the registers have been mapped
- On failure, MAP_FAILED (-1)

See Also:

`_sps2Open`

Comments:

This function attempts to map the Graphics Synthesizer registers at a specified location. If the location is zero or unsuitable for the mapping, the function will map the registers elsewhere and indicate, through its return value, where they have been mapped.

There are a number of preprocessor macros defined in `sps2registers.h` that can be used to determine the exact location of a specific Graphics Synthesizer register relative to the base pointer.

These macros are in the form of `DPUT_GS_*_OFF(base pointer, value)`, such as `DPUT_GS_BGCOLOR_OFF(base pointer, value)`. The value of `base pointer` that should be passed in is the return value of this function.

These registers will be unmapped once the device handle has been closed with `_sps2Close`.

_sps2MapVUMemory

Prototype:

```
static inline int _sps2MapVUMemory(void *pvWhere,
                                   int iSPS2Device);
```

Parameters:

pvWhere – The desired location at which to map the memory (can be zero). If this location is not suitable, the library will map the memory at an appropriate location

iSPS2Device – An SPS2 device descriptor returned by `_sps2Open` or by `sps2Init`

Return Value:

- On success, the base pointer of the address at which the Vertex Unit memories have been mapped
- On failure, MAP_FAILED (-1)

See Also:

`_sps2Open`

Comments:

This function attempts to map the Vertex Unit memories at a specified location. If the location is zero or unsuitable for the mapping, the function will map the memories elsewhere and indicate, through its return value, where they have been mapped.

There are a number of preprocessor macros defined in `sps2vumemory.h` that can be used to determine the exact location of a specific Vertex Unit memory relative to the base pointer.

These macros are `VU0_MEM_OFF(base pointer)`, `VU0_MICRO_MEM_OFF(base pointer)`, `VU1_MEM_OFF(base pointer)` and `VU1_MICRO_MEM_OFF(base pointer)`. The value of `base pointer` that should be passed in is the return value of this function.

These memories will be unmapped once the device handle has been closed with `_sps2Close`.

_sps2MapScratchPad

Prototype:

```
static inline int _sps2MapScratchPad(void *pvWhere,
                                     int iSPS2Device);
```

Parameters:

pvWhere – The desired location at which to map the memory (can be zero). If this location is not suitable, the library will map the memory at an appropriate location

iSPS2Device – An SPS2 device descriptor returned by `_sps2Open` or by `sps2Init`

Return Value:

- On success, the base pointer of the address at which the Scratch Pad memory has been mapped
- On failure, MAP_FAILED (-1)

See Also:

`_sps2Open`

Comments:

This function attempts to map the Scratch Pad memory at a specified location. If the location is zero or unsuitable for the mapping, the function will map the memory elsewhere and indicate, through its return value, where they have been mapped.

There is a preprocessor macro defined in `sps2scratchpad.h` that is offered for compatibility with the functionality of the other `_sps2Map*` functions.

This macro is called `SCRATCH_PAD_OFF(base pointer)`. The value of `base pointer` that should be passed in is the return value of this function. Since there is only one scratch pad memory this function does nothing except for return the `base pointer`.

The memory will be unmapped once the device handle has been closed with `_sps2Close`.

Index

EE_D0_TADR, 13
EE_D0_TADR_OFF, 14
EE_D1_ASR0, 13
EE_D1_ASR0_OFF, 14
EE_D1_ASR1, 13
EE_D1_ASR1_OFF, 14
EE_D1_CHCR, 13
EE_D1_CHCR_OFF, 14
EE_D1_MADR, 13
EE_D1_MADR_OFF, 14
EE_D1_QWC, 13
EE_D1_QWC_OFF, 14
EE_D1_TADR, 13
EE_D1_TADR_OFF, 14
EE_D2_ASR0, 13
EE_D2_ASR0_OFF, 14
EE_D2_ASR1, 13
EE_D2_ASR1_OFF, 14
EE_D2_CHCR, 13
EE_D2_CHCR_OFF, 14
EE_D2_MADR, 13
EE_D2_MADR_OFF, 14
EE_D2_QWC, 13
EE_D2_QWC_OFF, 14
EE_D2_TADR, 13
EE_D2_TADR_OFF, 14
EE_D3_CHCR, 13
EE_D3_CHCR_OFF, 15
EE_D3_MADR, 13
EE_D3_MADR_OFF, 15
EE_D3_QWC, 13
EE_D3_QWC_OFF, 15
EE_D4_CHCR, 13
EE_D4_CHCR_OFF, 15
EE_D4_MADR, 13
EE_D4_MADR_OFF, 15
EE_D4_QWC, 13
EE_D4_QWC_OFF, 15
EE_D4_TADR, 13
EE_D4_TADR_OFF, 15
EE_D5_CHCR, 13
EE_D5_CHCR_OFF, 15
EE_D5_MADR, 13
EE_D5_MADR_OFF, 15
EE_D5_QWC, 13
EE_D5_QWC_OFF, 15
EE_D6_CHCR, 13
EE_D6_CHCR_OFF, 15
EE_D6_MADR, 13
EE_D6_MADR_OFF, 15
EE_D6_QWC, 13
EE_D6_QWC_OFF, 15
EE_D6_TADR, 13
EE_D6_TADR_OFF, 15
EE_D7_CHCR, 13
EE_D7_CHCR_OFF, 15
EE_D7_MADR, 13
EE_D7_MADR_OFF, 15
EE_D7_QWC, 13
EE_D7_QWC_OFF, 15
EE_D8_CHCR, 13
EE_D8_CHCR_OFF, 15
EE_D8_MADR, 13
EE_D8_MADR_OFF, 15
EE_D8_QWC, 14
EE_D8_QWC_OFF, 15
EE_D8_SADR, 14
EE_D8_SADR_OFF, 15
EE_D9_CHCR, 11, 14, 21
EE_D9_CHCR_OFF, 15
EE_D9_MADR, 11, 14
EE_D9_MADR_OFF, 15
EE_D9_QWC, 11, 14
EE_D9_QWC_OFF, 15
EE_D9_SADR, 11, 14
EE_D9_SADR_OFF, 15
EE_D9_TADR, 14
EE_D9_TADR_OFF, 15
EE_GIF_CNT, 13
EE_GIF_CNT_OFF, 14
EE_GIF_CTRL, 13
EE_GIF_CTRL_OFF, 14
EE_GIF_MODE, 13
EE_GIF_MODE_OFF, 14
EE_GIF_P3CNT, 13
EE_GIF_P3CNT_OFF, 14
EE_GIF_P3TAG, 13
EE_GIF_P3TAG_OFF, 14
EE_GIF_STAT, 13
EE_GIF_STAT_OFF, 14
EE_GIF_TAG0, 13
EE_GIF_TAG0_OFF, 14
EE_GIF_TAG1, 13
EE_GIF_TAG1_OFF, 14
EE_GIF_TAG2, 13
EE_GIF_TAG2_OFF, 14
EE_GIF_TAG3, 13
EE_GIF_TAG3_OFF, 14
EE_I_MASK, 14
EE_I_MASK_OFF, 15
EE_I_STAT, 14
EE_I_STAT_OFF, 15
EE_IPU_BP, 13
EE_IPU_BP_OFF, 14
EE_IPU_CMD, 13
EE_IPU_CMD_OFF, 14
EE_IPU_CTRL, 13
EE_IPU_CTRL_OFF, 14
EE_IPU_TOP, 13
EE_IPU_TOP_OFF, 14
EE_SB_SMFLG, 14
EE_SB_SMFLG_OFF, 15

EE_T0_COMP, 13
EE_T0_COMP_OFF, 14
EE_T0_COUNT, 13
EE_T0_COUNT_OFF, 14
EE_T0_HOLD, 13
EE_T0_HOLD_OFF, 14
EE_T0_MODE, 13
EE_T0_MODE_OFF, 14
EE_T1_COMP, 13
EE_T1_COMP_OFF, 14
EE_T1_COUNT, 13
EE_T1_COUNT_OFF, 14
EE_T1_HOLD, 13
EE_T1_HOLD_OFF, 14
EE_T1_MODE, 13
EE_T1_MODE_OFF, 14
EE_T2_COMP, 13
EE_T2_COMP_OFF, 14
EE_T2_COUNT, 13
EE_T2_COUNT_OFF, 14
EE_T2_MODE, 13
EE_T2_MODE_OFF, 14
EE_T3_COMP, 13
EE_T3_COMP_OFF, 14
EE_T3_COUNT, 13
EE_T3_COUNT_OFF, 14
EE_T3_MODE, 13
EE_T3_MODE_OFF, 14
EE_VIF0_C0, 13
EE_VIF0_C0_OFF, 14
EE_VIF0_C1, 13
EE_VIF0_C1_OFF, 14
EE_VIF0_C2, 13
EE_VIF0_C2_OFF, 14
EE_VIF0_C3, 13
EE_VIF0_C3_OFF, 14
EE_VIF0_CODE, 13
EE_VIF0_CODE_OFF, 14
EE_VIF0_CYCLE, 13
EE_VIF0_CYCLE_OFF, 14
EE_VIF0_ERR, 13
EE_VIF0_ERR_OFF, 14
EE_VIF0_FBRST, 13
EE_VIF0_FBRST_OFF, 14
EE_VIF0_ITOP, 13
EE_VIF0_ITOP_OFF, 14
EE_VIF0_ITOPS, 13
EE_VIF0_ITOPS_OFF, 14
EE_VIF0_MARK, 13
EE_VIF0_MARK_OFF, 14
EE_VIF0_MASK, 13
EE_VIF0_MASK_OFF, 14
EE_VIF0_MODE, 13
EE_VIF0_MODE_OFF, 14
EE_VIF0_NUM, 13
EE_VIF0_NUM_OFF, 14
EE_VIF0_R0, 13
EE_VIF0_R0_OFF, 14
EE_VIF0_R1, 13
EE_VIF0_R1_OFF, 14
EE_VIF0_R2, 13
EE_VIF0_R2_OFF, 14
EE_VIF0_R3, 13
EE_VIF0_R3_OFF, 14
EE_VIF0_STAT, 13
EE_VIF0_STAT_OFF, 14
EE_VIF1_BASE, 13
EE_VIF1_BASE_OFF, 14
EE_VIF1_C0, 13
EE_VIF1_C0_OFF, 14
EE_VIF1_C1, 13
EE_VIF1_C1_OFF, 14
EE_VIF1_C2, 13
EE_VIF1_C2_OFF, 14
EE_VIF1_C3, 13
EE_VIF1_C3_OFF, 14
EE_VIF1_CODE, 13
EE_VIF1_CODE_OFF, 14
EE_VIF1_CYCLE, 13
EE_VIF1_CYCLE_OFF, 14
EE_VIF1_ERR, 13
EE_VIF1_ERR_OFF, 14
EE_VIF1_FBRST, 13
EE_VIF1_FBRST_OFF, 14
EE_VIF1_ITOP, 13
EE_VIF1_ITOP_OFF, 14
EE_VIF1_ITOPS, 13
EE_VIF1_ITOPS_OFF, 14
EE_VIF1_MARK, 13
EE_VIF1_MARK_OFF, 14
EE_VIF1_MASK, 13
EE_VIF1_MASK_OFF, 14
EE_VIF1_MODE, 13
EE_VIF1_MODE_OFF, 14
EE_VIF1_NUM, 13
EE_VIF1_NUM_OFF, 14
EE_VIF1_OFST, 13
EE_VIF1_OFST_OFF, 14
EE_VIF1_R0, 13
EE_VIF1_R0_OFF, 14
EE_VIF1_R1, 13
EE_VIF1_R1_OFF, 14
EE_VIF1_R2, 13
EE_VIF1_R2_OFF, 14
EE_VIF1_R3, 13
EE_VIF1_R3_OFF, 14
EE_VIF1_STAT, 13
EE_VIF1_STAT_OFF, 14
EE_VIF1_TOP, 13
EE_VIF1_TOP_OFF, 14
EE_VIF1_TOPS, 13
EE_VIF1_TOPS_OFF, 14

