Playstation2 Basics

"Welcome to the Machine"

Version 1.0, 2002/03/06 - Tony Saveski (dreamtime), t_saveski@yahoo.com

Introduction

Welcome to the first of the Playstation2 Development (ps2dev) Tutorials. I don't want to waste any time on fluffy introductions to the 'scene' and the PS2, so after a quick intro I'll get right into what everyone is reading this for.

Just a note first though: This tutorial is intended for the almost total newbie to PS2 development. If you're already compiling and running your own code you probably won't learn much here :-)

There is already a fair bit of information available on the net about programming the PS2. Some people and groups have even released small demos and source code. Many thanks to them! But the main thing missing is detailed explanations of what's going on in this code, so that the less experienced of us can learn and join in the fun.

I have to admit I am completely new to console development. I have never programmed a MIPS chip before and my knowledge of hardware is almost non-existent. This should be good for the reader because it means I won't be able to confuse you with detailed and cryptic explanations (although I have been called a 'vague bastard' in the past).

My aim is to carefully understand and fully explain everything that goes into achieving some programming task on the PS2 and end up with a clean and well-documented code base that I can share with everyone. Where things are not clear to me, I will put a note asking for someone to '*please explain*'. If I am totally wrong about something, and I'm sure that sometimes I will be, please tell me. I will make an effort to keep the tutorials up to date and as accurate as possible.

So, let's get started with some baby-steps. This tutorial will provide you with knowledge of basic PS2 hardware components and interfaces, and code for:

* Initialising the graphics system,

* Setting a video mode,

* Making something show up on the screen.

In the process, we will also end up with a set of DMA and other routines that we will reuse many times in the future.

Development Environment

Before you can start testing your code on a PS2, you will need to set up a development environment. I have chosen not to write anything about this, but instead point you to an

excellent tutorial currently maintained by *now3d* at *http://ps2dev.sourceforge.net/guide.txt* .

The main components you will need to have are:

* A USB to USB cable,

* A working set of compilers (GCC),

* Naplink.

More detailed information about setting up compilers and using Naplink can be found in the *Tutorials* section of *http://ps2dev.livemedia.com.au* (awsome site by *Oobles*).

Once you have everything set-up, compile some of the already available demos and run them in your environment. You can find the source code to some of these at *LiveMedia*.

By the way, I learned almost everything described in this tutorial from the *3stars* and *funslower* demos. Go out and get them now!

The PS2 Hardware

The main component of the PS2 is probably the Emotion Engine (EE). The EE contains, among other things, the MIPS R5900 CPU; two high performance Vector Units (VU); and the DMA Controller. We will look at the VU's in a future tutorial.

Connected to the EE is the PS2's Graphics Synthesizer (GS). The GS is accessed from the EE via the GIF (or Graphic InterFace).

MIPS R5900 CPU

The R5900 is a RISC chip based on a MIPS architecture (*http://www.mips.com*). It implements the full MIPS III ISA, some MIPS IV instructions, and a proprietary set of multimedia instructions. (TODO: *Sure it's not a full MIPS4?*)

While not a relatively powerful CPU, is serves its purpose of allowing us to load and run our programs, handle interrupts, and communicate with the other devices in the system.

Information about programming with the MIPS ISA, including a full instruction set reference, is freely available from the MIPS website (*http://www.mips.com/publications/index.html*).

Another great document about programming MIPS using Assembler and C is provided by IDT, the creators of the Playstation 1 core (R3000), at *http://decstation.unix-ag.org/docs/ic_docs/3715.pdf*. While the document focuses on the MIPS III architecture only, it contains LOTS of useful information about the MIPS CPUs, that you would have to otherwise pay for by buying books like "*See MIPS Run*" by Dominic Sweetman (ISBN: 1558604103), which I can't seem to find or order anywhere in Australia :-(

Another good reason to read the IDT document is because the PS2 contains one of these chips in addition to the R5900! They call it the IOP (Input Output Processor), and it is used to communicate with all external peripherals such as the controller, keyboard, and mouse. It is also how they achieve backward compatibility with the old Playstation to let you play the old games. Again, more will be said about this chip and I/O in a future tutorial.

As an inexperienced MIPS programmer, I have found that looking at the code generated by the GNU compiler is a great way to learn to program MIPS assembler (or any assembler for that matter). Just put a *-S* switch on your GCC command line to create a *.s* file containing the generated ASM code.

Graphics Syntheizer (GS)

The GS is programmed using a generous set of registers.

Some of these registers are mapped to the EE Core address space and can be accessed directly from the R5900 CPU. They are called the "Privileged GS Registers".

The other GS registers can only be accessed via the GIF. It's not possible to access the GIF directly either. Instead, we have to create 'GS Packets' containing a 'GIF tag' and data (it's just an array of register IDs and data) to send to the GIF using DMA transfers. In this tutorial we will only be looking at how to create these structures using the standard CPU, and leave the high-speed Vector Units for a future tutorial.

Together, the GS registers provide a fairly high level API for programming the GS. It is much easier to understand and program than the SVGA card on a PC (no banking...woohoo), but detailed documentation is sorely lacking. Hopefully these tutorials can compliment what's already available.

PS2 Data Types

OK. So let's write some code. The first thing is to establish some basic data types to use in our C/C++ code. Full source code should be available at the same place you got this tutorial. If it isn't, then you can always rely on finding it at *LiveMedia*.

Descriptions of PS2 registers and operations center around the EE's word size. For example, the GS registers are all a DoubleWord (dword) in size; while DMA data is transferred in 8 QuadWord (qword) ‘slices’. Given that the word size is 32 bits, this table summarizes the different data sizes, their common names and assembler names.

Name (asm)	Size (bits)	Size (bytes)
Half Word (.half)	16	2
Word (.word)	32	4
Double Word (.dword)	64	8
Quad Word	128	16

I prefer not to use data-type names relevant to a particular machine because it makes porting to machines with a different word size difficult. Instead, I use explicit names based on the data-type size in bits. Here are the definitions (from *defines.h*):

```
typedef char   int8;
typedef short  int16;
typedef int    int32;

typedef unsigned char  uint8;
typedef unsigned short uint16;
typedef unsigned int   uint32;

typedef unsigned long  uint64;
typedef long           int64;

typedef struct int128
{
    int64 lo, hi;
} int128 __attribute__((aligned(16)));

typedef struct uint128
{
    uint64 lo, hi;
} uint128 __attribute__((aligned(16)));
```

The *-mips3* switch on the *gcc* command line ensures that the *long* data-type is 64 bits. Don't forget it.

PS2 Register Definitions

The *regs.h* file contains definitions of all the registers needed for this tutorial. We will be adding to this list in the future. This file is written to be usable in both C and ASM code.

To use these macros, compile your assembler files (*.s* extension) with *gcc* and include the *-xassembler-with-cpp* switch. This tells *gcc* to pipe your source file through the C pre-processor before passing it to the assembler. In this way, you can use the nice meaningful register names instead of their ugly IDs.

I'll only be describing registers actually used in the sample code for this tutorial.

MIPS CPU Registers

The first block of code defines the standard MIPS registers using their common logical names (their actual names are $0 - $31). The IDT assembler manual has a great description of recommendations on using each register, and how they are generally used by C-compiler generated code.

```
#define zero   $0      // Always 0
#define at     $1      // Assembler temporary
#define v0     $2      // Function return
#define v1     $3      //
#define a0     $4      // Function arguments
#define a1     $5
#define a2     $6
#define a3     $7
#define t0     $8      // Temporaries. No need
#define t1     $9      // to preserve in your
#define t2     $10     // functions.
#define t3     $11
#define t4     $12
#define t5     $13
#define t6     $14
#define t7     $15
#define s0     $16     // Saved Temporaries.
#define s1     $17     // Make sure to restore
#define s2     $18     // to original value
#define s3     $19     // if your function
#define s4     $20     // changes their value.
#define s5     $21
#define s6     $22
#define s7     $23
#define t8     $24     // More Temporaries.
#define t9     $25
#define k0     $26     // Reserved for Kernel
#define k1     $27
#define gp     $28     // Global Pointer
#define sp     $29     // Stack Pointer
#define fp     $30     // Frame Pointer
#define ra     $31     // Function Return Address
```

Privileged GS Registers

As mentioned above, these registers are actually mapped to the EE's memory address space. Writing values to the addresses defined here will actually set the value of the GS register. These registers are all 64 bits wide.

```
#define pmode      0x12000000 // Setup CRT Controller
#define smode2     0x12000020 // CRTC Video Settings
#define dispfb1    0x12000070 // RC1 data source settings
#define display1   0x12000080 // RC1 display output settings
#define dispfb2    0x12000090 // RC2 data source settings
#define display2   0x120000a0 // RC2 display output settings
#define bgcolor    0x120000e0 // Set CRTC background color
#define csr        0x12001000 // System status and reset
#define imr        0x12001010 // Interrupt Mask Register
```

General Purpose GS Registers

These registers can only be set via the GIF using DMA transfers. The values defined in this block of code are used to tell the GIF which register to write each chunk of DMA data to. The registers are all 64 bits wide.

```
#define prim       0x00   // Set current drawing primitive
#define rgbaq      0x01   // Setup current vertex color
#define xyzf2      0x04   // Set vertex coordinate
#define xyz2       0x05   // Set vertex coordinate and 'kick' drawing
#define xyoffset_1 0x18   // Prim to Window coord mapping (Context 1)
#define xyoffset_2 0x19   // Prim to Window coord mapping (Context 2)
#define scissor_1  0x40   // Setup clipping rectangle (Context 1)
#define scissor_2  0x41   // Setup clipping rectangle (Context 2)
#define frame_1    0x4c   // Frame buffer settings (Context 1)
#define frame_2    0x4d   // Frame buffer settings (Context 2)
```

DMA Channel Registers

The PS2 has ten DMA channels (0-9) used to send data to and from different devices. Each channel has the same set of 32 bit registers, but mapped to different addresses (some channels don't actually have all registers).

All DMA transfers from EE to GIF are done through DMA Channel 2 and I have only defined the registers used to control this channel for now.

```
#define gif_chcr   0x1000a000 // GIF Channel Control Register
#define gif_madr   0x1000a010 // Transfer Address Register
#define gif_qwc    0x1000a020 // Transfer Size Register (in qwords)
#define gif_tadr   0x1000a030 // ...
```

Setting the Register Values

To make the PS2 do things, we need to write values to the registers defined above. This can be a complicated task, as each part of a register generally has a different meaning.

Privileged GS Registers

Let's take the *bgcolor* register as an example. It is a 64-bit register used to set the background color of the CRT controller, where: bits 0 to 7 represent the Red component; bits 8-15 represent the Green component; and bits 16-23 represent the Blue component.

To simplify the task of setting the register, I have defined a set of convenience macros in the *gs.h* file. For each privileged GS register, you will find a macro like this:

```
#define BGCOLOR    ((volatile uint64 *)(bgcolor))
#define GS_SET_BGCOLOR(R,G,B) \
*BGCOLOR = \
((uint64)(R)   << 0)      | \
((uint64)(G)   << 8)      | \
((uint64)(B)   << 16)
```

The macro accepts a parameter for each register field, shifts each by an appropriate number of bits, and performs a bitwise OR of all the fields to come up with the 'magic number' to assign to the register.

A cool side effect of using this method to set register values is that if any macro parameter value is 0, *gcc* will not generate the code to needlessly shift the 0 and OR it, even when you tell it to not optimize (*-O0*). Go ahead and try it! Some other techniques I've seen would generate code to assign 0 to the fields.

The *volatile* keyword tells the C compiler not to optimise access to and from the memory location pointed to. The value is always explicitly read just before usage, and written to immediately when asked to. This ensures that if the value of the register mapped to this address is changed by another thread or interrupt handler, we will always read the current value and not some 'cached' value that is out of date.

General Purpose GS Registers

These registers have to be set using DMA transfers, so we can't write GS_SET_XXX() macros for them to set their values directly. We do however need macros to simplify the task of creating the 'magic number' to assign to the DMA buffers to be sent across.

For each General Purpose GS register (well, the ones I could be bothered doing right now), the *gs.h* file contains a macro similar to the Privileged Register macros:

```
#define GS_FRAME(FBP,FBW,PSM,FBMSK) \
(((uint64)(FBP)    << 0)      | \
 ((uint64)(FBW)    << 16)     | \
 ((uint64)(PSM)    << 24)     | \
 ((uint64)(FBMSK)  << 32))
```

DMA Registers (GIF Channel)

The DMA register setting macros are in *dma.h*. They are similar to the Privileged GS macros, except the first parameter is always the volatile address of the register you want to set. This allows us to use the same macro in the future, when we define the addresses of the other channel registers.

```
#define GIF_MADR      ((volatile uint32 *)(gif_madr))

#define SET_MADR(WHICH,ADDR,SPR) \
*WHICH = \
((uint32)(ADDR)    << 0)      | \
((uint32)(SPR)     << 31)
```

Shameless Self Promotion

When I first started playing around with the PS2 I had a hard time understanding what people were assigning to registers in their undocumented code, and it was hell trying to work out the values of each register field. I wrote a Windows tool to let me generate these 'magic numbers' assigned to registers by filling in the values of each register component. In *Version 1.1* I added the ability to reverse this process and get each field's value given a 'magic number'. I still find it very useful.

The tool name is *PS2REG* (very original), and you can get it from the *LiveMedia* site.

Code Discussion

I can hear you screaming out, “GET TO THE POINT!”. Well, we now have enough information and code to start doing the interesting stuff.

I was originally planning on pasting the source code here and talking about it line by line. I think that would probably be way too boring for you (and take way to long for me to type it up, in addition to driving me crazy).

Instead, I will spend the next few sections only talking about some interesting and/or important things I've discovered by playing around with different registers. The rest is pretty self-explanatory by looking at the source code (don't they always say that). I've included some pretty detailed comments to help.

File Name	Description
defines.h	Contains common data type definitions.
regs.h	Defines the addresses and id's of the PS2 registers used throughout the code.
ps2.h, ps2.c, ps2_asm.s	This set of files contains any general PS2 'system' functions.
gs.h, gs.c, gs_asm.s	GS Specific functions and macros.
dma.h, dma.c, dma_asm.s	A bunch of DMA routines and macros.
gif.h	Contains macros to make creating GIF tags and buffers easier.
g2.h, g2.c	A 'high level' 2D graphics library, making use of all the above, to be used by the demos.
demo1.c	The final 'demo', which uses the G2 library.

Note: I've named the *xxx_asm.s* files that way to avoid accidentally overwriting the *.s* files when telling *gcc* to generate an assembler listing of a C file named *xxx.c* (as would happen if they were named just *xxx.s*).

Initializing The Graphics Mode - g2_init( )

When initializing the graphics mode of the PS2, I don't think it's immediately obvious that there are two bits of graphics hardware that need to be set up, so it might initially seem like you're redundantly setting up the same thing twice. The one you might not have been aware of is the *CRT Controller* (CRTC).

CRTC Initialization

The signal that is sent from the PS2 to your TV or Monitor is controlled by the CRTC. You must tell the CRTC whether you want the signal to be PAL or NTSC, and Interlaced or Non-Interlaced (among other things).

This is all controlled using the SMODE2 register. In the source code, I actually use a PS2 system call to set these values, mainly because I couldn't get it to work properly with just the SMODE2 register only (although I havn't tried recently). I hope someone can explain what syscall 0x02 is doing in addition to setting the SMODE2 register (come on all you 'BIOS hackers'). Note that it's not necessary to set the SMODE2 register if you've used the syscall, as all the demos I've seen so far seem to do.

Rectangular Area Read Circuits

The CRTC contains two Read Circuits (RC1 and RC2) that can each concurrently read a different region of display memory.

The *DISPFB_1* and *DISPFB_2* registers are used to tell each read circuit: where to read its image data from (in the 4meg GS memory space); how 'wide' each row of data is; and what format the pixels are in.

It is possible to mix the output of the circuits in a number of interesting ways before sending the final image to the screen. I'm sure some cool effects can be achieved using this feature. Maybe someone will write a tutorial about this in the near future.

One use I can think of is to implement something like the Quake console. You would set up two frame buffers (assume no double buffering of animation :-): a 32-bit one for the 3D game graphics; and a low color resolution, simple 2D buffer not using up much

memory. You could have debug and other info going to the text frame, but not being displayed until the user hits the 'Console Key'. You could then turn on mixing of the two RC's, using some wonderful blending method. Animated, of course.

I'd like to hear about any other ideas people have had about effects with the RC's.

The Magical DISPLAY Registers

Whereas the *DISPFB_x* registers tell the RC's where to get their data from, the *DISPLAY_x* registers tell them where on the output device to put the image.

The demos I've seen to date don't explain what they're assigning to these registers, and set some almost arbitrary-seeming 64-bit 'magic number'. Even after using my trusty *PS2REG* tool (get it at *LiveMedia*!) the numbers assigned to some fields seemed random to me. Here's what I've been able to figure out about each field from doing some experimentation (and some bare-bones and scattered documentation from Sony):

dx - X position on the screen where image output starts, specified in VCK units. What the f*ck are VCK units? Read on.

dy - Y position on the screen where image output starts, specified in Raster units (I always set it to Height in Pixels for the video modes in G2).

The *3stars* demo sets the above two values to (656, 36). This does make the image start at a nice place on the screen, but I don't understand why (0, 0) doesn't actually equal the very top-left of my monitor (via a crappy TV tuner card). Can someone explain this?

magh - Horizontal magnification of image. In PAL and NTSC mode the value you should put here depends on the Width of the screen mode in pixels. The goal is to make the total width fit nicely across the whole screen by ‘stretching the pixels’.

Mode Width (pixels)	MAGH Value
256	9 (10 times)
320	7 (8 times)
384	6 (7 times)
512	4 (5 times)
640	3 (4 times)

magv - Vertical magnification. I don't think we'd ever need to set it to anything other than zero.

dw - Display Area Width - 1, in VCK units. By observing the behaviour of setting different values here and in the MAGH field, I've worked out the following things about VCK units and video modes in general:

* A VCK is a unit of measure of a constant horizontal distance across the screen (like a pixel).

* You should set this value to: GraphicModeWidth * MAGH - 1. If you multiply all the 'standard' screen widths above with the recommended horizontal magnification amount, it will always equal about 2560 VCK's.

* You can actually do whatever you want with these values, even create a custom 'video mode', as long as you consistently set-up the CRTC and the GS. For custom modes you can probably calculate the MAGH value as (*2560 / CustomWidth*) and round down to the nearest integer or something.

* You could probably do some sort of split screen graphics mode using the two CRTC Read Circuits and two different frame buffers.

dh - Display Area Height - 1, in Pixels.

GS Initialization

Once the CRTC is set up and knows where to get it's image data from and what format that data is in, we pretty much need to tell the GS the same information. Only this time, the info is used by the drawing routines to control where in display memory the output goes (to be subsequently picked up by the CTRC for drawing to the screen).

You will notice that I've put the (one and only in this tutorial) frame buffer at the start of the GS video memory, and told both the CRTC (using *DISPFB_2)* and the GS (using *FRAME_1*) the same things about its location and pixel format.

I've only set up the bare minimum of GS registers to be able to draw in 2D and keep this tutorial simple.

GS Packets, GIFtags and DMA Transfers

Now we're ready to start drawing things to the screen. Briefly, the process for sending drawing commands to the GIF is as follows.

First, set up a qword-aligned buffer that contains a 128bit GIFtag, and a set of 128bit Address+Data entries. This structure is called a *GS Packet*.

In *gif.h* you will find a number of macros that can simplify this task (the *funslower* demo used a similar technique, thanks for the idea guys).

DECLARE_GS_PACKET(NAME, ITEMS) will generate C code to correctly create an appropriate array large enough to hold a single GIFtag and *ITEMS* A+D entries.

BEGIN_GS_PACKET(NAME) should be called when you want to populate up the buffer with data.

GIF_TAG_AD(NAME, NLOOP, EOP, PRE, PRIM, FLG) should be the first thing called after *BEGIN_GS_PACKET()*. It sets up the GIFtag for Address+Data operation, which tells the DMA controller that the GIFtag is followed by *NLOOP* A+D entries. You must set *NLOOP* with the number of A+D commands that follow, and set EOP to 1 if this is the last GS Packet.

GIF_DATA_AD(NAME, REG, DATA) is used to set up each Address+Data entry in the GS Packet. REG needs to be set to the GS Register ID you want to assign the DATA to. You should use the convenience macros defined in *gs.h* to set the DATA value.

Finally, ***SEND_GS_PACKET(NAME)*** is called to kick off the DMA transfer, using the buffer you just set up.

The DMA transfer process is very simple.

* Flush the EE's data cache.

* Set the QWC register to equal the number of qwords in the buffer you just set up (The GIFtag and NLOOP A+D entries).

* Set the MADR register with the address of the buffer.

* Tell the CHCR register to start the transfer.

* If we like, we can then do nothing until the STR bit of the CHCR register becomes 0, meaning that the transfer is complete.

You can find all this code in *dma.h*.

Drawing Routines

To save you the hassle of doing all the above by yourself, I've created a collection of simple 2D drawing routines (see *g2.h* and *g2.c*).

If you have understood everything in this tutorial, and have had a look at the code, then the G2 library drawing routines will be totally trivial for you and there's no point in me explaining them (u can tell I really want to finish this thing :-)

Here is are the prototypes of what's implemented so far (very Borland-like for those of you who remember):

```
int g2_init(g2_video_mode mode);
void g2_end(void);
uint16 g2_get_max_x(void);
uint16 g2_get_max_y(void);
void g2_set_color(uint8 r, uint8 g, uint8 b);
void g2_set_fill_color(uint8 r, uint8 g, uint8 b);
void g2_get_color(uint8 *r, uint8 *g, uint8 *b);
void g2_get_fill_color(uint8 *r, uint8 *g, uint8 *b);
void g2_put_pixel(uint16 x, uint16 y);
void g2_line(uint16 x0, uint16 y0, uint16 x1, uint16 y1);
void g2_rect(uint16 x0, uint16 y0, uint16 x1, uint16 y1);
void g2_fill_rect(uint16 x0, uint16 y0, uint16 x1, uint16 y1);
void g2_set_viewport(uint16 x0, uint16 y0, uint16 x1, uint16 y1);
void g2_get_viewport(uint16 *x0, uint16 *y0, uint16 *x1, uint16 *y1);
```

Our First ‘DEMO’

Again, very Borland-like. It goes through some of the drawing primitives and randomly draws a few thousand of them. Very boring, but it's a start and we can all understand the code.

Note that I've had to slow the demo down (in the *crap_rand()* function). I was surprised at how fast everything was being drawn, even though I'm drawing everything the 'slow way'.

Crap Antialiasing-like Junk Around Primitives

I use a crappy cheap TV tuner card in my second PC (which has a crappy cheap 14" monitor) connected to a crappy cheap VCR that my Playstation2 is connected to. This is where I do all my development and testing.

When running code on this setup, the lines and filled rectangles being drawn look pretty crap. It looks as if the PS2 is doing antialiasing-or-blending-gone-wrong even though the drawing commands tell it not to do any.

Some people on #ps2dev have experienced similar junk and say that when they run their demos on a better output device, it all looks OK. I havn't had a chance to run this code on a decent TV, so I hope this is the case and not just crap code from me.

Conclusion

We’ve covered quite a bit of ground in this tutorial and I hope you’ve been able to learn something new. I intentionally kept the 2D routines and DEMO for this tutorial simple so we could concentrate on understanding how the PS2 works. I promise this will be the

most boring tutorial of the set because there was so much background information and ‘supporting’ code to develop (that you probably already knew).

In the next tutorial I will be extending the G2 library with the following features:

* Routines for handling BMP and/or PCX files.

* Display an image to the screen.

* Define a simple Sprite format and write a ‘blit’ function that can handle transparent regions.

* Set up a double buffering system for animation.

I’ll also write a more interesting animated demo to tie it all together (maybe an animated 'line' if we’re lucky :-)

But I don't think I'll write so much text to describe it all next time. It takes too much time away from coding (I knew there was a reason I hated doing documentation!).

I'd love to hear what you thought of this tutorial. Please send any comments, corrections and/or suggestions to *t_saveski@yahoo.com.*

Cheers.