The Naked PSP

TyRaNiD
ps2dev.org

August 4, 2006

What This Presentation Covers

What I am going to talk about:

- Coding on the PSP using C
- PSP Internals
- Development tools

What I am not going to talk about:

- Games
- Piracy
- Modchips

Outline

Outline

Figure: Object of Desire: The PSP

PSP Time line

- December 2004: PSP released in Japan, 1.0 firmware
- March 2005: PSP released in USA, 1.5 firmware
- April 2005: First home brew, "Hello World!" by Nem. Version 1.0 only
- June 2005: PSP-DEV create method of running on 1.5 (kxploit)
- September 2005: PSP finally released in Europe
- September 2005: LibTIFF exploit, downgrader for 2.0 firmware
- April 2006: Home brew opened up via GTA for firmware greater than 2.0
- July 2006: Kernel bug in 2.5 and 2.6 discovered opened the way for more downgraders

Firmware Revisions

In the PSP's short history a number of firmware revisions have been released. Some purely to fix bugs being exploited for home brew software.

- Version 1.0 (Japan Only). Few restrictions on code execution
- Version 1.5 (USA). Use kxploit, then no restrictions
- Version 1.51, 1.52. No point hacking, downgrade from 2.0
- Version 2.0 (libTIFF/eLoader). User mode only unless downgraded
- Version 2.01 Bug fix release, upgrade to 2.5/2.6
- Version 2.5, 2.6 (GTA/eLoader). Kernel bug, access to kernel mode and downgradable.
- Version 2.7+. Not easy to develop on.

Running Your Own Code

How you run code depends on the firmware revision of the PSP

- For versions 1.0 and 1.5 build a special EBOOT.PBP file and run directly on the PSP
- For version 2.0 use eLoader in libTIFF mode
- For version 2.01+ use Grand Theft Auto UMD plus eLoader
- For version 2.7+ no known way of developing

Outline

Specifications

The PSP is quite a complex system containing a number of custom components

- MIPS System Controller (ALLEGREX)
- Media Engine (MIPS + VME + AVC decoder)
- 32MiB Main RAM
- 2MiB embedded DRAM for ME
- 2MiB VRAM
- General purpose audio system
- 480x272 full colour wide screen LCD (approx 1.8:1)
- 13 buttons, 4 direction digital pad, and analogue joystick for input
- Sony Memory Stick Duo port

Miscellaneous Hardware

Some hardware not normally needed for demo development

- 1.7GiB custom optical drive (UMD)
- USB function chip set, 802.11b Wireless Ethernet
- I2C bus for controlling the audio codec
- IRDA transceiver
- Multiple UARTS (IRDA, Remote Control, Debug Hardware)
- DMA controller for peripheral control (includes a general purpose DMA channel for user applications)
- Power and battery controller
- ATA controller for UMD drive

The ALLEGREX

The ALLEGREX is the name of the PSP's main CPU

- Customised MIPS32 core
- Instruction set additions (bit twiddling, user mode interrupt control, CPU halt)
- Limited Memory Management Unit (MMU) (no TLB present)
- Variable clocking (approx 33MHz to 333MHz)
- On board single precision (32bit) FPU
- On board vector/matrix co-processor (VFPU)
- Built in hardware debug unit
- 16KiB scratch pad RAM (pretty useless)
- On board profiler (monitor cache hits, instructions executed etc.)
- Separate instruction and data cache

Vector Processing

To supplement the 3D graphics capability of the PSP the main CPU has a vector co-processor (VFPU)

- 128 32bit registers
- Registers reconfigurable to operate as single values, 2x2/3x3/4x4 rows, columns, matrices, transposed matrices
- Can multiply whole matrices in one operation
- Usual selection of trigonometric/square root instructions
- On-board pseudo-random number generator

VFPU Registers

Figure: Register Mapping

Developing VFPU Code

VFPU is written in assembly, the Toolchain assembler supports all known VFPU operations.

```
/* Mult matrix $a0−result, $a1−a, $a2−b */
mult_matrix:
/*   Load matrices to internal registers */
  ulv.q C000,  0($a1); ulv.q C010, 16($a1)
  ulv.q C020, 32($a1); ulv.q C030, 48($a1)
  ulv.q C100,  0($a2); ulv.q C110, 16($a2)
  ulv.q C120, 32($a2); ulv.q C130, 48($a2)
/* Multiply matrices */
  vmmul.q M200, M000, M100
/* Store result */
  usv.q C200,  0($a0); usv.q C210, 16($a0)
  usv.q C220, 32($a0); usv.q C230, 48($a0)
```

The Graphics Engine

The Graphics Engine (GE) is the core of the PSP's graphics capability

- Modelled on a simplistic GL like API
- Drawing performed with in memory display lists
- 2MiB VRAM for frame buffer/textures/display lists
- Can operate in a bus master mode to pull in textures/lists from main RAM
- Only data which needs to be in VRAM are framebuffers

The Graphics Engine

Other features of the Graphics Engine

- Multiple texture formats, CLUT mode, DXT compression
- Can operate in a 2D or 3D mode
- Limited on-board 3D clipping
- Splines/bezier surfaces
- Object morphing and skinning
- Lighting and stencil buffers
- Alpha blending and logical operators

Display Lists

Drawing graphics can be done by hitting VRAM directly, for decent performance and 3D work you need to use the display lists.

- Each entry in the display list is 32bits wide, 8 bit command, 24bit data
- 24bit data can represent an integer, a pointer or a cut down float
- Can create sub-lists and which are called from your main list
- Can specify custom vertex types, different size (8/16 bit fixed point, 32bit float)
- Each vertex type can include, position, colour, normal or weight
- Vertices can be specified as a direct list or indexed
- Due to floats only being 24bit precision can be lost during transformation stage

Display Lists

The display lists are transferred to the GE using a DMA mechanism, this has important ramifications when developing the lists.

- A list can be transferred while building it using a 'Stall' address
- The list must be written using an uncached memory area or the data cache written back before use
- Lists should be terminated with a FINISH command

```
uint32_t list[256*1024]; // Define a display list

int qid = sceGeListEnQueue(list, list, cbid, NULL);
/* Fill in the display list */
sceKernelDcacheWritebackInvalidateAll(); // Writeback cache
sceGeListUpdateStallAddr(qid, &list[endp]);
```

PSP Memory Map

While the PSP's main CPU does not have a general purpose MMU there is the ability to do limited memory segmentation.

- Kernel/User mode specification, prevents a user application modifying kernel memory
- 32MiB main RAM split into 4MiB kernel only, 4MiB volatile memory, 24MiB user space
- 4MiB volatile memory only available through a system call (except on v1.0 where it is available always)
- Devices mapped into memory space
- There is no virtual memory, all memory space is accessible as long as you have the permissions.

PSP Memory Map

Some of the key addresses you will see when developing.

Starting Address	Size	Description
0x00010000	16KiB	Scratch pad
0x04000000	2MiB	VRAM
0x44000000	2MiB	VRAM (Cache through)
0x08800000	24MiB	Main RAM
0x48800000	24MiB	Main RAM (Cache through)
0x88000000	4MiB	Kernel RAM

Outline

Compilers

The PSP compiler was developed using the experience of maintaining the similar PS2 build tools.

- Compiler based on the 4.0.2 revision of the GNU Compiler Collection
- C and C++ development
- Support for ALLGREX specific instructions including VFPU
- Port of Newlib to support normal libc and stdc++ functions (stdio/iostream etc.)
- Designed for *nix like systems, to use on Windows requires either Cygwin or MingW (DevkitPRO)

The PSPSDK

A free software development kit to provide access to the libraries present on the PSP, comes with many libraries and tools to ease development.

- User and kernel library import tables with accompanying headers
- Reverse engineered version of libGU and libGUM
- Simple PCM audio library
- Debugging support libraries and utility functions
- Tools to build custom export tables, post process ELF files and convert executables to PRX
- Tools to pack and unpack the custom EBOOT.PBP format
- 60+ samples of using the SDK libraries and the PSP

Other Development Environments

Whilst C and C++ are the main languages used for developing PSP software you are not limited to using them.
Other options are:

- LUA (LUAPlayer) - Ported version of LUA with a PSP specific interface and libraries
- Python - Port of python with PSP specific extensions
- Flash - 2.7+ PSP firmware comes with a limited flash player

Modular Kernel

The PSP Kernel is built up of multiple separate 'Modules'. It is not a monolithic kernel in the usual sense. User applications are also considered modules.

- Modules can be loaded and started almost at will
- Can be loaded to fixed locations or relocatable
- Can export and import functions from other modules
- Kernel has no real concept of a 'process' as you would find in other operating systems.

Executable Formats

The PSP uses the Executable and Linking Format (ELF) as the base for
its modules

The kernel can load 3 types of files natively

- ELF - Basic ELF format, linked to a hard coded address (typically
  0x8900000)
- PRX - Customised ELF format, can be relocated anywhere in memory
  space
- ~PSP - An encrypted wrapper format for ELF or PRX

The PSPSDK is capable of producing ELF or PRX files but not the
encrypted ~PSP format

Module Information

In order to load and track an executable it must contain a module information section.

This consists of information such as :

- The module name
- Module attributes, for example if it is a kernel module
- Import and export tables
- Module version

PSPSDK does most of the work for you,
just specify PSP_MODULE_INFO in one of your source files.

Library Imports and Exports

In order to do something useful the kernel modules can export library sets to user mode applications.
The import of functions is done through special stub files, this is handled for you by the SDK

- User to kernel transitions are handled through a syscall gateway
- User to user calls are just resolved to direct jumps
- Functions are identified using a NID, for the kernel libraries this is the first 32bits of the function name's SHA1 hash
- You cannot link user libraries to kernel modules. This has caused alot of problems in the past.

While it is possible to export functions from your code using an export file, it is not normally necessary.

Loading Modules

Loading and starting new modules is theoretically quite simple, there are however a few important caveats.

- The locations where modules can be loaded from is quite limited in user mode
- While there are few restrictions in kernel mode you then hit a linking issue with some libraries
- The kernel cannot load up plain text kernel PRX modules, although ELF works fine
- The SDK contains patches to ease some of these restrictions

```
SceUID modid;
/* Load module */
modid = sceKernelLoadModule(path, 0, NULL);
/* Start module */
sceKernelStartModule(modid, args, argp, NULL, NULL);
```

Kernel Reset

As the PSP's kernel does not properly track resources for a module the best way to get the PSP into a clean state is to the reset the kernel. There are two normal ways of doing this:

1. sceKernelLoadExec - Reboot the kernel and start a new executable
2. sceKernelExitGame - Reboot the kernel back to the PSP's menu

The System Memory Manager

Perhaps the most important part of the kernel, it is loaded first after reboot.

Its tasks are as follows :

- Manages the PSP's memory allocation
- Segments main RAM into 3 allocation partitions (kernel/ME/user)
- Controls resource tracking (Unique Identifier Table)
- Handle system event notification
- Implements basic debug functionality

Resource Tracking

The PSP's kernel maintains alot of state information, to track this in a consistent fashion it uses a Unique Identifier (UID) Table.

- The UID table is maintained in heirachical tree structure
- Each UID can be assigned a name, this does not need to be unique
- Each kernel module can specify its own UID types
- Each entry holds a control block containing information specific to the type
- Kernel can call operations on a UID to perform specified tasks.
- UID values aren't quite as random as they look, it is actually an encoded address.

```
void *cntladdr = 0x88000000 + ((uid >> 5) & ~3);
```

Thread Overview

The threading model on the PSP is quite simple. It is similar to the model used in the PS2's IOP kernel.

- Threads are cooperative
- Scheduling based on thread priority, the lower the number the higher the priority
- Small amount of thread local storage (TLS), referenced with the CPU register K0.
- To use the VFPU in your code the thread must be created with the PSP_THREAD_ATTR_VFPU attribute

Thread Creation

Threads must first be created then started. Once a thread is created it is given a UID with which further actions can be performed.

```
int my_thread(SceSize args, void *argp)
{ /* Do something */ }

SceUID thid;
/* Create a user thread */
thid = sceKernelCreateThread("MyThread",     // Name
                             my_thread,      // Entry Point
                             20,             // Priority
                             16384,          // Stack size
                             0,              // Attribute
                             NULL);
/* Start thread with arguments */
sceKernelStartThread(thid, args, argp);
```
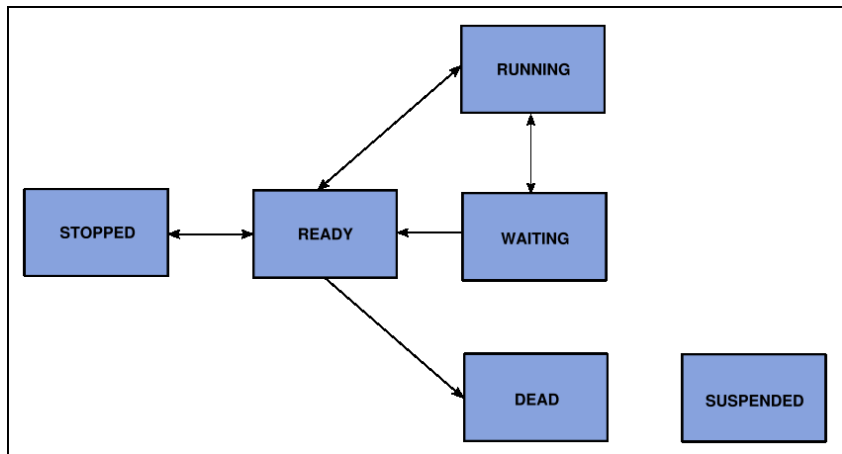
Thread States

Figure: Simplified State Transition Diagram

Synchronization Primitives

The PSP kernel provides a number of ways of performing thread synchronization and control. All waits can have an optional timeout.

- Mutual Exclusion
  - Semaphores
  - Interrupt Control
- Message Posting
  - Event Flags
  - Message Boxes and Pipe
  - Callbacks
- Synchronized Memory Pools
  - Variable Pools
  - Fixed Pools
- Timers
  - Alarms
  - Virtual Timers

Mutual Exclusion

Semaphores are synchronized counters, there is no mutex type on the PSP but semaphores can be used to simulate them.

- Creation and destruction using sceKernelCreateSema, sceKernelDeleteSema
- Locked and unlocked with sceKernelWaitSema, sceKernelSignalSema

Disabling interrupts can be used to do low cost mutual exclusion

```
int intc;
intc = pspSdkDisableInterrupts();
/* Do something to your data */
pspSdkEnableInterrupts(intc);
```

Event Flags

Event flags allow you to post 'events' between 2 or more thread.

- An event flag consists of a 32x1bit events
- One or more bits can be waited on at one time
- Bits can be set to auto or manual reset modes

```
SceUID evid;
evid = sceKernelCreateEventFlag("Event", // Name
                                0,        // Initial pattern
                                0,        // Attributes
                                NULL);
void th_one() { sceKernelSetEventFlag(evid, BITMASK); }
void th_two() {
 sceKernelWaitEventFlag(evid, BITMASK,
  PSP_EVENT_WAITCLEAR | PSP_EVENT_WAITOR, NULL, NULL);
  /* Event posted, do something */
}
```

Callbacks

Callbacks are used by the kernel to send special events in a thread context.
A thread must either poll for callbacks occurring or call a special wait
function, normally suffixed with CB.

```
int exit_cb(int a, int b, void *p)
{ sceKernelExitGame(); return 0; }

/* Create callback */
SceUID cbid;
cbid = sceKernelCreateCallback("Exit", exit_cb, NULL);
sceKernelRegisterExitCallback(cbid);

/* Sleep and wait for callback */
sceKernelSleepThreadCB();
```

Interrupts

The kernel provides a mechanism to provide a callback for certain
interrupts in user mode.

- Interrupt callbacks called in thread independant state
- Do not call functions which could wait

```
int vblank_handler(void *p)
{ /* Do something */ }

sceKernelRegisterSubIntrHandler(PSP_VBLANK_INT,
                                1, vblank_handler,
                                NULL);
sceKernelEnableSubIntr(PSP_VBLANK_INT, 1);
```

Thread Programming Tips

Some tips for making thread programming easier:

- Never sit in a tight loop, where possible enter a wait state
- Choose thread priorities carefully, give higher priority to interrupt driven threads
- Always check return codes of wait system calls
- When calling a waiting function use the timeout facility to detect deadlocks
- If you are finished with a thread delete its resources

File IO

The kernel provides a general purpose IO subsystem to access files on
available devices

- Different types of IO devices can be created
- A file IO device accessed through a device prefix
- Allows aliases to be setup
- Block devices can be mounted into file system devices using
  sceIoAssign
- POSIX like system calls (e.g. sceIoOpen for open, sceIoWrite for
  write etc.)

File IO

File IO device prefixes consist of two parts

- The device name
- Optional file system unit number

Some examples are as follows :

ms0:	Memory stick file system
host0:	Host file system (PSPLink extension)
flash0:	Flash file system
irda0:	Character driver for the IRDA transceiver
tty0:	Text terminal character driver

Graphics Libraries

For basic graphics work such as directly writing pixels to VRAM only the display library is needed.

- Can setup frambuffer in 16bit (565, 5551, 4444) or 32bit colour.
- Setup frame buffer with sceDisplaySetFrameBuf
- sceDisplayWaitVblankStart puts the current thread into a wait state until the next vblank interrupt, useful for synchronization and to allow other threads to run

```
/* Setup display mode */
sceDisplaySetMode(0, 480, 272);
/* Set frame buffer to 32bit colour */
sceDisplaySetFrameBuf(0x4000000, 512, 3, 1);
```

Graphics Libraries

libGU is the mainstay of developing hardware accelerated graphics for the PSP.
It is based on a reverse engineered version of the official Sony libraries.

- It acts as a wrapper around the GE and display hardware
- Automatically manages display lists, can create online or offline lists
- Handles setting up of the framebuffer and flipping

libGUM is a companion library to provide matrix manipulation for direct use by libGU.

- Comes in floating point and VFPU flavours
- Maintains a matrix stack similar to GL
- Provides utility functions to setup projection matrices, rotate vectors etc.

Graphics Libraries

LibGU and libGUM are not the only ways to display graphics on the PSP. Some important libraries available are:

- PSPGL, a hardware accelerated library with a GL style interface
- SDL, useful for 2D graphics

Other libraries ported to the PSP for use with graphics display.

- libPNG, PNG decoding
- libJPEG, JPEG decoding
- FreeType, TTF font library

Audio Libraries

The PSP's audio hardware is exported by a number of libraries. In combination with the ME it can output atrac3 audio with virtually no CPU usage.

- PSPSDK comes with a simple streaming audio library. A callback is invoked when more data is required.
- The kernel's atrac3 decoder can be used, encoding in atrac3 is so far only available on Windows platforms
- Ports of libmad, libogg/libvorbis for MP3 and Ogg/Vorbis decoding respectively
- Port of mikmod for module playback, there is a port of fmod but it is commercial only

Joypad

Using the joypad is simple affair, although not directly useful for demos it can be useful for debugging.

- All digital buttons are mapped in a button bitmask
- The analogue stick is mapped to two unsigned 8bit words

```
SceCtrlData pad;
/* Initialise the pad, analogue mode */
sceKernelSetSamplingCycle(0);
sceKernelSetSamplingMode(PSP_CTRL_MODE_ANALOG);

for(;;) {
    sceCtrlReadBufferPositive(&pad, 1);
    /* Do something with pad.Buttons or pad.Lx/Ly */
}
```

What security measures?

The designers of the PSP implemented a number of security measures in order to prevent non-official code taking over the system.

- Hardware crypto engine
- Chain of trust boot process
- Kernel/User separation to mitigate bugs in game software
- Kernel places limits on the types of modules which can be loaded and from where
- Custom encrypted/signed wrapper format for executables and boot lists

PSP Boot process

As the PSP's kernel is loaded from a Flash ROM it might be possible to
replace it, the designers chose to use a chain of trust to prevent
replacement of the kernel.

- Kernel is stored encrypted in Flash ROM
- The PSP kernel boots with the following steps:
    1. Embedded bootstrap decrypts stage 1 IPL from flash
    2. Stage 1 IPL decrypts stage 2 loader from memory
    3. Stage 2 loader decrypts and loads kernel from flash
- Stage 2 loader checks some critical modules before running them
  (1.5+ only)
- Difficult if not impossible to create a customized firmware from
  scratch

Thread Security

Threads have a privilege level which indicates if they can run in kernel or user mode, this allows protection of kernel memory from user applications.

- When creating a new thread you cannot escalate its privilege level
- Thread primitives inherit permissions of the thread which created them
- Cannot enumerate or use thread primitives without the same privilege
- During syscall transition a separate allocated stack is used to prevent attacks or data leakage
- All kernel functions which take pointer arguments check them based on the threads privilege.

Executable Loading from a user application

Different privilege levels handle module loading differently, kernel mode for example does few checks.

An example of how a module is loaded is as follows:

1. Step 1. Check type of executable (user/kernel/encrypted/not encrypted) etc.

2. Step 2. Check we are allowed to load from the specified device for this type (user mode can only load from UMD)

3. Step 3. Load (and possibly decrypt) executable into memory

4. Step 4. Start module with a thread of the correct type (e.g. kernel for kernel mode)

Is it really secure?

Security is only as good as its weakest link

- PSP's weakest link is its software, the only protection is really not being able to access kernel mode
- Comes down to security through obscurity
- Once PSP is running only two software flaws are needed to full system control
  1. Exploit a game to run your own code (for example GTA). User mode control.
  2. Exploit kernel to control whole system (2.5/2.6 LoadExec bug)
- V1.0 and V1.5 didn't have any security at all, would load ELFs in kernel mode directly

Outline

Debugging

When coding in C/C++ it is almost inevitable that bugs will creep in making debugging mandatory.
Being a consumer facing device there is no real debug capability directly available.

- Default action of exception handlers is to spin loop, eventually PSP powers off
- No immediate means of outputting debug text bar the screen

Of course all is not lost, there are ways of debugging without requiring an expensive devkit from Sony

Blind Debugging

Originally the only option was blind debugging, build your application, copy to the PSP and run it.
There are some features in the SDK to aid in this process (some only for 1.0 and 1.5 firmware)

- On screen and via SIO textual output
- Basic SIO based GDB stub
- Functions to add a user specified exception handler
- Miscellaneous functions to do call stack traces, enumerate threads etc.

Not exactly an efficient development process, all applications must be on memory stick. Might be only option on 2.00+.

PSP Inside

A development and hacking tool created by Hitmen.

Figure: PSP Inside in operation

- Displays the running state on the PSPs screen
- Can disassemble instructions in real time
- Can load up new modules from memory stick/flash etc.

PSPLink

PSPLink is a development tool which provides a text terminal on the PSP. Some of the features it provides are:

- Terminal access through SIO, Wifi or USB.
- Access to the PC's file system through USB and the host: device
- Preconfigured tty (stdin, stdout, stderr)
- Exception handler for catching software bugs
- Remote GDB stub over Wifi and USB for source level debugging
- Real time kernel inspection (Thread/Module lists, memory dumping and patching)

Exceptions

When the inevitable crash occurs PSPLink or PSP Inside will normally split out a CPU exception. From this information the location of the crash can be tracked and fixed.

- Always build you application with the -g compiler switch to enable debugging information
- The exception will give an indication of what caused it
- The exception will also print the EPC register, this is the memory address at which the exception occurred
- Pass the value of EPC to psp-addr2line to get a line number in your executable
- If it comes to it try and use GDB

```
psp-addr2line -e program.elf 0x08900340
main.c:68
```

Resources and References

Some useful places for information on PSP programming

- http://www.pspdev.org - Home of the toolchain and subversion archives
- http://psp.jim.sh/pspsdk-doc - Online version of the PSPSDK docs
- #pspdev on FreeNode IRC - We might help you, maybe.