

SPS2

A Development Library for
Linux (for PlayStation 2)

Steven "Sauce" Osman
Terratron® Technologies Inc.

Table of Contents

Table of Contents	2
Introduction	4
License Agreement.....	5
Acknowledgements	6
History of Changes	7
Version 0.2.0	7
Version 0.2.0a	7
Programming with SPS2	8
SPS2 Files and Directories.....	9
Installing and Loading SPS2	10
Before You Build the Kernel Module	10
Building the Kernel Module	10
Loading the Kernel Module	10
Unloading the Kernel Module.....	10
Removing SPS2 From Your System.....	10
Building the SPS2 Utility Library, libsp2util	11
Building and Running the Sample Applications	11
A Sample SPS2 Program	13
SPS2 Programmer's Guide	15
Performing a DMA Transfer	15
Accessing the Emotion Engine Registers	16
Accessing the Graphics Synthesizer Registers	19
Accessing the Scratch Pad Memory	21
Accessing the Vector Unit Memories	22
Using SPS2 from Assembly Language.....	23
Drawing a Triangle using libsp2dev & sp2UScreen	24
SPS2 Core Function Set Reference.....	27
sp2Init	28
sp2Release.....	30
sp2Allocate	31
sp2Free	33
sp2Remap	34
sp2GetPhysicalAddress	35
sp2FlushCache.....	36
sp2WaitForDMA	37
SPS2 Extended Function Set Reference	38
_sp2Open	40
_sp2Close.....	41
_sp2MapEERegisters.....	42
_sp2MapGSRegisters	43
_sp2MapVUMemory	44
_sp2MapScratchPad	45
SPS2 Utility Library, libsp2util.....	46
sp2UScreenInit	47
sp2UScreenShutdown.....	49
sp2UScreenSwap	50
sp2UScreenGetFirstFreeGSPage.....	51
sp2UScreenGetZPtr	52
sp2UScreenGetDrawBuff1	53
sp2UScreenGetDrawBuff2	54
sp2UScreenGetDrawBuffCurrent	55

sps2UScreenGetWidth.....	56
sps2UScreenGetHeight.....	57
sps2UScreenGetZDepth	58
sps2UScreenGetVideoMode.....	59
sps2UScreenGetPixelFormat.....	60
Index	61

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 Vector 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 560,000 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.

"Linux" is a trademark or registered trademark of Linus Torvalds
PlayStation is a registered trademark of Sony Computer Entertainment Inc.
Terratron is a registered trademark of Terratron Technologies Inc.

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

History of Changes

Version 0.2.0

First public release.

Version 0.2.0a

- No new kernel module – the kernel module is still at 0.2.0.
- Some minor additions and corrections to the sps2 headers as well as a little bit of restructuring. These changes include adding structures for some of the PS2 tags (e.g. DMA tag & GIF tag). Refer to the headers for more details
- Updated the samples directory and added the tshow sample. In addition, the readme.txt in the samples directory file has been expanded.
- Introduced the *SPS2 Utility Library*, libsp2util. This library contains a simple set of screen handling routines as well as non-inline version of the sps2 functions for those wanting to link to sps2 functions from other programming languages (e.g. assembly language)

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*.

Version 0.2.0a of SPS2 introduces a new library called the *SPS2 Utility Library*, or `libsps2util`. This library currently serves two major functions. First, it defines non-inline equivalents of all the functions in the *SPS2 Core Function Set* and the *SPS2 Extended Function Set*. These sets of functions, prefixed with `sps2UNL` and `_sps2UNL` respectively, can be used to leverage SPS2 functionality from any programming language that is capable of linking to libraries that use the C calling convention. In addition, the *SPS2 Utility Library* adds functions that enable a programmer to easily enable a graphics mode in the GS, and switch between double buffers. This functionality, which is robust enough to be used by all of the samples provided with SPS2 allows developers to quickly set up the screen and focus more on developing their application than initializing the system.

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 <i>SPS2 ioctl Command Set</i> and the format of their parameters and return values
sps2lib.h	File	Defines the <i>SPS2 Core Function Set</i> and the <i>SPS2 Extended Function Set</i> 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
sps2tags.h	File	A header file that defines the structure of various PS2 tags
sps2types.h	File	A header file the defines the data types used by SPS2
sps2util.h	File	The header file that defines the functions in the <i>SPS2 Utility Library</i> , libsp2util
sps2util.c	File	The source code the the <i>SPS2 Utility Library</i> , libsp2util
sps2vumemory.h	File	Defines macros to access the vector unit memories
Makefile	File	The makefile used to build the SPS2 Utility Library, libsp2util
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
dyntaxs	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
tshower	Directory	Shows a high performance particle system that takes advantage of loop unrolling for maximum efficiency.
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.

Before You Build the Kernel Module

The build process assumes that the kernel source code is available in `/usr/src/linux`. For those of you who have installed a new kernel (for example, the xRhino kernel), chances are that your `/usr/src/linux` will still point to the original Linux (for PlayStation2) kernel. You'll need to install the xRhino kernel source in `/usr/src` and create a symbolic link to `/usr/src/linux`.

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 the SPS2 Utility Library, libsp2util

The *SPS2 Utility Library*, libsp2util, is built simply by running make from within the top-level SPS2 development directory, like so:

```
make
```

In order to link to the library, you must use the `-L` directive to specify where libsp2util.a resides, and then the `-lsp2util` directive to include the library. For example, to compile a source file called `hello.c` into a program called `hello`, and assuming that you have extracted the SPS2 development files in `~/sps2dev-0.2.0a`, issue:

```
gcc -c -o hello.o hello.c
gcc -o hello -lsp2util -L~/sps2dev-0.2.0a
```

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
./intapp
```

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.

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

```
cd tshower  
./tshower
```

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 in `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_VIF0_MASK	DMAC
EE_T0_COUNT	EE_VIF0_CODE	EE_D0_CHCR
EE_T0_MODE	EE_VIF0_ITOPS	EE_D0_MADR
EE_T0_COMP	EE_VIF0_ITOP	EE_D0_QWC
EE_T0_HOLD	EE_VIF0_R0	EE_D0_TADR
	EE_VIF0_R1	EE_D0_ASR0
	EE_VIF0_R2	EE_D0_ASR1
EE_T1_COUNT	EE_VIF0_R3	
EE_T1_MODE	EE_VIF0_C0	EE_D1_CHCR
EE_T1_COMP	EE_VIF0_C1	EE_D1_MADR
EE_T1_HOLD	EE_VIF0_C2	EE_D1_QWC
	EE_VIF0_C3	EE_D1_TADR
EE_T2_COUNT		EE_D1_ASR0
EE_T2_MODE	VIF1	EE_D1_ASR1
EE_T2_COMP	EE_VIF1_STAT	
	EE_VIF1_FBRST	EE_D2_CHCR
EE_T3_COUNT	EE_VIF1_ERR	EE_D2_MADR
EE_T3_MODE	EE_VIF1_MARK	EE_D2_QWC
EE_T3_COMP	EE_VIF1_CYCLE	EE_D2_TADR
	EE_VIF1_MODE	EE_D2_ASR0
IPU	EE_VIF1_NUM	EE_D2_ASR1
EE_IPU_CMD	EE_VIF1_MASK	
EE_IPU_CTRL	EE_VIF1_CODE	EE_D3_CHCR
EE_IPU_BP	EE_VIF1_ITOPS	EE_D3_MADR
EE_IPU_TOP	EE_VIF1_BASE	EE_D3_QWC
	EE_VIF1_OFST	
GIF	EE_VIF1_TOPS	EE_D4_CHCR
EE_GIF_CTRL	EE_VIF1_ITOP	EE_D4_MADR
EE_GIF_MODE	EE_VIF1_TOP	EE_D4_QWC
EE_GIF_STAT	EE_VIF1_R0	EE_D4_TADR
EE_GIF_TAG0	EE_VIF1_R1	
EE_GIF_TAG1	EE_VIF1_R2	EE_D5_CHCR
EE_GIF_TAG2	EE_VIF1_R3	EE_D5_MADR
EE_GIF_TAG3	EE_VIF1_C0	EE_D5_QWC
EE_GIF_CNT	EE_VIF1_C1	
EE_GIF_P3CNT	EE_VIF1_C2	EE_D6_CHCR
EE_GIF_P3TAG	EE_VIF1_C3	EE_D6_MADR
		EE_D6_QWC
VIF0	FIFO	EE_D6_TADR
EE_VIF0_STAT	DPUT_EE_VIF0_FIFO(val)	
EE_VIF0_FBRST	DPUT_EE_VIF1_FIFO(val)	EE_D7_CHCR
EE_VIF0_ERR	DGET_EE_VIF1_FIFO(val)	EE_D7_MADR
EE_VIF0_MARK	DPUT_EE_GIF_FIFO(val)	EE_D7_QWC
EE_VIF0_CYCLE	DGET_EE_IPU_out_FIFO	
EE_VIF0_MODE	DPUT_EE_IPU_in_FIFO(val)	EE_D8_CHCR
EE_VIF0_NUM		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)

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)

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(base, val)
DGET_EE_IPU_out_FIFO_OFF(base)
DPUT_EE_IPU_in_FIFO_OFF(base, val)

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)

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)

DMAC

EE_D0_CHCR_OFF(base)
EE_D0_MADR_OFF(base)
EE_D0_QWC_OFF(base)
EE_D0_TADR_OFF(base)
EE_D0_ASRO_OFF(base)
EE_D0_ASRI_OFF(base)

EE_D1_CHCR_OFF(base)
EE_D1_MADR_OFF(base)
EE_D1_QWC_OFF(base)
EE_D1_TADR_OFF(base)
EE_D1_ASRO_OFF(base)
EE_D1_ASRI_OFF(base)

EE_D2_CHCR_OFF(base)
EE_D2_MADR_OFF(base)
EE_D2_QWC_OFF(base)
EE_D2_TADR_OFF(base)
EE_D2_ASRO_OFF(base)
EE_D2_ASRI_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 Vector 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 Vector Unit Memories

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

The Vector 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 Vector 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 Vector Unit memories with `_sps2MapVUMemory`
4. Access the Vector 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).

Using SPS2 from Assembly Language

Version 0.2.0a of SPS2 introduces the *SPS2 Utility Library*, libsp2util which is a library that contains non-inline equivalents of all the sps2 core and extended functions documented in this manual. These have prefixes of sps2UNI and _sps2UNI (for core and extended functions, respectively), with the U signifying the SPS2 (U)tility library, libsp2util, and the NI standing for non inline function.

These functions can be linked to using standard C calling conventions from other programming languages, allowing any programming language that can utilize C libraries to take advantage of the functionality offered by SPS2.

Drawing a Triangle using libsp2dev & sps2UScreen

The following is a small program that uses the *SPS2 Core Function Set* and libsp2dev to draw a multi-colored triangle on the screen. Note that it does so by directly poking values into the GIF FIFO, which is probably the slowest way in which a triangle can be drawn. On the other hand, it shows how the `sps2UScreen` screen handling routine functions can be used to create a graphical application with double buffering.

This code performs simple initialization by calling

- `sps2Init()`
- `sps2UScreenInit(0)`

It then defines a function called `drawTriangle()` which:

- Creates a GIF tag (see `sps2tags.h` for details)
- Sends it to the GIF through the GIF FIFO
- Then sends three color and x,y,z coordinate pairs to draw the triangle.

Once the triangle has been drawn, it uses `sps2UScreenSwap()` to wait for VSync and to swap buffers. Finally, had there been code to break out of the loop, this code would have shut everything down by calling:

- `sps2UScreenShutdown()`
- `sps2Release()`.

```
#include <stdio.h>

#include <sps2lib.h>
#include <sps2tags.h>
#include <sps2util.h>

void drawTriangle();

// Define the coordinates of our triangle
typedef int vertex_t[2];
vertex_t aVertices[]={
    {-150, -150}, // Up from the center, to the left
    { 150, -150}, // Up from the center, to the right
    {   0,  150}  // Down from the center, in the middle
}; // triangle coordinates

// Define the colors of the points
// Make sure there's one of these for each coordinate!
typedef int color_t[3];
color_t aColors[]={
    {255,  0,  0}, // All red
    {  0,255,  0}, // All green
    {  0,  0,255}  // All blue
}; // triangle colors

/**
 * This is the main function of our tutorial. It will perform some simple
 * initialization and then set up a small loop to repeatedly draw our
 * triangle.
 */
int main() {
```



```

int iSPS2Descriptor;          // The handle we use for sps2

iSPS2Descriptor=sps2Init();    // Initialize sps2

// Initialize the screen. Zero tells sps2UScreenInit to exit gracefully
// when it receives a signal.
sps2UScreenInit(0);

while (1) {
    // Draw the triangle
    drawTriangle();

    // Swap displays now that we're done
    sps2UScreenSwap();
}

// Shut down the screen
sps2UScreenShutdown();

// and close sps2
sps2Release(iSPS2Descriptor);
return 0;
}

/**
 * This function will draw our triangle. It's a little bit inefficient because
 * it builds the whole GIF tag every frame (the GIF tag could have been
 * prepared earlier, for instance). On the other hand, since we're only
 * drawing one triangle on the screen, this is of very little concern.
 */
void drawTriangle() {
    sps2GIFTag_t gifTag;
    sps2GIFPackedRegister_t gifRegister;
    int iPointLooper;

    // Prepare the GIF tag
    gifTag.i128=0;          // Blank it out
    gifTag.s.NLOOP=3;       // 3 entries. One entry for each of the points.
    gifTag.s.PRE=1;         // We are providing a valid PRIM value
    gifTag.s.EOP=1;         // End of packet -- draw me please!
    gifTag.s.PRIM=11;       // 3 = triangle | 8 = gourard shading
    gifTag.s.FLG=GIF_FLG_PACKED; // We're using the packed format for data
    gifTag.s.NREG=0;        // Start off with zero registers, increment as we
                           // populate the descriptions

    // Add a register. It will be an RGBAQ register
    SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_RGBAQ);
    // Add another register. It will be an XYZ2 register
    SPS2_SET_GIF_REG(gifTag, gifTag.s.NREG++, GIF_REG_XYZ2);

    // Send the GIF tag
    DPUT_EE_GIF_FIFO(gifTag.i128);

    // Now that we promised three sets of RGBAQ and XYZ2 registers, we should
    // deliver them.
    for (iPointLooper=0; iPointLooper<3;iPointLooper++) {
        // First, prepare and send the RGBAQ register
        gifRegister.i128=0;
        gifRegister.RGBAQ.s.R=aColors[iPointLooper][0];
        gifRegister.RGBAQ.s.G=aColors[iPointLooper][1];
        gifRegister.RGBAQ.s.B=aColors[iPointLooper][2];
        DPUT_EE_GIF_FIFO(gifRegister.i128);
    }
}

```

```
// Next, prepare and send the XYZ2 register
gifRegister.i128=0;
gifRegister.XYZ2.s.X=(aVertices[iPointLooper][0]+2048) << 4;
gifRegister.XYZ2.s.Y=(aVertices[iPointLooper][1]+2048) << 4;
gifRegister.XYZ2.s.Z=(aVertices[iPointLooper][2]) << 4;
DPUT_EE_GIF_FIFO(gifRegister.i128);
}
}
```

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();
```

Non-inline version in libsp2util:

```
int sps2UNIIInit();
```

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 Vector 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 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 Vector 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);
```

Non-inline version in libsp2util:

```
int sps2UNIRelease (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 Vector 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);
```

Non-inline version in libsp2util:

```
sps2Memory_t *sps2UNIAAllocate(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, -1, 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 sps2Free(sps2Memory_t *pMapping);
```

Non-inline version in libsp2util:

```
void sps2UNIFree(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);
```

Non-inline version in libsp2util:

```
sps2Memory_t *sps2UNIRemap(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, -1, 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);
```

Non-inline version in libsp2util:

```
unsigned long sps2UNIGetPhysicalAddress(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);
```

Non-inline version in libsp2util:

```
int sps2UNIFlushCache(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);
```

Non-inline version in libsp2util:

```
int sps2UNIWaitForDMA(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 Vector 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();
```

Non-inline version in libsp2util:

```
int sps2UNIOpen();
```

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);
```

Non-inline version in libsp2util:

```
int _sps2UNIClose(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 Vector 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);
```

Non-inline version in libsp2util:

```
int _sps2UNIMapEERegisters(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(base 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);
```

Non-inline version in libsps2util:

```
int _sps2UNIMapGSRegisters(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);
```

Non-inline version in libsp2util:

```
int _sps2UNIMapVUMemory(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 Vector Unit memories have been mapped
- On failure, MAP_FAILED (-1)

See Also:

`_sps2Open`

Comments:

This function attempts to map the Vector 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 Vector 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);
```

Non-inline version in libsp2util:

```
int _sps2UNIMapScratchPad(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`.

SPS2 Utility Library, libsp2util

The *SPS2 Utility Library*, also known as libsp2util defines non-inline versions of the various functions that comprise the *SPS2 Core Function Set* and the *SPS2 Extended Function Set*. These functions will NOT be described in detail here, instead, readers are directed to the documentation of the aforementioned function sets for details. All non-inline functions that begin with sps2UNI have counterparts in the *SPS2 Core Function Set* and all non-inline functions that begin with _sps2UNI have counterparts in the *SPS2 Extended Function Set*.

In summary, these functions are, for the *SPS2 Core Function Set*:

```
sps2UNIInit
sps2UNIRelease
sps2UNIAllocate
sps2UNIRemap
sps2UNIFree
sps2UNIGetPhysicalAddress
sps2UNIFlushCache
sps2UNIWaitForDMA
```

And for the *SPS2 Extended Function Set*:

```
_sps2UNIOpen
_sps2UNIClose
_sps2UNIMapEERegisters
_sps2UNIMapGSRegisters
_sps2UNIMapVUMemory
_sps2UNIMapScratchPad
```

In addition, libsp2util introduces a number of screen handling routines. All of these routines have a prefix of sps2UScreen. Their functionality is described in the following pages.

Important: sps2UScreenInit must be called first before any of the other sps2UScreen functions.

All libsp2util functions are declared in the include file sps2util.h. In order to build the library, and to link to it, please refer to the installation chapter earlier in this manual.

sps2UScreenInit

Prototype:

```
void sps2UScreenInit(int iSuppressSignalHandlers);
```

Parameters:

iSuppressSignalHandlers – set this to a non-zero value if you do not want *sps2UScreenInit* to trap all signals with handlers that close the virtual console before aborting your program.

Return Value:

- *None*

See Also:

```
sps2UScreenShutdown  
sps2UScreenGetVideoMode
```

Comments:

This function is used to initialize the screen. The current implementation will autodetect the video hardware and select an appropriate resolution, creating a virtual console. The screen is initialized for PSMCT32 pixel format and a 24 bit Z buffer, with double buffering. The display resolution is set as follows:

For NTSC, the screen is set to 640x448

For VESA, the screen is set to 640x480

For PAL, the screen is set to 640x512

If the library has trouble detecting the video mode, it will default to VESA.

Caution

Please make sure to call *sps2UScreenShutdown* after calling *sps2UScreenInit*. Failure to do so may not properly shut down your virtual console, which would require a reboot to correct (the machine does not crash; the console is just left in an unusable state).

sps2UScreenInit uses the *atexit* function to try to invoke *sps2UScreenShutdown* for you. In addition, unless you disallow it, *sps2UScreenInit* will trap more or less all signals (except *SIGSTOP* and *SIGCONT*) and gracefully abort upon receipt of a signal. If you need to create your own signal handlers, you have one of two choices:

1. Create them after invoking *sps2UScreenInit(0)*, understanding that *sps2UScreenShutdown* will remove them.
2. Make sure you trap all of the signals by defining a signal handler that will abort gracefully. For example, the following signal handler will call *sps2UScreenShutdown* before aborting:

```
/**  
 * This is our signal handler. Since it doesn't really know what to do when  
 * it gets a signal it will just abort the program. This beats the program  
 * crashing when it gets the unknown signal because the display code will  
 * have a chance to shut itself down.  
 * iSignal is the signal being delivered  
 */  
void signalHandler(int iSignal) {  
    printf("Quitting on signal %i\n", iSignal);  
    exit(0);  
}
```

Also, trap all the signals with the following code snippet:

```
#include <signal.h>

int iSignalLooper;

for(iSignalLooper=0; iSignalLooper<128; iSignalLooper++) {
    signal(iSignalLooper, signalHandler);
}
```

If you do not need to create your own signal handlers, then merely call `sps2UScreenInit(0)` and everything will be taken care of for you.

sps2UScreenShutdown

Prototype:

```
void sps2UScreenShutdown();
```

Parameters:

None

Return Value:

- *None*

See Also:

`sps2UScreenInit`

Comments:

This function is used to destroy the virtual console created by `sps2UScreenInit` and to revert the display to its original settings. Also, if `sps2UScreenInit` trapped the signal handlers, then this function will restore them.

sps2UScreenSwap

Prototype:

```
void sps2UScreenSwap();
```

Parameters:

None

Return Value:

- *None*

See Also:

`sps2UScreenInit`

Comments:

This function performs three tasks:

1. It waits for VSync
2. It swaps the display and draw buffers (what was being drawn is now being displayed and vice-verse)
3. It clears the draw buffer screen.

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetFirstFreeGSPage

Prototype:

```
unsigned int sps2UScreenGetFirstFreeGSPage ();
```

Parameters:

None

Return Value:

- The page number of the first available page in GS memory after the double buffers and the Z buffer.

See Also:

```
sps2UScreenInit  
sps2UScreenGetZPtr  
sps2UScreenGetDrawBuff2  
sps2UScreenGetDrawBuff1  
sps2UScreenGetDrawBuffCurrent
```

Comments:

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetZPtr

Prototype:

```
unsigned int sps2UScreenGetZPtr ();
```

Parameters:

None

Return Value:

- The first page number of the Z buffer in GS memory.

See Also:

```
sps2UScreenInit  
sps2UScreenGetFirstFreeGSPage  
sps2UScreenGetDrawBuff1  
sps2UScreenGetDrawBuff2  
sps2UScreenGetDrawBuffCurrent
```

Comments:

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetDrawBuff1

Prototype:

```
unsigned int sps2UScreenGetDrawBuff1();
```

Parameters:

None

Return Value:

- The first page in GS memory of the first draw buffer from the double buffer.

See Also:

```
sps2UScreenInit  
sps2UScreenGetFirstFreeGSPage  
sps2UScreenGetZPtr  
sps2UScreenGetDrawBuff2  
sps2UScreenGetDrawBuffCurrent
```

Comments:

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetDrawBuff2

Prototype:

```
unsigned int sps2UScreenGetDrawBuff2();
```

Parameters:

None

Return Value:

- The first page in GS memory of the second draw buffer from the double buffer.

See Also:

```
sps2UScreenInit  
sps2UScreenGetFirstFreeGSPage  
sps2UScreenGetZPtr  
sps2UScreenGetDrawBuff1  
sps2UScreenGetDrawBuffCurrent
```

Comments:

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetDrawBuffCurrent

Prototype:

```
unsigned int sps2UScreenGetDrawBuffCurrent();
```

Parameters:

None

Return Value:

- The first page in GS memory of the draw buffer from the double buffer that is currently being drawn to (i.e. not the display buffer).

See Also:

```
sps2UScreenInit  
sps2UScreenGetFirstFreeGSPage  
sps2UScreenGetZPtr  
sps2UScreenGetDrawBuff1  
sps2UScreenGetDrawBuff2
```

Comments:

This function will return either `sps2UScreenGetDrawBuff1()` or `sps2UScreenGetDrawBuff2()`

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetWidth

Prototype:

```
unsigned int sps2UScreenGetWidth();
```

Parameters:

None

Return Value:

- The width, in number of pixels, of the display resolution.

See Also:

```
sps2UScreenInit  
sps2UScreenGetHeight
```

Comments:

The current implementation of `sps2UScreen` always sets the screen width to 640 pixels.

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetHeight

Prototype:

```
unsigned int sps2UScreenGetHeight();
```

Parameters:

None

Return Value:

- The height, in number of pixels, of the display resolution.

See Also:

```
sps2UScreenInit  
sps2UScreenGetWidth  
sps2UScreenGetZDepth  
sps2UScreenGetVideoMode  
sps2UScreenGetPixelFormat
```

Comments:

The current implementation of `sps2UScreen` always sets the screen height to:

- 480 for `sps2UScreenGetVideoMode() == SPS2U_SCR_VESA`
- 448 for `sps2UScreenGetVideoMode() == SPS2U_SCR_NTSC`
- 512 for `sps2UScreenGetVideoMode() == SPS2U_SCR_PAL`

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetZDepth

Prototype:

```
unsigned int sps2UScreenGetZDepth();
```

Parameters:

None

Return Value:

- The largest Z value allowed in the Z buffer.

See Also:

```
sps2UScreenInit  
sps2UScreenGetWidth  
sps2UScreenGetHeight  
sps2UScreenGetVideoMode  
sps2UScreenGetPixelFormat
```

Comments:

This current implementation of `sps2UScreen` always uses a 24 bit Z buffer. This function will always return `0xffffffff`.

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetVideoMode

Prototype:

```
unsigned int sps2UScreenGetVideoMode();
```

Parameters:

None

Return Value:

The video mode auto-detected by `sps2UScreenInit`. This is one of:

- `SPS2U_SCR_VESA` for VESA
- `SPS2U_SCR_NTSC` for NTSC
- `SPS2U_SCR_PAL` for PAL

See Also:

```
sps2UScreenInit  
sps2UScreenGetWidth  
sps2UScreenGetHeight  
sps2UScreenGetZDepth  
sps2UScreenGetPixelFormat
```

Comments:

If `sps2UScreenInit` cannot properly detect your display type, it will assume VESA.

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

sps2UScreenGetPixelFormat

Prototype:

```
unsigned int sps2UScreenGetPixelFormat();
```

Parameters:

None

Return Value:

The pixel format of the double buffers.

See Also:

```
sps2UScreenInit  
sps2UScreenGetWidth  
sps2UScreenGetHeight  
sps2UScreenGetZDepth  
sps2UScreenGetVideoMode
```

Comments:

In this current implementation of `sps2UScreen`, this will always return zero, which corresponds to the PSMCT32 pixel format.

This function may only be used after `sps2UScreenInit` has been called but before `sps2UScreenShutdown` has been called.

Index

- `_sps2Close`, 15, 17, 20, 21, 22, 38, 40, 41, 42, 43, 44, 45
- `_sps2MapEERegisters`, 17, 38, 42
- `_sps2MapGSRegisters`, 20, 43
- `_sps2MapScratchPad`, 21, 45
- `_sps2MapVUMemory`, 22, 44
- `_sps2Open`, 15, 17, 20, 21, 22, 28, 31, 33, 34, 36, 37, 38, 40, 41, 42, 43, 44, 45
- `_sps2UNIClose`, 41, 46
- `_sps2UNIMapEERegisters`, 42, 46
- `_sps2UNIMapGSRegisters`, 43, 46
- `_sps2UNIMapScratchPad`, 45, 46
- `_sps2UNIMapVUMemory`, 44, 46
- `_sps2UNIOpen`, 40, 46
- Acknowledgements, 6
- ADDR, 14
- ASP, 15
- assembly language, 7, 23
- base pointer, 17, 20, 21, 22, 38, 42, 43, 44, 45
- bumpmap, 9, 11
- cache, 13, 15, 34, 36
- cached, 4, 13, 31, 34, 36
- common, 9
- `DGET_EE_IPU_out_FIFO`, 16
- `DGET_EE_IPU_out_FIFO_OFF`, 17
- `DGET_EE_VIF1_FIFO`, 16
- `DGET_EE_VIF1_FIFO_OFF`, 17
- DIR, 15
- Directories. *See* SPS2 Files and Directories
- display buffer, 55
- display resolution, 47, 56, 57
- DMA, 4, 8, 13, 14, 15
 - controller, 4, 13, 14, 15, 31, 35
 - controllet, 15
 - transfer, 4, 8, 13, 14, 15, 32, 35, 36, 37
- DMA tag, 7
- DMAC, 16, 17, *See* DMA:controller
- `Dn_CHCR`, 13, 14, 15
- `Dn_CHCR_t`, 13, 15
- `Dn_MADR_t`, 13
- `Dn_SADR_t`, 13
- double buffer*, 8, 24, 47, 51, 53, 54, 55, 60
- `DPUT_EE_GIF_FIFO`, 16, 17, 25, 26
- `DPUT_EE_GIF_FIFO_OFF`, 17
- `DPUT_EE_IPU_in_FIFO`, 16
- `DPUT_EE_IPU_in_FIFO_OFF`, 17
- `DPUT_EE_VIF0_FIFO`, 16
- `DPUT_EE_VIF0_FIFO_OFF`, 17
- `DPUT_EE_VIF1_FIFO`, 16
- `DPUT_EE_VIF1_FIFO_OFF`, 17
- `DPUT_GS_BGCOLOR`, 14, 19, 28, 43
- `DPUT_GS_BGCOLOR_OFF`, 20, 43
- `DPUT_GS_BUSDIR`, 19
- `DPUT_GS_BUSDIR_OFF`, 20
- `DPUT_GS_CSR`, 19
- `DPUT_GS_CSR_OFF`, 20
- `DPUT_GS_DISPFB1`, 19
- `DPUT_GS_DISPFB1_OFF`, 20
- `DPUT_GS_DISPFB2`, 19
- `DPUT_GS_DISPFB2_OFF`, 20
- `DPUT_GS_DISPLAY1`, 19
- `DPUT_GS_DISPLAY1_OFF`, 20
- `DPUT_GS_DISPLAY2`, 19
- `DPUT_GS_DISPLAY2_OFF`, 20
- `DPUT_GS_EXTBUF`, 19
- `DPUT_GS_EXTBUF_OFF`, 20
- `DPUT_GS_EXTDATA`, 19
- `DPUT_GS_EXTDATA_OFF`, 20
- `DPUT_GS_EXTWRITE`, 19
- `DPUT_GS_EXTWRITE_OFF`, 20
- `DPUT_GS_IMR`, 19
- `DPUT_GS_IMR_OFF`, 20
- `DPUT_GS_PMODE`, 19
- `DPUT_GS_PMODE_OFF`, 20
- `DPUT_GS_SIGBLID`, 19
- `DPUT_GS_SIGBLID_OFF`, 20
- `DPUT_GS_SMODE1`, 19
- `DPUT_GS_SMODE1_OFF`, 20
- `DPUT_GS_SMODE2`, 19
- `DPUT_GS_SMODE2_OFF`, 20
- `DPUT_GS_SRFSH`, 19
- `DPUT_GS_SRFSH_OFF`, 20
- `DPUT_GS_SYNCH1`, 19
- `DPUT_GS_SYNCH1_OFF`, 20
- `DPUT_GS_SYNCH2`, 19
- `DPUT_GS_SYNCH2_OFF`, 20
- `DPUT_GS_SYNCV`, 19
- `DPUT_GS_SYNCV_OFF`, 20
- draw buffer, 50, 53, 54, 55
- dyntexs, 9, 11
- EE. *See* Emotion Engine
- `EE_D_CTRL`, 17
- `EE_D_CTRL_OFF`, 17
- `EE_D_ENABLER`, 17
- `EE_D_ENABLER_OFF`, 18
- `EE_D_ENABLEW`, 17
- `EE_D_ENABLEW_OFF`, 18
- `EE_D_PCR`, 17
- `EE_D_PCR_OFF`, 18
- `EE_D_RBOR`, 17
- `EE_D_RBOR_OFF`, 18
- `EE_D_RBSR`, 17
- `EE_D_RBSR_OFF`, 18
- `EE_D_SQWC`, 17

EE_D_SQWC_OFF, 18	EE_D5_CHCR_OFF, 18
EE_D_STADR, 17	EE_D5_MADR, 16
EE_D_STADR_OFF, 18	EE_D5_MADR_OFF, 18
EE_D_STAT, 17	EE_D5_QWC, 16
EE_D_STAT_OFF, 18	EE_D5_QWC_OFF, 18
EE_D0_ASR0, 16	EE_D6_CHCR, 16
EE_D0_ASR0_OFF, 17	EE_D6_CHCR_OFF, 18
EE_D0_ASR1, 16	EE_D6_MADR, 16
EE_D0_ASR1_OFF, 17	EE_D6_MADR_OFF, 18
EE_D0_CHCR, 16	EE_D6_QWC, 16
EE_D0_CHCR_OFF, 17, 42	EE_D6_QWC_OFF, 18
EE_D0_MADR, 16	EE_D6_TADR, 16
EE_D0_MADR_OFF, 17	EE_D6_TADR_OFF, 18
EE_D0_QWC, 16, 38	EE_D7_CHCR, 16
EE_D0_QWC_OFF, 17, 38	EE_D7_CHCR_OFF, 18
EE_D0_TADR, 16	EE_D7_MADR, 16
EE_D0_TADR_OFF, 17	EE_D7_MADR_OFF, 18
EE_D1_ASR0, 16	EE_D7_QWC, 16
EE_D1_ASR0_OFF, 17	EE_D7_QWC_OFF, 18
EE_D1_ASR1, 16	EE_D8_CHCR, 16
EE_D1_ASR1_OFF, 17	EE_D8_CHCR_OFF, 18
EE_D1_CHCR, 16	EE_D8_MADR, 16
EE_D1_CHCR_OFF, 17	EE_D8_MADR_OFF, 18
EE_D1_MADR, 16	EE_D8_QWC, 17
EE_D1_MADR_OFF, 17	EE_D8_QWC_OFF, 18
EE_D1_QWC, 16	EE_D8_SADR, 17
EE_D1_QWC_OFF, 17	EE_D8_SADR_OFF, 18
EE_D1_TADR, 16	EE_D9_CHCR, 14, 17, 28
EE_D1_TADR_OFF, 17	EE_D9_CHCR_OFF, 18
EE_D2_ASR0, 16	EE_D9_MADR, 14, 17
EE_D2_ASR0_OFF, 17	EE_D9_MADR_OFF, 18
EE_D2_ASR1, 16	EE_D9_QWC, 14, 17
EE_D2_ASR1_OFF, 17	EE_D9_QWC_OFF, 18
EE_D2_CHCR, 16	EE_D9_SADR, 14, 17
EE_D2_CHCR_OFF, 17	EE_D9_SADR_OFF, 18
EE_D2_MADR, 16	EE_D9_TADR, 17
EE_D2_MADR_OFF, 17	EE_D9_TADR_OFF, 18
EE_D2_QWC, 16	EE_GIF_CNT, 16
EE_D2_QWC_OFF, 17	EE_GIF_CNT_OFF, 17
EE_D2_TADR, 16	EE_GIF_CTRL, 16
EE_D2_TADR_OFF, 17	EE_GIF_CTRL_OFF, 17
EE_D3_CHCR, 16	EE_GIF_MODE, 16
EE_D3_CHCR_OFF, 18	EE_GIF_MODE_OFF, 17
EE_D3_MADR, 16	EE_GIF_P3CNT, 16
EE_D3_MADR_OFF, 18	EE_GIF_P3CNT_OFF, 17
EE_D3_QWC, 16	EE_GIF_P3TAG, 16
EE_D3_QWC_OFF, 18	EE_GIF_P3TAG_OFF, 17
EE_D4_CHCR, 16	EE_GIF_STAT, 16
EE_D4_CHCR_OFF, 18	EE_GIF_STAT_OFF, 17
EE_D4_MADR, 16	EE_GIF_TAG0, 16
EE_D4_MADR_OFF, 18	EE_GIF_TAG0_OFF, 17
EE_D4_QWC, 16	EE_GIF_TAG1, 16
EE_D4_QWC_OFF, 18	EE_GIF_TAG1_OFF, 17
EE_D4_TADR, 16	EE_GIF_TAG2, 16
EE_D4_TADR_OFF, 18	EE_GIF_TAG2_OFF, 17
EE_D5_CHCR, 16	EE_GIF_TAG3, 16

EE_GIF_TAG3_OFF, 17	EE_VIF0_ERR_OFF, 17
EE_I_MASK, 17	EE_VIF0_FBRST, 16
EE_I_MASK_OFF, 18	EE_VIF0_FBRST_OFF, 17
EE_I_STAT, 17	EE_VIF0_ITOP, 16
EE_I_STAT_OFF, 18	EE_VIF0_ITOP_OFF, 17
EE_IPU_BP, 16	EE_VIF0_ITOPS, 16
EE_IPU_BP_OFF, 17	EE_VIF0_ITOPS_OFF, 17
EE_IPU_CMD, 16	EE_VIF0_MARK, 16
EE_IPU_CMD_OFF, 17	EE_VIF0_MARK_OFF, 17
EE_IPU_CTRL, 16	EE_VIF0_MASK, 16
EE_IPU_CTRL_OFF, 17	EE_VIF0_MASK_OFF, 17
EE_IPU_TOP, 16	EE_VIF0_MODE, 16
EE_IPU_TOP_OFF, 17	EE_VIF0_MODE_OFF, 17
EE_SB_SMFLG, 17	EE_VIF0_NUM, 16
EE_SB_SMFLG_OFF, 18	EE_VIF0_NUM_OFF, 17
EE_T0_COMP, 16	EE_VIF0_R0, 16
EE_T0_COMP_OFF, 17	EE_VIF0_R0_OFF, 17
EE_T0_COUNT, 16	EE_VIF0_R1, 16
EE_T0_COUNT_OFF, 17	EE_VIF0_R1_OFF, 17
EE_T0_HOLD, 16	EE_VIF0_R2, 16
EE_T0_HOLD_OFF, 17	EE_VIF0_R2_OFF, 17
EE_T0_MODE, 16	EE_VIF0_R3, 16
EE_T0_MODE_OFF, 17	EE_VIF0_R3_OFF, 17
EE_T1_COMP, 16	EE_VIF0_STAT, 16
EE_T1_COMP_OFF, 17	EE_VIF0_STAT_OFF, 17
EE_T1_COUNT, 16	EE_VIF1_BASE, 16
EE_T1_COUNT_OFF, 17	EE_VIF1_BASE_OFF, 17
EE_T1_HOLD, 16	EE_VIF1_C0, 16
EE_T1_HOLD_OFF, 17	EE_VIF1_C0_OFF, 17
EE_T1_MODE, 16	EE_VIF1_C1, 16
EE_T1_MODE_OFF, 17	EE_VIF1_C1_OFF, 17
EE_T2_COMP, 16	EE_VIF1_C2, 16
EE_T2_COMP_OFF, 17	EE_VIF1_C2_OFF, 17
EE_T2_COUNT, 16	EE_VIF1_C3, 16
EE_T2_COUNT_OFF, 17	EE_VIF1_C3_OFF, 17
EE_T2_MODE, 16	EE_VIF1_CODE, 16
EE_T2_MODE_OFF, 17	EE_VIF1_CODE_OFF, 17
EE_T3_COMP, 16	EE_VIF1_CYCLE, 16
EE_T3_COMP_OFF, 17	EE_VIF1_CYCLE_OFF, 17
EE_T3_COUNT, 16	EE_VIF1_ERR, 16
EE_T3_COUNT_OFF, 17	EE_VIF1_ERR_OFF, 17
EE_T3_MODE, 16	EE_VIF1_FBRST, 16
EE_T3_MODE_OFF, 17	EE_VIF1_FBRST_OFF, 17
EE_VIF0_C0, 16	EE_VIF1_ITOP, 16
EE_VIF0_C0_OFF, 17	EE_VIF1_ITOP_OFF, 17
EE_VIF0_C1, 16	EE_VIF1_ITOPS, 16
EE_VIF0_C1_OFF, 17	EE_VIF1_ITOPS_OFF, 17
EE_VIF0_C2, 16	EE_VIF1_MARK, 16
EE_VIF0_C2_OFF, 17	EE_VIF1_MARK_OFF, 17
EE_VIF0_C3, 16	EE_VIF1_MASK, 16
EE_VIF0_C3_OFF, 17	EE_VIF1_MASK_OFF, 17
EE_VIF0_CODE, 16	EE_VIF1_MODE, 16
EE_VIF0_CODE_OFF, 17	EE_VIF1_MODE_OFF, 17
EE_VIF0_CYCLE, 16	EE_VIF1_NUM, 16
EE_VIF0_CYCLE_OFF, 17	EE_VIF1_NUM_OFF, 17
EE_VIF0_ERR, 16	EE_VIF1_OFST, 16

EE_VIF1_OFST_OFF, 17
 EE_VIF1_R0, 16
 EE_VIF1_R0_OFF, 17
 EE_VIF1_R1, 16
 EE_VIF1_R1_OFF, 17
 EE_VIF1_R2, 16
 EE_VIF1_R2_OFF, 17
 EE_VIF1_R3, 16
 EE_VIF1_R3_OFF, 17
 EE_VIF1_STAT, 16
 EE_VIF1_STAT_OFF, 17
 EE_VIF1_TOP, 16
 EE_VIF1_TOP_OFF, 17
 EE_VIF1_TOPS, 16
 EE_VIF1_TOPS_OFF, 17
 Emotion Engine, 4, 9, 16, 17, 19, 28, 30, 38, 41, 42
 EOP, 25
 FIFO, 16, 17, 24
 Files. *See* SPS2 Files and Directories
 FLG, 25
 GIF, 16, 17, 24
 GIF FIFO, 24
 GIF tag, 7, 24
 GIF_FLG_PACKED, 25
 GIF_REG_RGBAQ, 25
 GIF_REG_XYZ2, 25
 Graphics Synthesizer, 4, 9, 13, 19, 20, 28, 30, 38, 41, 43
 GS. *See* Graphics Synthesizer
 GS_BGCOLOR, 19, 20
 int_lock, 9, 11
 INTC, 17, 18
 IPU, 16, 17
 Kernel Module, 10, 11
 libsp2dev, 24
 libsp2util, 7, 8, 9, 11, 23, 46
 License, 5
 MIPS 3, 19
 MOD, 15
 NLOOP, 25
 NREG, 25
 NTSC, 47, 57, 59
 PAL, 47, 57, 59
 physical address, 14, 32, 35
 pixel format, 47, 60
 PRE, 25
 PRIM, 25
 ps2lframework, 9, 11
 PSMCT32, 47, 60
 pvEERegisters, 38
 pvStart, 13, 31
 registers. *See* Graphics Synthesizer, Emotion Engine
 resolution, 47, *See* display resolution
 RGBAQ, 25
 Scratch Pad, 4, 13, 21, 28, 30, 38, 41, 45
 SCRATCH_PAD, 14, 21, 28, 30, 45
 SCRATCH_PAD_OFF, 21, 45
 SIF, 17, 18
 SIGCONT, 47
 signal handler, 47
 signal handlers, 49
 signals, 47
 SIGSTOP, 47
 spr. *See* Scratch Pad
 SPR, 14
 SPS2 Core Function Set, 8, 9, 16, 19, 21, 22, 24, 27, 28, 38, 46
 SPS2 Extended Function Set, 8, 9, 17, 20, 21, 22, 28, 38, 46
 SPS2 Files and Directories, 9
 SPS2 ioctl Command Set, 8
 SPS2 Utility Library. *See* libsp2util
 SPS2_EE_REGISTERS_START, 28, 30
 SPS2_GS_REGISTERS_START, 28, 30
 sps2_load, 10
 SPS2_MAP_BLOCK_128K, 31, 34
 SPS2_MAP_BLOCK_16K, 31, 34
 SPS2_MAP_BLOCK_32K, 31, 34
 SPS2_MAP_BLOCK_4K, 13, 31, 34
 SPS2_MAP_BLOCK_64K, 31, 34
 SPS2_MAP_BLOCK_8K, 31, 34
 SPS2_MAP_CACHED, 13, 31, 34
 SPS2_MAP_UNCACHED, 31, 34
 sps2_mod, 10
 SPS2_SCRATCH_PAD_START, 28, 30
 SPS2_SET_GIF_REG, 25
 sps2_unload, 10
 SPS2_VU_MEMORY_START, 28, 30
 sps2Allocate, 13, 15, 28, 31, 33, 34, 35, 36
 sps2FlushCache, 14, 15, 31, 32, 34, 36
 sps2Free, 15, 28, 30, 31, 33, 34, 41
 sps2GetPhysicalAddress, 14, 31, 32, 34, 35
 sps2GIFPackedRegister_t, 25
 sps2GIFTag_t, 25
 sps2Init, 13, 15, 16, 19, 21, 22, 24, 25, 28, 29, 30, 31, 33, 34, 36, 37, 38, 40, 42, 43, 44, 45
 sps2interface.h, 9
 sps2lib.h, 9, 13, 15, 16, 17, 19, 20, 21, 22, 24
 sps2Memory_t, 13, 30, 31, 33, 34, 35, 41
 sps2registers.h, 9, 15, 16, 19, 28, 42, 43
 sps2Release, 14, 15, 16, 19, 21, 22, 24, 25, 28, 29, 30, 38
 sps2Remap, 33, 34, 35
 sps2scratchpad.h, 9, 21, 45
 sps2tags.h, 9, 24
 sps2types.h, 9
 SPS2U_SCR_NTSC, 57, 59

SPS2U_SCR_PAL, 57, 59
 SPS2U_SCR_VESA, 57, 59
 sps2UNIAAllocate, 31, 46
 sps2UNIFlushCache, 36, 46
 sps2UNIFree, 33, 46
 sps2UNIGetPhysicalAddress, 35, 46
 sps2UNIInit, 28, 46
 sps2UNIRelease, 30, 46
 sps2UNIRemap, 34, 46
 sps2UNIWaitForDMA, 37, 46
 sps2UScreen, 24, 46, 56, 57, 58, 60
 sps2UScreenGetDrawBuff1, 51, 52, 53, 54, 55
 sps2UScreenGetDrawBuff2, 51, 52, 53, 54, 55
 sps2UScreenGetDrawBuffCurrent, 51, 52, 53, 54, 55
 sps2UScreenGetFirstFreeGSPage, 51, 52, 53, 54, 55
 sps2UScreenGetHeight, 56, 57, 58, 59, 60
 sps2UScreenGetPixelFormat, 57, 58, 59, 60
 sps2UScreenGetVideoMode, 47, 57, 58, 59, 60
 sps2UScreenGetWidth, 56, 57, 58, 59, 60
 sps2UScreenGetZDepth, 57, 58, 59, 60
 sps2UScreenGetZPtr, 51, 52, 53, 54, 55
 sps2UScreenInit, 24, 25, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60
 sps2UScreenShutdown, 24, 25, 47, 49
 sps2UScreenSwap, 24, 25, 50
 sps2util.c, 9
 sps2util.h, 9, 24, 46
 sps2vumemory.h, 9, 22, 44
 sps2WaitForDMA, 14, 15, 37
 STR, 14, 15, 37
 TAG_ID, 15
 TAG_IRQ, 15
 TAG_PCE, 15
 TIE, 15
 Timer, 16, 17
 tshower, 7, 9, 12
 TTE, 15
 uncached, 4, 31, 34
 Vector Unit, 4, 9, 22, 28, 30, 38, 41, 44
 VESA, 47, 57, 59
 video mode, 47
 VIF0, 16, 17
 VIF1, 16, 17
 virtual address, 31, 35
 virtual console, 47, 49
vspeed, 4, 9, 11
 VSync, 24, 50
 VU. *See* Vector Unit
 VU0_MEM, 22, 44
 VU0_MEM_OFF, 22, 44
 VU0_MICRO_MEM, 22, 28, 44
 VU0_MICRO_MEM_OFF, 22, 44
 VU1_MEM, 22, 44
 VU1_MEM_OFF, 22, 44
 VU1_MICRO_MEM, 22, 44
 VU1_MICRO_MEM_OFF, 22, 44
 XYZ2, 25, 26
 Z buffer, 47, 51, 52, 58