PlayStation Specifications - psx-spx

None

None

None

Table of contents

1. Home

1.1 IMPORTANT UPDATE

On the 20th of August 2022, Martin surprisingly released a new version of this documentation. While this fork will try to incorporate the changes, one important footnote that got added is the following:

> I am homeless in Hamburg, please help me out!

The authors of this fork thought that this deserves more than a footnote, hence this notice here.

1.2 Home

This is a conversion/edition of Martin "nocash" Korth's Playstation specs document originally hosted at https://problemkaputt.de/psx-spx.htm. See https://github.com/psx-spx/psx-spx.github.io#readme for more details.
You can also download this website as a single-page pdf.

To discuss the contents of this document, or hang out with likely minded people on development, hacking, and reverse engineering of Sony's first console, feel free to join the PSX.Dev Discord Server.

Memory Map
I/O Map
Graphics Processing Unit (GPU)
Geometry Transformation Engine (GTE)
Macroblock Decoder (MDEC)
Sound Processing Unit (SPU)
Interrupts
DMA Channels
Timers
CDROM Drive
Controllers and Memory Cards
Pocketstation
Serial Port (SIO)

2. Memory Map

Memory Map

Additionally, there are a number of memory mirrors.

Additional Memory (not mapped to the CPU bus)

KUSEG,KSEG0,KSEG1,KSEG2 Memory Regions

Kernel Memory: KSEG1 is the normal physical memory (uncached), KSEG0 is a mirror thereof (but with cache enabled). KSEG2 is usually intended to contain virtual kernel memory, but in the PSX it's containing Cache Control hardware registers.
User Memory: KUSEG is intended to contain 2GB virtual memory (on extended MIPS processors), the PSX doesn't support virtual memory, and KUSEG simply contains a mirror of KSEG0/KSEG1 (in the first 512MB) (trying to access memory in the remaining 1.5GB causes an exception).

i-Cache

The i-Cache can hold 4096 bytes, or 1024 instructions.
It is only active in the cached regions (KUSEG and KSEG0).
There are reportedly some restrictions... not sure there... eventually it is using the LSBs of the address as cache-line number... so, for example, it couldn't simultaneously memorize opcodes at BOTH address 80001234h, AND at address 800F1234h (?)

Scratchpad

MIPS CPUs usually have a d-Cache, but, in the PSX, Sony has assigned it as what's referenced as the "Scratchpad", mapped to a fixed memory location at 1F800000h.. 1F8003FFh, ie. it's used as Fast RAM, rather than as cache.
There \<might> be a way to disable that behavior (via Port FFFE0130h or so), but, the Kernel is accessing I/O ports via KUSEG, so activating Data Cache would cause the Kernel to access cached I/O ports.
The purpose of the scratchpad is to have a more flexible cache system available to the programmer. Neither the kernel nor the Sony libraries will try to make use of it, so it is therefore completely up for grabs to the programmer. A good example would be if you were to write a piece of code that's doing a lot of CRC computation, to use the 1KB scratchpad to initially load the CRC lookup tables, which incidentally, is exactly 1KB large. Doing this will relieve SDRAM page changes overhead while reading the data to checksum linearly, while also keeping the whole CRC code in the i-Cache, hence being more optimal than what you'd get with an automatic d-Cache system.

Memory Mirrors

As described above, the 512Mbyte KUSEG, KSEG0, and KSEG1 regions are mirrors of each other. Additional mirrors within these 512MB regions are:

The size of the RAM, BIOS, Expansion regions can be configured by software, for Expansion Region it's also possible to change base address, see:
Memory Control
The Scratchpad is mirrored only in KUSEG and KSEG0, but not in KSEG1.

Memory Exceptions

Write queue

The MIPS CPU has a 4-words deep pass-through write queue, in order to relieve some bus contention when writing to memory. If reading the same memory location that just got written into the write queue, it will first be flushed before being read back from memory.
It is important to realize that the write queue's mechanism is only viable for normal memory attached to the main CPU, and that any hardware register state machine will get messed up by it.
The typical example is the typical JEDEC standard to access flash, which usually does the following sequence to read the ID of a flash chip:

In this example above, if is located in a memory segment that has the write queue enabled, even if the low level assembly code will do the first 3 stores before doing 2 loads, the physical signals sent to that device through the CPU bus will be seen in the sequence:

Therefore, using KSEG1 that disables the write queue is the only way to ensure that the operations are done in the proper way.

The above is valid for most of the hardware connected to the main CPU, such as the CDROM controller, exp1, exp2, the SPU, or the GPU. Therefore, using BF80180xh to access the CDROM registers is more correct than using 1F80180xh.

It is noteworthy that the Sony code will still incorrectly use KUSEG as the memory map for all hardware registers, and they then spend a lot of time writing 4 dummy values somewhere, in order to ensure the write queue has been flushed.

The SN debugger in contrast is properly using the KSEG1 memory map for all the hardware registers, nullifying the need to flush the write queue when accessing it.

It's also noteworthy that doing ANY KSEG1 access (read OR write) will automatically stall the CPU in order to flush the whole write queue before proceeding with the operation. Therefore, all BIOS ROM operations will naturally and effectively have the write queue disabled, as this code requires the CPU to read from KSEG1 constantly.

This also means that if using KUSEG for the hardware registers, another method to flush the write queue, albeit potentially slightly less efficient, would be to simply read the first byte located at BFC00000h. The latter is what is effectively described as the official method to flush the write queue in the MIPS handbook. This could be potentially useful to flush the write queue all at once, instead of flushing it word by word.

More Memory Info

For Info on Exception vectors, Unused/Garbage memory locations, I/O Ports, Expansion ROM Headers, and Memory Waitstate Control, etc. see:
I/O Map
Memory Control
EXP1 Expansion ROM Header
BIOS Memory Map
BIOS Memory Allocation
COP0 - Exception Handling
Unpredictable Things

3. I/O Map

Expansion Region 1

Scratchpad

Memory Control 1

Peripheral I/O Ports

Memory Control 2

Interrupt Control

DMA Registers

Timers (aka Root counters)

CDROM Registers (Address.Read/Write.Index)

GPU Registers

MDEC Registers

SPU Voice 0..23 Registers

SPU Control Registers

SPU Reverb Configuration Area

SPU Internal Registers

Expansion Region 2 (default 128 bytes, max 8 KBytes)

Expansion Region 2 - Dual Serial Port (for TTY Debug Terminal)

Expansion Region 2 - Int/Dip/Post

Expansion Region 2 - Nocash Emulation Expansion

Expansion Region 3 (default 1 byte, max 2 MBytes)

BIOS Region (default 512 Kbytes, max 4 MBytes)

Memory Control 3 (Cache Control)

Coprocessor Registers

4. Graphics Processing Unit (GPU)

The GPU can render Polygons, Lines, or Rectangles to the Drawing Buffer, and sends the Display Buffer to the Television Set. Polygons are useful for 3D graphics (or rotated/ scaled 2D graphics), Rectangles are useful for 2D graphics and Text output.

4.1 GPU I/O Ports, DMA Channels, Commands, VRAM

GPU I/O Ports (1F801810h and 1F801814h in Read/Write Directions)

It (=GP0 only?) has a 64-byte (16-word) command FIFO buffer.
Optionally, Port 1F801810h (Read/Write) can be also accessed via DMA2.
The communication between the CPU and the GPU is a 32-bits data-only bus called the VBUS. Aside from address line 2 being connected, in order to make the difference

between port 0 and 1, there are no other address line between the two chips.
Thus the GPU can be seen as a blackbox that executes 32 bits commands.

GPU Timers / Synchronization

Most of the Timers are bound to GPU timings, see
Timers
Interrupts

GPU-related DMA Channels (DMA2 and DMA6)

Note: Before using DMA2, set up the DMA Direction in GP1(04h).
DMA2 is equivalent to accessing Port 1F801810h (GP0/GPUREAD) by software.
DMA6 just initializes data in Main RAM (not physically connected to the GPU).

GPU Command Summary

While it is probably more simple for the MIPS software to see GPU commands as a collection of bytes, the GPU will only see 32 bits words being sent to it. Therefore, while the Sony libraries will fill up structures to send to the GPU using byte-level granularity, it is much more simple to see these as bitmasks from the GPU's point of view.
So when processing commands on GP0, the GPU will first inspect the top 3 bits of the 32 bits command being sent. Depending on the value of these 3 bits, further decoding of the other bits can be done.
Commands sent to GP1 are more simple in nature to decode.

Top 3 bits of a GP0 command:

Some GP0 commands require additional parameters, which are written (following the initial command) as further 32bit values to GP0. The execution of the command starts when all parameters have been received (or, in case of Polygon/Line commands, when the first 3/2 vertices have been received).

The astute reader will realize that there are shared bits between primitives, such as the gouraud shading flag.

Unlike all the others, the environment commands are more clear to be seen as a single 8 bits command, therefore the rest of the document will refer to them by their full 8 bits value.

Clear Cache

The GPU has a small texture cache, in order to reduce VRAM access. This command flushes it, when mutating the VRAM, similar to how the CPU i-cache must be flushed after writing new code and before executing it.
Note that it is possible to abuse the texture cache by changing pixels in VRAM that the GPU loaded in its cache, therefore creating weird drawing effects, but this is only seen in some demos, and never in actual games.

Quick Rectangle Fill

Fills the area in the frame buffer with the value in RGB. Horizontally the filling is done in 16-pixel (32-bytes) units (see below masking/rounding).
The "Color" parameter is a 24bit RGB value, however, the actual fill data is 16bit: The hardware linearly converts the 24bit RGB value to 15bit RGB by dropping the lower 3 bits of each color value and additionally sets the mask bit (bit15) to 0.
Rectangle filling is not affected by the GP0(E6h) mask setting, acting as if GP0(E6h).0 and GP0(E6h).1 are both zero.
This command is typically used to do a quick clear, as it'll be faster to run than an equivalent Render Rectangle command.

VRAM Overview / VRAM Addressing

VRAM is 1MByte (not mapped to the CPU bus) (it can be read/written only via I/O or DMA). The memory is used for:

The 1MByte VRAM is organized as 512 lines of 2048 bytes. It is accessed via coordinates, ranging from (0,0)=Upper-Left to (N,511)=Lower-Right.

The horizontal coordinates are addressing memory in 4bit/8bit/16bit/24bit/halfword units (depending on what data formats you are using) (or a mixup thereof, eg. a halfword-base address, plus a 4bit texture coordinate).

4.2 GPU Render Polygon Commands

When the upper 3 bits of the first GP0 command are set to 1 (001), then the command can be decoded using the following bitfield:

Subsequent data sent to GP0 to complete this command will be the vertex data for the command. The meaning and count of these words will be altered by the initial flags sent in the first command.

If doing flat rendering, no further color will be sent. If doing gouraud shading, there will be one more color per vertex sent, and the initial color will be the one for vertex 0.

If doing 3 or 4 vertices rendering, this will affect the total number of vertex sent.

If doing textured rendering, each vertex sent will also have a U/V texture coordinate attached to it, as well as a CLUT index.

So each vertex data can be seen as the following set of words:

The upper 16 bits of the first two UV words contain extra information. The first word holds the Clut index. The second word contains texture page information. Any further clut/page bits should be set to 0.

So for example, a solid flat blue triangle of coordinate (10, 20), (30, 40), (50, 60) will be drawn using the following draw call data:

And a quad with gouraud shading texture-blend will have the following structure:

Some combination of these flags can be seen as nonsense however, but it's important to realize that the GPU will still process them properly. For instance, specifying gouraud shading without texture blending will force the user to send the colors for each vertex to satisfy the GPU's state machine, without them being actually used for the rendering.

Notes

Polygons are displayed up to \<excluding> their lower-right coordinates.
Four-point polygons are internally processed as two Three-point polygons, the first

consisting of Vertices 1,2,3, and the second of Vertices 2,3,4.
Within the Three-point polygons, the ordering of the vertices is don't care at the GPU side (a front-back check, based on clockwise or anti-clockwise ordering, can be implemented at the GTE side).
Dither enable (in Texpage command) affects ONLY polygons that do use Gouraud Shading or Texture Blending.

4.3 GPU Render Line Commands

When the upper 3 bits of the first GP0 command are set to 2 (010), then the command can be decoded using the following bitfield:

So each vertex can be seen as the following list of words:

When polyline mode is active, at least two vertices must be sent to the GPU. The vertex list is terminated by the bits 12-15 and 28-31 equaling , or . The terminator value occurs on the first word of the vertex (i.e. the color word if it's a gouraud shaded).

Note

Lines are displayed up to \<including> their lower-right coordinates (ie. unlike as for polygons, the lower-right coordinate is not excluded).
If dithering is enabled (via Texpage command), then both monochrome and shaded lines are drawn with dithering (this differs from monochrome polygons and monochrome rectangles).

Wire-Frame

Poly-Lines can be used (among others) to create Wire-Frame polygons (by setting the last Vertex equal to Vertex 1).

4.4 GPU Render Rectangle Commands

Rectangles are drawn much faster than polygons. Unlike polygons, gouraud shading is not possible, dithering isn't applied, the rectangle must forcefully have horizontal and vertical edges, textures cannot be rotated or scaled, and, of course, the GPU does render Rectangles as a single entity, without splitting them into two triangles.

The Rectangle command can be decoded using the following bitfield:

The parameter can be seen as the following enum:

Therefore, the whole draw call can be seen as the following sequence of words:

Unlike for Textured-Polygons, the "Texpage" must be set up separately for Rectangles, via GP0(E1h). Width and Height can be up to 1023x511, however, the maximum size of the texture window is 256x256 (so the source data will be repeated when trying to use sizes larger than 256x256).

Texture Origin and X/Y-Flip

Vertex & Texcoord specify the upper-left edge of the rectangle. And, normally, screen coords and texture coords are both incremented during rendering the rectangle pixels.
Optionally, X/Y-Flip bits can be set in Texpage.Bit12/13, these bits cause the texture coordinates to be decremented (instead of incremented). The X/Y-Flip bits do affect only Rectangles (not Polygons, nor VRAM Transfers).
Caution: Reportedly, the X/Y-Flip feature isn't supported on old PSX consoles (unknown which ones exactly, maybe such with PU-7 mainboards, and unknown how to detect flipping support; except of course by reading VRAM).

Note

There are also two VRAM Transfer commands which work similar to GP0(60h) and GP0(65h). Eventually, that commands might be even faster... although not sure if they do use the Texture Cache?
The difference is that VRAM Transfers do not clip to the Drawig Area boundary, do not support fully-transparent nor semi-transparent texture pixels, and do not convert color depths (eg. without 4bit texture to 16bit framebuffer conversion).

4.5 GPU Rendering Attributes

Vertex (Parameter for Polygon, Line, Rectangle commands)

Size Restriction: The maximum distance between two vertices is 1023 horizontally, and 511 vertically. Polygons and lines that are exceeding that dimensions are NOT rendered. For example, a line from Y1=-300 to Y2=+300 is NOT rendered, a line from Y1=-100 to Y2=+400 is rendered (as far as it is within the drawing area).
If portions of the polygon/line/rectangle are located outside of the drawing area, then the hardware renders only the portion that is inside of the drawing area. Not sure if the hardware is skipping all clipped pixels at once (within a single clock cycle), or if it's (slowly) processing them pixel by pixel?

Color Attribute (Parameter for all Rendering commands, except Raw Texture)

Caution: For untextured graphics, 8bit RGB values of FFh are brightest. However, for texture blending, 8bit values of 80h are brightest (values 81h..FFh are "brighter than bright" allowing to make textures about twice as bright as than they were originially stored in memory; of course the results can't exceed the maximum brightness, ie. the 5bit values written to the framebuffer are saturated to max 1Fh).

Texpage Attribute (Parameter for Textured-Polygons commands)

This attribute is used in all Textured-Polygons commands.

Clut Attribute (Color Lookup Table, aka Palette)

This attribute is used in all Textured Polygon/Rectangle commands. Of course, it's relevant only for 4bit/8bit textures (don't care for 15bit textures).

Specifies the location of the CLUT data within VRAM.

GP0(E1h) - Draw Mode setting (aka "Texpage")

The GP0(E1h) command is required only for Lines, Rectangle, and Untextured-Polygons (for Textured-Polygons, the data is specified in form of the Texpage attribute; except that, Bit9-10 can be changed only via GP0(E1h), not via the Texpage attribute).
Texture page colors setting 3 (reserved) is same as setting 2 (15bit).
Note: GP0(00h) seems to be often inserted between Texpage and Rectangle commands, maybe it acts as a NOP, which may be required between that commands, for timing reasons...?

GP0(E2h) - Texture Window setting

Mask specifies the bits that are to be manipulated, and Offset contains the new values for these bits, ie. texture X/Y coordinates are adjusted as so:

The area within a texture window is repeated throughout the texture page. The data is not actually stored all over the texture page but the GPU reads the repeated patterns as if they were there.

GP0(E3h) - Set Drawing Area top left (X1,Y1)

GP0(E4h) - Set Drawing Area bottom right (X2,Y2)

Sets the drawing area corners. The Render commands GP0(20h..7Fh) are automatically clipping any pixels that are outside of this region.

GP0(E5h) - Set Drawing Offset (X,Y)

If you have configured the GTE to produce vertices with coordinate "0,0" being located in the center of the drawing area, then the Drawing Offset must be "X1+(X2-X1)/2, Y1+(Y2-Y1)/2". Or, if coordinate "0,0" shall be the upper-left of the Drawing Area, then Drawing Offset should be "X1,Y1". Where X1,Y1,X2,Y2 are the values defined with GP0(E3h-E4h).

GP0(E6h) - Mask Bit Setting

When bit0 is off, the upper bit of the data written to the framebuffer is equal to bit15 of the texture color (ie. it is set for colors that are marked as "semi-transparent") (for untextured polygons, bit15 is set to zero).
When bit1 is on, any (old) pixels in the framebuffer with bit15=1 are write-protected, and cannot be overwritten by (new) rendering commands.
The mask setting affects all rendering commands, as well as CPU-to-VRAM and VRAM-to-VRAM transfer commands (where it acts on the separate halfwords, ie. as for 15bit textures). However, Mask does NOT affect the Fill-VRAM command.
This setting is used in games such as Metal Gear Solid and Silent Hill.

Note

GP0(E3h..E5h) do not take up space in the FIFO, so they are probably executed immediately (even if there're still other commands in the FIFO). Best use them only if you are sure that the FIFO is empty (otherwise the new Drawing Area settings might accidentally affect older Rendering Commands in the FIFO).

4.6 GPU Memory Transfer Commands

The next three commands being described are when the high 3 bits are set to the values 4 (100), 5 (101), and 6 (110). For them, the remaining 29 bits are ignored, and can be set to any arbitrary value.

VRAM to VRAM blitting - command 4 (100)

Copies data within framebuffer. The transfer is affected by Mask setting.

CPU to VRAM blitting - command 5 (101)

Transfers data from CPU to frame buffer. If the number of halfwords to be sent is odd, an extra halfword should be sent, as packets consist of 32bits words. The transfer is affected by Mask setting.

VRAM to CPU blitting - command 6 (110)

Transfers data from frame buffer to CPU. Wait for bit27 of the status register to be set before reading the image data. When the number of halfwords is odd, an extra halfword is added at the end, as packets consist of 32bits words.

Masking and Rounding for FILL Command parameters

Fill does NOT occur when Xsiz=0 or Ysiz=0 (unlike as for Copy commands). Xsiz=400h works only indirectly: Param=400h is handled as Xsiz=0, however, Param=3F1h..3FFh is rounded-up and handled as Xsiz=400h.

Note that because of the height (Ysiz) masking, a maximum of 511 rows can be filled in a single command. Calling a fill with a full VRAM height of 512 rows will be ineffective as the height will be masked to 0.

Masking for COPY Commands parameters

Parameters are just clipped to 10bit/9bit range, the only special case is that Size=0 is handled as Size=max.

Notes

The coordinates for the above VRAM transfer commands are absolute framebuffer addresses (not relative to Draw Offset, and not clipped to Draw Area).
Non-DMA transfers seem to be working at any time, but GPU-DMA Transfers seem to be working ONLY during V-Blank (outside of V-Blank, portions of the data appear to be skipped, and the following words arrive at wrong addresses), unknown if it's possible to change that by whatever configuration settings...? That problem appears ONLY for continous DMA aka VRAM transfers (linked-list DMA aka Ordering Table works even outside V-Blank).

Wrapping

If the Source/Dest starting points plus the width/height value exceed the 1024x512 pixel VRAM size, then the Copy/Fill operations wrap to the opposite memory edge (without any carry-out from X to Y, nor from Y to X).

4.7 GPU Other Commands

GP0(1Fh) - Interrupt Request (IRQ1)

Requests IRQ1. Can be acknowledged via GP1(02h). This feature is rarely used.
Note: The command is used by Blaze'n'Blade, but the game doesn't have IRQ1 enabled, and the written value (1F801810h) looks more like an I/O address, rather than like a command, so not sure if it's done intentionally, or if it is just a bug.

GP0(03h) - Unknown?

Unknown. Doesn't seem to be used by any games. Unlike the "NOP" commands, GP0(03h) does take up space in FIFO, so it is apparently not a NOP.

GP0(00h) - NOP (?)

This command doesn't take up space in the FIFO (eg. even if a VRAM-to-VRAM transfer is still busy, one can send dozens of GP0(00h) commands, without the command FIFO becoming full. So, either the command is ignored (or, if it has a function, it is executed immediately, even while the transfer is busy).
...
GP0(00h) unknown, used with parameter = 08A16Ch... or rather 08FDBCh ... the written value seems to be a bios/ram memory address, anded with 00FFFFFFh... maybe a bios bug?
GP0(00h) seems to be often inserted between Texpage and Rectangle commands, maybe it acts as a NOP, which may be required between that commands, for timing reasons...?

GP0(04h..1Eh,E0h,E7h..EFh) - Mirrors of GP0(00h) - NOP (?)

Like GP0(00h), these commands don't take up space in the FIFO. So, maybe, they are same as GP0(00h), however, the Drawing Area/Offset commands GP0(E3h..E5h) don't take up FIFO space either, so not taking up FIFO space doesn't neccessarily mean that the command has no function.

4.8 GPU Display Control Commands (GP1)

GP1 Display Control Commands are sent by writing the 8bit Command number (MSBs), and 24bit parameter (LSBs) to Port 1F801814h. Unlike GP0 commands, GP1 commands are passed directly to the GPU (ie. they can be sent even when the FIFO is full).

GP1(00h) - Reset GPU

Resets the GPU to the following values:

Accordingly, GPUSTAT becomes 14802000h. The x1,y1 values are too small, ie. the upper-left edge isn't visible. Note that GP1(09h) is NOT affected by the reset command.

GP1(01h) - Reset Command Buffer

Resets the command buffer and CLUT cache.

GP1(02h) - Acknowledge GPU Interrupt (IRQ1)

Resets the IRQ flag in GPUSTAT.24. The flag can be set via GP0(1Fh).

GP1(03h) - Display Enable

Turns display on/off. "Note that a turned off screen still gives the flicker of NTSC on a PAL screen if NTSC mode is selected."
The "Off" settings displays a black picture (and still sends /SYNC signals to the television set). (Unknown if it still generates vblank IRQs though?)

GP1(04h) - DMA Direction / Data Request

Notes: Manually sending/reading data by software (non-DMA) is ALWAYS possible, regardless of the GP1(04h) setting. The GP1(04h) setting does affect the meaning of GPUSTAT.25.

Display start/end

Specifies where the display area is positioned on the screen, and how much data gets sent to the screen. The screen sizes of the display area are valid only if the horizontal/vertical start/end values are default. By changing these you can get bigger/smaller display screens. On most TV's there is some black around the edge, which can be utilised by setting the start of the screen earlier and the end later. The size of the pixels is NOT changed with these settings, the GPU simply sends more data to the screen. Some monitors/TVs have a smaller display area and the extended size might not be visible on those sets. "(Mine is capable of about 330 pixels horizontal, and 272 vertical in 320*240 mode)"

GP1(05h) - Start of Display area (in VRAM)

Upper/left Display source address in VRAM. The size and target position on screen is set via Display Range registers; target=X1,Y2; size=(X2-X1/cycles_per_pix), (Y2-Y1).

GP1(06h) - Horizontal Display range (on Screen)

Specifies the horizontal range within which the display area is displayed. For resolutions other than 320 pixels it may be necessary to fine adjust the value to obtain an exact match (eg. X2=X1+pixels*cycles_per_pix).
The number of displayed pixels per line is "(((X2-X1)/cycles_per_pix)+2) AND NOT 3" (ie. the hardware is rounding the width up/down to a multiple of 4 pixels).
Most games are using a width equal to the horizontal resolution (ie. 256, 320, 368, 512, 640 pixels). A few games are using slightly smaller widths (probably due to programming bugs). Pandemonium 2 is using a bigger "overscan" width (ensuring an intact picture without borders even on mis-calibrated TV sets).
The 260h value is the first visible pixel on normal TV Sets, this value is used by MOST NTSC games, and SOME PAL games (see below notes on Mis-Centered PAL games).
Video clock unit used depends on console region, regardless of NTSC/PAL video mode set by GP1(08h).3; see section on nominal video clocks for values.

GP1(07h) - Vertical Display range (on Screen)

Specifies the vertical range within which the display area is displayed. The number of lines is Y2-Y1 (unlike as for the width, there's no rounding applied to the height). If Y2 is set to a much too large value, then the hardware stops to generate vblank interrupts (IRQ0).
The 88h/A3h values are the middle-scanlines on normal TV Sets, these values are used by MOST NTSC games, and SOME PAL games (see below notes on Mis-Centered PAL games).
The 240/288 values are for fullscreen pictures. Many NTSC games display 240 lines, but on most analog television sets, only 224 lines are visible (8 lines of overscan on top and 8 lines of overscan on bottom). Many PAL games display only 256 lines (underscan with black borders).
Some games such as Chrono Cross will occasionally adjust these values to create a screen shake effect, so proper emulation of this command is necessary for those particular cases.

GP1(08h) - Display mode

Note: Interlace must be enabled to see all lines in 480-lines mode (interlace causes ugly flickering, so a non-interlaced low resolution image typically has better quality than a high resolution interlaced image, a pretty bad example is the intro screens shown by the BIOS). The Display Area Color Depth bit does NOT affect GP0 draw commands, which always draw in 15 bit. However, the Vertical Interlace flag DOES affect GP0 draw commands.
When the "Reverseflag" is set, the display scrolls down 2 lines or so, and colored regions are getting somehow hatched/distorted, but black and white regions are still looking okay. Don't know what that's good for? Probably relates to PAL/NTSC-Color Clock vs PSX-Dot Clock mismatches: Bit7=0 causes Flimmering errors (errors at different locations in each frame), and Bit7=1 causes Static errors (errors at same locations in all frames)?

GP1(10h) - Get GPU Info

GP1(11h..1Fh) - Mirrors of GP1(10h), Get GPU Info

After sending the command, the result can be read (immediately) from GPUREAD register (there's no NOP or other delay required) (namely GPUSTAT.Bit27 is used only for VRAM-Reads, but NOT for GPU-Info-Reads, so do not try to wait for that flag).

On Old 180pin GPUs, following values can be selected:

On New 208pin GPUs, following values can be selected:

The selected data is latched in GPUREAD, the same/latched value can be read multiple times, but, the latch isn't automatically updated when changing GP0 registers.

GP1(09h) - New Texture Disable

This feature seems to be intended for debugging purposes (most released games do contain program code for disabling textures, but do never execute it).
GP1(09h) seems to be supported only on New GPUs. Old GPUs don't support it all, and there seem to be some Special/Prototype GPUs that use GP1(20h) instead of GP1(09h).

GP1(20h) - Special/Prototype Texture Disable

Seems to be a used only on whatever arcade/prototype GPUs. New GPUs are using GP1(09h) instead of GP1(20h).

GP1(0Bh) - Unknown/Internal?

The register doesn't seem to be used by any games.

GP1(0Ah,0Ch..0Fh,21h..3Fh) - N/A

Not used?

GP1(40h..FFh) - N/A (Mirrors)

Mirrors of GP1(00h..3Fh).

Mis-Centered PAL Games (wrong GP1(06h)/GP1(07h) settings)

NTSC games are typically well centered (using X1=260h, and Y1/Y2=88h+/-N).
PAL games should be centered as X1=260h, and Y1/Y2=A3h+/-N) - these values would be looking well on a Philips Philetta TV Set, and do also match up with other common picture positions (eg. as used by Nintendo's SNES console).
However, most PAL games are using completely different "random" centering values (maybe caused by different developers trying to match the centering to the different TV Sets) (although it looks more as if the PAL developers just went amok: Many PAL games are even using different centerings for their Intro, Movie, and actual Game sequences).
In result, most PAL games are looking like crap when playing them on a real PSX. For PSX emulators it may be recommended to ignore the GP1(06h)/GP1(07h) centering, and instead, apply auto-centering to PAL games.
For PAL game developers, it may be recommended to add a screen centering option (as found in Tomb Raider 3, for example). Unknown if this is really required... or if X1=260h, and Y1/Y2=A3h+/-N would work fine on most or all PAL TV Sets?

4.9 GPU Status Register

1F801814h - GPUSTAT - GPU Status Register (R)

In 480-lines mode, bit31 changes per frame. And in 240-lines mode, the bit changes per scanline. The bit is always zero during Vblank (vertical retrace and upper/lower screen border).

Note

Further GPU status information can be retrieved via GP1(10h) and GP0(C0h).

Ready Bits

Bit28: Normally, this bit gets cleared when the command execution is busy (ie. once when the command and all of its parameters are received), however, for Polygon and Line Rendering commands, the bit gets cleared immediately after receiving the command word (ie. before receiving the vertex parameters). The bit is used as DMA request in DMA Mode 2, accordingly, the DMA would probably hang if the Polygon/Line parameters are transferred in a separate DMA block (ie. the DMA probably starts ONLY on command words).
Bit27: Gets set after sending GP0(C0h) and its parameters, and stays set until all data words are received; used as DMA request in DMA Mode 3.
Bit26: Gets set when the GPU wants to receive a command. If the bit is cleared, then the GPU does either want to receive data, or it is busy with a command execution (and doesn't want to receive anything).
Bit25: This is the DMA Request bit, however, the bit is also useful for non-DMA transfers, especially in the FIFO State mode.

4.10 GPU Versions

Summary of GPU Differences

Shaded Textures

The Old GPU crops 8:8:8 bit gouraud shading color to 5:5:5 bit before multiplying it with the texture color, resulting in rather poor graphics. For example, the snow scence in the first level of Tomb Raider I looks a lot smoother on New GPUs.
The cropped colors are looking a bit as if dithering would be disabled (although, technically dithering works fine, but due to the crippled color input, it's always using the same dither pattern per 8 intensities, instead of using 8 different dither patterns).

Memory/Rendering Timings

The Old GPU uses two Dual-ported VRAM chips (each with two 16bit databusses, one for CPU/DMA/rendering access, and one for output to the video DAC). The New GPU uses s normal DRAM chip (with single 32bit databus).
The exact timing differences are unknown, but the different memory types should result in quite different timings:
The Old GPU might perform better on non-32bit aligned accesses, and on memory accesses performed simultaneously with DAC output.
On the other hand, the New GPU's DRAM seems to be faster in some cases (for example, during Vblank, it's fast enough to perform DMA's with blksiz>10h, which exceeds the GPU's FIFO size, and causes lost data on Old GPUs).

X/Y-Flip and 2MB Video RAM

The X/Y-flipping feature may be used by arcade games (provided that the arcade board is fitted with New GPUs). The flipping feature does also work on retail consoles with New GPUs, but PSX games should never use that feature (for maintaining compatiblity with older PSX consoles).

2Mbyte Video RAM is used on some arcade boards. Whilst PSX retail consoles are always containing only 1MByte RAM, so the feature cannot be used even if the console contains a New GPU. There's one special case: Some PSone consoles are actually fitted with 2MB chips (maybe because smaller chips haven't been in production anymore), but the chips are wired so that only half of the memory is accessible (the extra memory could be theoretically unlocked with some minimal hardware modification).

GPU Detection (and optional texture disable)

Below is slightly customized GPU Detection function taken from Perfect Assassin (the index7 latching works ONLY on New GPUs, whilst old GPUs would leave the latched value unchanged; as a workaround, the index4 latching is used to ensure that the latch won't contain 000002h on old GPUs, assuming that index4 is never set to 000002h).

GP0(02h) FillVram

The FillVram command does normally ignore the lower 4bit of the x-coordinate (and software should always set those bits to zero). However, if the 4bits are all set, then the Old GPU does write each 2nd pixel to wrong memory address. For example, a 32x4 pixel fill produces following results for x=0..1Fh:

Arcade GPUs

Some arcade boards are using normal retail GPUs, however, there are also two special non-retail 208pin GPUs which seem to be solely used on arcade boards:

The exact differences to retail GPUs are unknown. One of the special GPUs is said to use entierly different command numbers for rendering commands (maybe some old prototype variant, or maybe some protection against cloning arcade boards with retail chips).

4.11 GPU Depth Ordering

Absent Depth Buffer

The PlayStation's GPU stores only RGB colors in the framebuffer (ie. unlike modern 3D processors, it's NOT buffering Depth values; leaving apart the Mask bit, which could be considered as a tiny 1bit "Depth" or "Priority" value). In fact, the GPU supports only X,Y coordinates, and it's totally unaware of Z coordinates. So, when rendering a polygon, the hardware CANNOT determine which of the new pixels are in front/behind of the old pixels in the buffer.

Simple Ordering

The rendering simply takes place in the ordering as the data is sent to the GPU (ie. the most distant objects should be sent first). For 2D graphics, it's fairly easy follow that order (eg. even multi-layer 2D graphics can be using DMA2-continous mode).

Depth Ordering Table (OT)

For 3D graphics, the ordering of the polygons may change more or less randomly (eg. when rotating/moving the camera). To solve that problem, the whole rendering data is usually first stored in a Depth Ordering Table (OT) in Main RAM, and, once when all polygons have been stored in the OT, the OT is sent to the GPU via "DMA2-linked-list" mode.

Initializing an empty OT (via DMA6)

DMA channel 6 can be used to set up an empty linked list, in which each entry points to the previous:

Each entry has a size of 00h words (upper 8bit), and a pointer to the previous entry (lower 24bit). With the above Example values, the generated table would look like so:

Inserting Entries (Passing GTE data to the OT) (by software)

The GTE commands AVSZ3 and AVSZ4 can be used to calculate the Average Z coordinates of a polygon (based on its three or four Z coordinates). The result is returned as a 16bit Z value in GTE register OTZ, the commands do also allow to divide the result, to make it less than 16bit (the full 16bit would require an OT of 256KBytes - for the EMPTY table, which would be a waste of memory, and which would slowdown the DMA2/DMA6 operations) (on the other hand, a smaller table means less depth resolution).

If there's been already an entry (at the same OTZ index), then the new polygon will be processed first (ie. it will appear "behind" of the old entry).

Not sure if the packet size must be limited to max N=16 words (ie. as for the DMA2-continous block size) (due to GP0 FIFO size limits)?

Sending the OT to the CPU (via DMA2-linked-list mode)

4.12 GPU Video Memory (VRAM)

Framebuffer

The framebuffer contains the image that is to be output to the Television Set. The GPU supports 10 resolutions, with 16bit or 24bit per pixel.

Note: In most cases, you'll need TWO framebuffers (one being displayed, and used as rendering target) (unless you are able to draw the whole new image during vblank, or unless when using single-layer 2D graphics). So, resolutions that occupy more than 512K would exceed the available 1MB VRAM when using 2 buffers. Also, high resolutions mean higher rendering load, and less texture memory.

Note: The 24bit pixels occupy 3 bytes (not 4 bytes with unused MSBs), so each 6 bytes contain two 24bit pixels. The 24bit display mode works only with VRAM transfer commands like GP0(A0h); the rendering commands GP0(20h..7Fh) cannot output 24bit data. Ie. 24bit mode is used mostly for MDEC videos (and some 2D games like Heart of Darkness).

Texture Bitmaps

A texture is an image put on a polygon or sprite. The data of a texture can be stored in 3 different modes:

A Texture Page is a 256x256 texel region in VRAM (the Polygon rendering commands are using Texcoords with 8bit X,Y coordinates, so polygons cannot use textures bigger than 256x256) (the Rectangle rendering commands with width/height parameters could theoretically use larger textures, but the hardware clips their texture coordinates to 8bit, too).
The GP0(E2h) Texture Window (aka Texture Repeat) command can be used to reduce the texture size to less than 256x256 texels.
The Texture Pages can be located in the frame buffer on X multiples of 64 halfwords and Y multiples of 256 lines.

Texture Palettes - CLUT (Color Lookup Table)

The clut is a the table where the colors are stored for the image data in the CLUT modes. The pixels of those images are used as indexes to this table. The clut is arranged in the frame buffer as a 256x1 image for the 8bit clut mode, and a 16x1 image for the 4bit clut mode.

The clut data can be arranged in the frame buffer at X multiples of 16 (X=0,16,32,48,etc) and anywhere in the Y range of 0-511.

Texture Color Black Limitations

On the PSX, texture color 0000h is fully-transparent, that means textures cannot contain Black pixels. However, in some cases, Color 8000h (Black with semi-transparent flag) can be used, depending on the rendering command:

So, with semi-transparent rendering commands, it isn't possible to use Non-Transparent Black pixels in textures, the only workaround is to use colors like 0001h (dark red) or 0400h (dark blue). However, due to the PSX's rather steeply increasing intensity ramp, these colors are clearly visible to be brighter than black.

RGB Intensity Notes

The Playstations RGB values aren't linear to normal RGB values (as used on PCs). The min/max values are of course the same, but the medium values differ:

Ie. on the PSX, the intensity increases steeply from 0 to 15, and less steeply from 16 to 31.

4.13 GPU Texture Caching

The GPU has 2 Kbyte Texture Cache
There is also a CLUT cache that is preserved between GPU drawing commands. The CLUT cache is invalidated when different CLUT index values are used or when GP0(01h) is issued. It is unknown if the CLUT cache overlaps or is shared with the Texture Cache.

If polygons with texture are displayed, the GPU needs to read these from the frame buffer. This slows down the drawing process, and as a result the number of polygons that can be drawn in a given timespan. To speed up this process the GPU is equipped with a texture cache, so a given piece of texture needs not to be read multiple times in succession.
The texture cache size depends on the color mode used for the textures.
In 4 bit CLUT mode it has a size of 64x64, in 8 bit CLUT it's 32x64 and in 15bitDirect is 32x32. A general speed up can be achieved by setting up textures according to these sizes. For further speed gain a more precise knowledge of how the cache works is necessary.

Cache blocks

The texture page is divided into non-overlapping cache blocks, each of a unit size according to color mode. These cache blocks are tiled within the texture page.

Cache entries

Each cache block is divided into 256 cache entries, which are numbered sequentially, and are 8 bytes wide. So a cache entry holds 16 4bit clut pixels 8 8bit clut pixels, or 4 15bitdirect pixels.

The cache can hold only one cache entry by the same number, so if f.e. a piece of texture spans multiple cache blocks and it has data on entry 9 of block 1, but also on entry 9 of block 2, these cannot be in the cache at once.

4.14 GPU Timings

Nominal Video Clock

Consoles will always use the video clock for its region, regardless of the GPU being configured in NTSC or PAL output mode, because an NTSC console lacks a PAL reference clock and vice versa. Without modifications for an additional oscillator for the other region, consoles may experience drift over time when playing content from a different video region. See vertical refresh rates below.

Vertical Video Timings

Horizontal blanking and vertical blanking signals occur on the video output side as expected for NTSC/PAL signals. These are not necessarily the same as the timer/ interrupt HBLANK and VBLANK.

Vertical Refresh Rates

For emulation purposes, it's recommended to use an NTSC video clock when running NTSC content (or in NTSC mode) and a PAL clock when running PAL content (or in PAL mode).

TODO: Derivations for vertical refresh rates; horizontal timing notes

Nocash's original GPU Timings notes:

Video Clock

The PSone/PAL video clock is the cpu clock multiplied by 11/7.

For other PSX/PSone PAL/NTSC variants, see:
Pinouts - CLK Pinouts

Vertical Timings

Timer1 can use the hblank signal as input, allowing to count scanlines (unless the display is configured to 0 pixels width, which would cause an endless hblank). The hblank signal is generated even during vertical blanking/retrace.

Horizontal Timings

Dotclocks:

Dots per scanline are, depending on horizontal resolution, and on PAL/NTSC:

Timer0 can use the dotclock as input, however, the Timer0 input "ignores" the fractional portions (in most cases, the values are rounded down, ie. with 340.6 dots/line, the timer increments only 340 times/line; the only value that is rounded up is 425.75 dots/line) (for example, due to the rounding, the timer isn't running exactly twice as fast in 512pix/PAL mode than in 256pix/PAL mode). The dotclock signal is generated even during horizontal/vertical blanking/retrace.

Frame Rates

Note

Above values include "hidden" dots and scanlines (during horizontal and vertical blanking/retrace).

4.15 GPU (MISC)

GP0(20h..7Fh) - Render Command Bits

Perspective (in-)correct Rendering

The PSX doesn't support perspective correct rendering: Assume a polygon to be rotated so that it's right half becomes more distant to the camera, and it's left half becomes closer. Due to the GTE's perspective division, the right half should appear smaller than the left half.
The GPU supports only linear interpolations for rendering - that is correct concerning the X and Y screen coordinates (which are still linear to each other, even after perspective division, since both are divided by the same value).
However, texture coordinates (and Gouraud shaded colors) are NOT linear to the screen coordinates, and so, the linear interpolated PSX graphics are often looking rather distorted, that especially for textures that contain straight lines. For color shading the problem is less obvious (since shading is kinda blurry anyways).

Perspective correct Rendering

For perspective correct rendering, the polygon's Z-coordinates would be needed to be passed from the GTE to the GPU, and, the GPU would then need to use that Z-coordinates to "undo" the perspective division for each pixel (that'd require some additional memory, and especially a powerful division unit, which isn't implemented in the hardware).
As a workaround, you can try to reduce the size of your polygons (the interpolation errors increase in the center region of larger polygons). Reducing the size would be only required for polygons that occupy a larger screen region (which may vary depending on the distance to the camera).
Ie. you may check the size AFTER perspective division, if it's too large, then break it into smaller parts (using the original coordinates, NOT the screen coordinates), and then pass the fragments to the GTE another time.
Again, perspective correction would be relevant only for certain textures (not for randomly dithered textures like sand, water, fire, grass, and not for untextured polygons, and of course not for 2D graphics, so you may exclude those from size reduction).

24bit RGB to 15bit RGB Dithering (enabled in Texpage attribute)

For dithering, VRAM is broken to 4x4 pixel blocks, depending on the location in that 4x4 pixel region, the corresponding dither offset is added to the 8bit R/G/B values, the result is saturated to +00h..+FFh, and then divided by 8, resulting in the final 5bit R/G/B values.

POLYGONs (triangles/quads) are dithered ONLY if they do use gouraud shading or texture blending.
LINEs are dithered (no matter if they are mono or do use gouraud shading).
RECTs are NOT dithered (no matter if they do use texture blending).

Shading information

"Texture RGB values control the brightness of the individual colors ($00-$7f). A value of $80 in a color will take the former value as data." (What...? probably means the "double brightness" effect... or does it want to tell that ALL colors of 80h..FFh have only single brightness.. rather than reaching double brightness at FFh...?)

Shading

The GPU has a shading function, which will scale the color of a primitive to a specified brightness. There are 2 shading modes: Flat shading, and gouraud shading. Flat shading is the mode in which one brightness value is specified for the entire primitive. In Gouraud shading mode, a different brightness value can be given for each vertex of a primitive, and the brightness between these points is automatically interpolated.

Semi Transparency

When semi transparency is set for a pixel, the GPU first reads the pixel it wants to write to, and then calculates the color it will write from the 2 pixels according to the semitransparency mode selected. Processing speed is lower in this mode because additional reading and calculating are necessary. There are 4 semitransparency modes in the GPU.

Draw to display enable

This will enable/disable any drawing to the area that is currently displayed. Not sure yet WHY one should want to disable that?
Also not sure HOW and IF it works... the SIZE of the display area is implied by the screen size - which is horizontally counted in CLOCK CYCLES, so, to obtain the size in PIXELS, the hardware would require to divide that value by the number of cycles per pixel, depending on the current resolution...?

5. Geometry Transformation Engine (GTE)

5.1 GTE Overview

GTE Operation

The GTE doesn't have any memory or I/O ports mapped to the CPU memory bus, instead, it's solely accessed via coprocessor opcodes:

GTE (memory-?) load and store instructions have a delay of 2 instructions, for any GTE commands or operations accessing that register. Any? That's wrong!
GTE instructions and functions should not be used in

If an instruction that reads a GTE register or a GTE command is executed before the current GTE command is finished, the CPU will hold until the instruction has finished. The number of cycles each GTE instruction takes is shown in the command list.

GTE Command Encoding (COP2 imm25 opcodes)

The MVMVA bits are used only by the MVMVA opcode (the bits are zero for all other opcodes).
The "sf" and "lm" bits are usually fixed (either set, or cleared, depending on the command) (for MVMVA, the bits are variable) (also, "sf" can be changed for some commands like SQR) (although they are usually fixed for most other opcodes, changing them might have some effect on some/all opcodes)?

GTE Data Register Summary (cop2r0-31)

GTE Control Register Summary (cop2r32-63)

5.2 GTE Registers

Note in some functions format is different from the one that's given here.

Matrix Registers

Each element is 16bit (1bit sign, 3bit integer, 12bit fraction). Reading the last elements (RT33,L33,LB3) returns the 16bit value sign-expanded to 32bit.

Translation Vector (TR) (Input, R/W?)

Each element is 32bit (1bit sign, 31bit integer).
Used only for MVMVA, RTPS, RTPT commands.

Background Color (BK) (Input?, R/W?)

Each element is 32bit (1bit sign, 19bit integer, 12bit fraction).

Far Color (FC) (Input?) (R/W?)

Each element is 32bit (1bit sign, 27bit integer, 4bit fraction).

Screen Offset and Distance (Input, R/W?)

The X and Y values are each 32bit (1bit sign, 15bit integer, 16bit fraction).
The H value is 16bit unsigned (0bit sign, 16bit integer, 0bit fraction). BUG: When reading the H register, the hardware does accidently \<sign-expand> the \<unsigned> 16bit value (ie. values +8000h..+FFFFh are returned as FFFF8000h..FFFFFFFFh) (this bug applies only to "mov rd,cop2r58" opcodes; the actual calculations via RTPS/RTPT opcodes are working okay).
The DQA value is only 16bit (1bit sign, 7bit integer, 8bit fraction).
The DQB value is 32bit (1bit sign, 7bit integer, 24bit? fraction).
Used only for RTPS/RTPT commands.

Average Z Registers (ZSF3/ZSF4=Input, R/W?) (OTZ=Result, R)

Used only for AVSZ3/AVSZ4 commands.

Screen XYZ Coordinate FIFOs

SX,SY,SZ are used as Output for RTPS/RTPT. Additionally, SX,SY are used as Input for NCLIP, and SZ is used as Input for AVSZ3/AVSZ4.
The SZn Fifo has 4 stages (required for AVSZ4 command), the SXYn Fifo has only 3 stages, and a special mirrored register: SXYP is a mirror of SXY2, the difference is that writing to SXYP moves SXY2/SXY1 to SXY1/SXY0, whilst writing to SXY2 (or any other SXYn or SZn registers) changes only the written register, but doesn't move any other Fifo entries.

16bit Vectors (R/W)

All elements are signed 16bit. The IRn and VZn elements occupy a whole 32bit register, reading these registers returns the 16bit value sign-expanded to 32bit. Note: IRn can be also indirectly accessed via IRGB/ORGB registers.

Color Register and Color FIFO

RES1 seems to be unused... looks like an unused Fifo stage... RES1 is read/write-able... unlike SXYP (for SXYn Fifo) it does not mirror to RGB2, nor does it have a move-on-write function...

Interpolation Factor

Used as Output for RTPS/RTPT, and as Input for various commands.

XX...

XX...

cop2r28 - IRGB - Color conversion Input (R/W)

Expands 5:5:5 bit RGB (range 0..1Fh) to 16:16:16 bit RGB (range 0000h..0F80h).

After writing to IRGB, the result can be read from IR3 after TWO nop's, and from IR1,IR2 after THREE nop's (for uncached code, ONE nop would work). When using IR1,IR2,IR3 as parameters for GTE commands, similar timing restrictions might apply... depending on when the specific commands use the parameters?

cop2r29 - ORGB - Color conversion Output (R)

Collapses 16:16:16 bit RGB (range 0000h..0F80h) to 5:5:5 bit RGB (range 0..1Fh). Negative values (8000h..FFFFh/80h) are saturated to 00h, large positive values (1000h..7FFFh/80h) are saturated to 1Fh, there are no overflow or saturation flags set in cop2r63 though.

Any changes to IR1,IR2,IR3 are reflected to this register (and, actually also to IRGB) (ie. ORGB is simply a read-only mirror of IRGB).

cop2r30 - LZCS - Count Leading Bits Source data (R/W)

cop2r31 - LZCR - Count Leading Bits Result (R)

Reading LZCR returns the leading 0 count of LZCS if LZCS is positive and the leading 1 count of LZCS if LZCS is negative. The results are in range 1..32.

cop2r63 (cnt31) - FLAG - Returns any calculation errors.

See GTE Saturation chapter.

5.3 GTE Saturation

Maths overflows are indicated in FLAG register. In most cases, the result is saturated to MIN/MAX values (except MAC0,MAC1,MAC2,MAC3 which aren't saturated). For IR1,IR2,IR3 many commands allow to select the MIN value via "lm" bit of the GTE opcode (though not all commands, RTPS/RTPT always act as if lm=0).

cop2r63 (cnt31) - FLAG - Returns any calculation errors.

Bit30-12 are read/write-able, ie. they can be set/reset by software, however, that's normally not required - all bits are automatically reset at the begin of a new GTE command.

Bit31 is apparently intended for RTPS/RTPT commands, since it triggers only on flags that are affected by these two commands, but even for that commands it's totally useless since one could as well check if FLAG is nonzero.
Note: Writing 32bit values to 16bit GTE registers by software does not trigger any overflow/saturation flags (and does not do any saturation), eg. writing 12008900h (positive 32bit) to a signed 16bit register sets that register to FFFF8900h (negative 16bit).

5.4 GTE Opcode Summary

GTE Command Summary (sorted by Real Opcode bits) (bit0-5)

Unknown if/what happens when using the "N/A" opcodes?

GTE Command Summary (sorted by Fake Opcode bits) (bit20-24)

The fake opcode number in bit20-24 has absolutely no effect on the hardware, it seems to be solely used to (or not to) confuse developers. Having the opcodes sorted by their fake numbers gives a more or less well arranged list:

For the sort-effect, DCPT should use fake=08h, but Sony seems to have accidently numbered it fake=0Fh in their devkit (giving it the same fake number as for NCDT). Also, "Wipeout 2097" accidently uses 0140006h (fake=01h and distorted bit18) instead of 1400006h (fake=14h) for NCLIP.

Additional Functions

The LZCS/LZCR registers offer a Count-Leading-Zeroes/Leading-Ones function.
The IRGB/ORGB registers allow to convert between 48bit and 15bit RGB colors.
These registers work without needing to send any COP2 commands. However, unlike for commands (which do automatically halt the CPU when needed), one must insert dummy opcodes between writing and reading the registers.

5.5 GTE Coordinate Calculation Commands

COP2 0180001h - 15 Cycles - RTPS - Perspective Transformation (single)

COP2 0280030h - 23 Cycles - RTPT - Perspective Transformation (triple)

RTPS performs final Rotate, translate and perspective transformation on vertex V0. Before writing to the FIFOs, the older entries are moved one stage down. RTPT is same as RTPS, but repeats for V1 and V2. The "sf" bit should be usually set.

If the result of the "(((H*20000h/SZ3)+1)/2)" division is greater than 1FFFFh, then the division result is saturated to +1FFFFh, and the divide overflow bit in the FLAG register gets set; that happens if the vertex is exceeding the "near clip plane", ie. if it is very close to the camera (SZ3\<=H/2), exactly at the camara position (SZ3=0), or behind the camera (negative Z coordinates are saturated to SZ3=0). For details on the division, see:
GTE Division Inaccuracy
For "far plane clipping", one can use the SZ3 saturation flag (MaxZ=FFFFh), or the IR3 saturation flag (MaxZ=7FFFh) (eg. used by Wipeout 2097), or one can compare the SZ3 value with any desired MaxZ value by software.
Note: The command does saturate IR1,IR2,IR3 to -8000h..+7FFFh (regardless of lm bit). When using RTP with sf=0, then the IR3 saturation flag (FLAG.22) gets set \<only> if "MAC3 SAR 12" exceeds -8000h..+7FFFh (although IR3 is saturated when "MAC3" exceeds -8000h..+7FFFh).

COP2 1400006h - 8 Cycles - NCLIP - Normal clipping

The sign of the result indicates whether the polygon coordinates are arranged clockwise or anticlockwise (ie. whether the front side or backside is visible). If the result is zero, then it's neither one (ie. the vertices are all arranged in a straight line). Note: The GPU probably renders straight lines as invisble 0 pixel width lines?

COP2 158002Dh - 5 Cycles - AVSZ3 - Average of three Z values (for Triangles)

COP2 168002Eh - 6 Cycles - AVSZ4 - Average of four Z values (for Quads)

Adds three or four Z values together and multplies them by a fixed point value. The result can be used as index in the GPU's Ordering Table (OT).
GPU Depth Ordering
The scaling factors would be usually ZSF3=N/30h and ZSF4=N/40h, where "N" is the number of entries in the OT (max 10000h). SZn and OTZ are unsigned 16bit values, for whatever reason ZSFn registers are signed 16bit values (negative values would allow a negative result in MAC0, but would saturate OTZ to zero).

5.6 GTE General Purpose Calculation Commands

COP2 0400012h - 8 Cycles - MVMVA(sf,mx,v,cv,lm)

Multiply vector by matrix and vector addition.

Calculation:

Multiplies a vector with either the rotation matrix, the light matrix or the color matrix and then adds the translation vector or background color vector.
The GTE also allows selection of the far color vector (FC), but this vector is not added correctly by the hardware: The return values are reduced to the last portion of the formula, ie. MAC1=(Mx13*Vx3) SAR (sf*12), and similar for MAC2 and MAC3, nethertheless, some bits in the FLAG register seem to be adjusted as if the full operation would have been executed. Setting Mx=3 selects a garbage matrix (with elements -60h, +60h, IR0, RT13, RT13, RT13, RT22, RT22, RT22).

COP2 0A00428h+sf*80000h - 5 Cycles - SQR(sf) - Square vector

Calculates the square of a vector. The result is, of course, always positive, so the "lm" flag for negative saturation has no effect.

COP2 170000Ch+sf*80000h - 6 Cycles - OP(sf,lm) - Outer product of 2 vectors

Calculates the cross product of two signed 16bit vectors. Note: D1,D2,D3 are meant to be the RT11,RT22,RT33 elements of the RT matrix "misused" as vector. lm should be usually zero.

LZCS/LZCR registers - ? Cycles - Count-Leading-Zeroes/Leading-Ones

The LZCS/LZCR registers offer a Count-Leading-Zeroes/Leading-Ones function.

5.7 GTE Color Calculation Commands

COP2 0C8041Eh - 14 Cycles - NCS - Normal color (single)

COP2 0D80420h - 30 Cycles - NCT - Normal color (triple)

COP2 108041Bh - 17 Cycles - NCCS - Normal Color Color (single vector)

COP2 118043Fh - 39 Cycles - NCCT - Normal Color Color (triple vector)

COP2 0E80413h - 19 Cycles - NCDS - Normal color depth cue (single vector)

COP2 0F80416h - 44 Cycles - NCDT - Normal color depth cue (triple vectors)

In: V0=Normal vector (for triple variants repeated with V1 and V2), BK=Background color, RGBC=Primary color/code, LLM=Light matrix, LCM=Color matrix, IR0=Interpolation value.

COP2 138041Ch - 11 Cycles - CC(lm=1) - Color Color

COP2 1280414h - 13 Cycles - CDP(...) - Color Depth Que

In: [IR1,IR2,IR3]=Vector, RGBC=Primary color/code, LCM=Color matrix, BK=Background color, and, for CDP, IR0=Interpolation value, FC=Far color.

COP2 0680029h - 8 Cycles - DCPL - Depth Cue Color light

COP2 0780010h - 8 Cycles - DPCS - Depth Cueing (single)

COP2 0x8002Ah - 17 Cycles - DPCT - Depth Cueing (triple)

COP2 0980011h - 8 Cycles - INTPL - Interpolation of a vector and far color

In: [IR1,IR2,IR3]=Vector, FC=Far Color, IR0=Interpolation value, CODE=MSB of RGBC, and, for DCPL, R,G,B=LSBs of RGBC.

DPCT executes thrice, and reads the R,G,B values from RGB0 (ie. reads from the Bottom of the Color FIFO, instead of from the RGBC register) (the CODE value is kept read from RGBC as usually), so, after DPCT execution, the RGB0,RGB1,RGB2 Fifo entries are modified.

COP2 190003Dh - 5 Cycles - GPF(sf,lm) - General purpose Interpolation

COP2 1A0003Eh - 5 Cycles - GPL(sf,?) - General Interpolation with base

Note: Although the SHL in GPL is theoretically undone by the SAR, 44bit overflows can occur internally when sf=1.

Details on "MAC+(FC-MAC)*IR0"

Note: Above "[IR1,IR2,IR3]=(FC-MAC)" is saturated to -8000h..+7FFFh (ie. as if lm=0), anyways, further writes to [IR1,IR2,IR3] (within the same command) are saturated as usually (ie. depening on lm setting).

Details on "(LLM*V0) SAR (sf*12)" and "(BK*1000h + LCM*IR) SAR (sf*12)"

Works like MVMVA command (see there), but with fixed Tx/Vx/Mx parameters, the sf/lm bits can be changed and do affect the results (although normally both bits should be set for use with color matrices).

Notes

The 8bit RGB values written to the top of Color Fifo are the 32bit MACn values divided by 16, and saturated to +00h..+FFh, and of course, the older Fifo entries are moved downwards. Note that, at the GPU side, the meaning of the RGB values depends on whether or not texture blending is used (for untextured polygons FFh is max brightness) (for texture blending FFh is double brightness and 80h is normal brightness).
The 8bit CODE value is intended to contain a GP0(20h..7Fh) Rendering command, allowing to automatically merge the 8bit command number, with the 24bit color value.
The IRGB/ORGB registers allow to convert between 48bit and 15bit RGB colors.
Although the result of the commands in this chapter is written to the Color FIFO, some commands like GPF/GPL may be also used for other purposes (eg. to scale or scale/translate single vertices).

5.8 GTE Division Inaccuracy

GTE Division Inaccuracy (for RTPS/RTPT commands)

Basically, the GTE division does (attempt to) work as so (using 33bit maths):

alternatly, below would give (almost) the same result (using 32bit maths):

in both cases, the result is saturated about as so:

However, the real GTE hardware is using a fast, but less accurate division mechanism (based on Unsigned Newton-Raphson (UNR) algorithm):

the GTE's unr_table[000h..100h] consists of following values:

Above can be generated as "unr_table[i]=min(0,(40000h/(i+100h)+1)/2-101h)".
Some special cases: NNNNh/0001h uses a big multiplier (d=20000h), in practice, this can occur only for 0000h/0001h and 0001h/0001h (due to the H\<SZ3*2 overflow check).
The min(1FFFFh) limit is needed for cases like FE3Fh/7F20h, F015h/780Bh, etc. (these do produce UNR result 20000h, and are saturated to 1FFFFh, but without setting overflow FLAG bits).

6. Macroblock Decoder (MDEC)

The MDEC is a JPEG-style Macroblock Decoder, that can decompress pictures (or a series of pictures, for being displayed as a movie).

6.1 MDEC I/O Ports

1F801820h - MDEC0 - MDEC Command/Parameter Register (W)

Used to send command word, followed by parameter words to the MDEC (usually, only the command word is written to this register, and the parameter words are transferred via DMA0).

1F801820h.Read - MDEC Data/Response Register (R)

The data is always output as a 8x8 pixel bitmap, so, when manually reading from this register and using colored 16x16 pixel macroblocks, the data from four 8x8 blocks must be re-ordered accordingly (usually, the data is received via DMA1, which is doing the re-ordering automatically). For monochrome 8x8 macroblocks, no re-ordering is needed (that works with DMA1 too).

1F801824h - MDEC1 - MDEC Status Register (R)

If there's data in the output fifo, then the Current Block bits are always set to the current output block number (ie. Y1..Y4; or Y for mono) (this information is apparently passed to the DMA1 controller, so that it knows if and how it must re-order the data in RAM). If the output fifo is empty, then the bits indicate the currently processsed incoming block (ie. Cr,Cb,Y1..Y4; or Y for mono).

1F801824h - MDEC1 - MDEC Control/Reset Register (W)

The data requests are required to be enabled for using DMA (and for reading the request status flags by software). The Data-Out request acts a bit strange: It gets set when a block is available, but, it gets cleared after reading the first some words of that block (nethertheless, one can keep reading the whole block, until the fifo-empty flag gets set).

DMA

MDEC decompression uses a lot of DMA channels,

DMA0 and DMA1 should be usually used with a blocksize of 20h words. If necessary, the parameters for the MDEC(1) command should be padded with FE00h halfwords to match the 20h words (40h halfwords) DMA blocksize.

6.2 MDEC Commands

MDEC(1) - Decode Macroblock(s)

This command is followed by one or more Macroblock parameters (usually, all macroblocks for the whole image are sent at once).

MDEC(2) - Set Quant Table(s)

The command word is followed by 64 unsigned parameter bytes for the Luminance Quant Table (used for Y1..Y4), and if Command.Bit0 was set, by another 64 unsigned parameter bytes for the Color Quant Table (used for Cb and Cr).

MDEC(3) - Set Scale Table

The command is followed by 64 signed halfwords with 14bit fractional part, the values should be usually/always the same values (based on the standard JPEG constants, although, MDEC(3) allows to use other values than that constants).

MDEC(0) - No function

This command has no function. Command bits 25-28 are reflected to Status bits 23-26 as usually. Command bits 0-15 are reflected to Status bits 0-15 (similar as the "number of parameter words" for MDEC(1), but without the "minus 1" effect, and without actually expecting any parameters).

MDEC(4..7) - Invalid

These commands act identical as MDEC(0).

6.3 MDEC Decompression

decode_colored_macroblock ;MDEC(1) command (at 15bpp or 24bpp depth)

decode_monochrome_macroblock ;MDEC(1) command (at 4bpp or 8bpp depth)

rl_decode_block(blk,src,qt)

fast_idct_core(blk) ;fast "idct_core" version

Fast code with only 80 multiplications, works only if the scaletable from MDEC(3) command contains standard values (which is the case for all known PSX games).

real_idct_core(blk) ;low level "idct_core" version

Low level code with 1024 multiplications, using the scaletable from the MDEC(3) command. Computes dst=src*scaletable (using normal matrix maths, but with "src" being diagonally mirrored, ie. the matrices are processed column by column, instead of row by column), repeated with src/dst exchanged.

The "(sum+0fffh)/2000h" part is meant to strip fractional bits, and to round-up the result if the fraction was BIGGER than 0.5. The hardware appears to be working roughly like that, still the results aren't perfect.
Maybe the real hardware is doing further roundings in other places, possibly stripping some fractional bits before summing up "sum", possibly stripping different amounts of bits in the two "pass" cycles, and possibly keeping a final fraction passed on to the y_to_mono stage.

yuv_to_rgb(xx,yy)

Note: The exact fixed point resolution for "yuv_to_rgb" is unknown. And, there's probably also some 9bit limit (similar as in "y_to_mono").

y_to_mono

set_iqtab ;MDEC(2) command

iqtab_core(iq,src) ;src = 64 unsigned paramter bytes

Note: For "fast_idct_core" one could precalc "iq[i]=src[i]*scalezag[i]", but that would conflict with the RLE saturation/rounding steps (though those steps aren't actually required, so a very-fast decoder could omit them).

scalefactor[0..7] = cos((0..7)*90'/8) ;for [1..7]: multiplied by sqrt(2)

zigzag[0..63] =

scalezag[0..63] (precalulated factors, for "fast_idct_core")

zagzig[0..63] (reversed zigzag table)

set_scale_table: ;MDEC(3) command

This command defines the IDCT scale matrix, which should be usually/always:

Note that the hardware does actually use only the upper 13bit of those 16bit values. The values are choosen like so,

whereas, s0..s7 = scalefactor[0..7], multiplied by sqrt(2) (ie. by 1.414), and multiplied by 4000h (ie. with 14bit fractional part).

6.4 MDEC Data Format

Colored Macroblocks (16x16 pixels) (in 15bpp or 24bpp depth mode)

Each macroblock consists of six blocks: Two low-resolution blocks with color information (Cr,Cb) and four full-resolution blocks with luminance (grayscale) information (Y1,Y2,Y3,Y4). The color blocks are zoomed from 8x8 to 16x16 pixel size, merged with the luminance blocks, and then converted from YUV to RGB format.

Native PSX files are usually containing vertically arranged Macroblocks (eg. allowing to send them to the GPU as 16x240 portion) (JPEG-style horizontally arranged Macroblocks would require to send the data in 16x16 pixel portions to the GPU) (something like 320x16 won't work, since that'd require to wrap from the bottom of the first macroblock to the top of the next macroblock).

Monochrome Macroblocks (8x8 pixel) (in 4bpp or 8bpp depth mode)

Each macroblock consist of only one block: with luminance (grayscale) information (Y), the data comes out as such (it isn't converted to RGB).

The output is an 8x8 bitmap (not 16x16), so it'd be send to the GPU as 8x8 pixel rectangle, or multiple blocks at once as 8x240 pixel rectangle. Since the data isn't RGB, it should be written to Texture memory (and then it can be forwarded to the frame buffer in form of a texture with monochrome 15bit palette with 32 grayscales). Alternately, one could convert the 8bpp image to 24bpp by software (this would allow to use 256 grayscales).

Blocks (8x8 pixels)

An (uncompressed) block consists of 64 values, representing 8x8 pixels. The first (upper-left) value is an absolute value (called "DC" value), the remaining 63 values are relative to the DC value (called "AC" values). After decompression and zig-zag reordering, the data in unfiltered horizontally and vertically (IDCT conversion, ie. the relative "AC" values are converted to absolute "DC" values).

.STR Files

PSX Video files are usually having file extension .STR (for "Streaming").

MDEC vs JPEG

The MDEC data format is very similar to the JPEG file format, the main difference is that JPEG uses Huffman compressed blocks, whilst MDEC uses Run-Length (RL) compressed blocks.
The (uncompressed) blocks are same as in JPEGs, using the same zigzag ordering, AC to DC conversion, and YUV to RGB conversion (ie. the MDEC hardware can be also used to decompress JPEGs, when handling the file header and huffman decompression by software).
Some other differences are that MDEC has only 2 fixed-purpose quant tables, whilst JPEGs \<can> use up to 4 general-purpose quant tables. Also, JPEGs \<can> use other color resolutions than the 8x8 color info for 16x16 pixels. Whereas, JPEGs \<can> do

that stuff, but most standard JPEG files aren't actually using 4 quant tables, nor higher color resolution.

Run-Length compressed Blocks

Within each block the DCT information and RLE compressed data is stored:

DCT (1st value)

DCT data has the quantization factor and the Direct Current (DC) reference.

Contains the absolute DC value (the upper-left value of the 8x8 block).

RLE (Run length data, for 2nd through 64th value)

Example: AC values "000h,000h,123h" would be compressed as "(2 shl 10)+123h".

EOB (End Of Block)

Indicates the end of a 8x8 pixel block, causing the rest of the block to be padded with zero AC values.

EOB isn't required if the block was already fully defined (up to including blk[63]), however, most games seem to append EOB to all blocks (although it's just acting as dummy/padding value in case of fully defined blocks).

Dummy halfwords

Data is sent in units of words (or, when using DMA, even in units of 32-words), which is making it neccessary to send some dummy halfwords (unless the compressed data size should match up the transfer unit). The value FE00h can be used as dummy value: When FE00h appears at the begin of a new block, or after the end of block, then it is simply ignored by the hardware (if it occurs elsewhere, then it acts as EOB end code, as described above).

7. Sound Processing Unit (SPU)

7.1 SPU Overview

SPU I/O Port Summary

SPU Memory layout (512Kbyte RAM)

As shown above, the first 4Kbytes are used as special capture buffers, and, if desired, one can also use the Reverb hardware to capture output from other voice(s).
The SPU memory is not mapped to the CPU bus, it can be accessed only via I/O, or via DMA transfers (DMA4).

Voices

The SPU has 24 hardware voices. These voices can be used to reproduce sample data, noise or can be used as frequency modulator on the next voice. Each voice has it's own programmable ADSR envelope filter. The main volume can be programmed independently for left and right output.

Voice Capabilities

All 24 voices are having exactly the same capabilities(?), with the exception that Voice 1 and 3 are having a special Capture feature (see SPU Memory map).
There seems to be no way to produce square waves (without storing a square wavefrom in memory... although, since SPU RAM isn't connected to the CPU bus, the "useless" DMA for square wave data wouldn't slowdown the CPU bus)?

Additional Sound Inputs

External Audio can be input (from the Expansion Port?), and the CDROM drive can be commanded to playback normal Audio CDs (via Play command), or XA-ADPCM sectors (via Read command), and to pass that data to the SPU.

Unstable and Delayed I/O

The SPU occassionally seems to "miss" I/O writes (not sure if that can be fixed by any Memory Control settings?), a stable workaround is too write all values twice (except of course, Fifo writes). The SPU seems to process written values at 44100Hz rate (so it may take 1/44100 seconds (300h clock cycles) until it has actually realized the new value).

Mono/Stereo Audio Output

The standard PSX Audio cables have separate Left/Right signals, that is good for stereo TVs, but, when using a normal mono TV, only one of the two audio signals (Left or Right) can be connected. PSX programs should thus offer an option to disable stereo effects, and to output an equal volume to both cables.

SPU Bus-Width

The SPU is connected to a 16bit databus. 8bit/16bit/32bit reads and 16bit/32bit writes are implemented. However, 8bit writes are NOT implemented: 8bit writes to ODD addresses are simply ignored (without causing any exceptions), 8bit writes to EVEN addresses are executed as 16bit writes (eg. "movp r1,12345678h, movb [spu_port],r1" will write 5678h instead of 78h).

7.2 SPU ADPCM Samples

The SPU supports only ADPCM compressed samples (uncompressed samples seem to be totally unsupported; leaving apart that one can write uncompressed 16bit PCM samples to the Reverb Buffer, which can be then output at 22050Hz, as long as they aren't overwritten by the hardware).

1F801C06h+N*10h - Voice 0..23 ADPCM Start Address (R/W)

This register holds the sample start address (not the current address, ie. the register doesn't increment during playback).

Writing to this register has no effect on the currently playing voice.
The start address is copied to the current address upon Key On.

1F801C0Eh+N*10h - Voice 0..23 ADPCM Repeat Address (R/W)

If the hardware finds an ADPCM header with Loop-Start-Bit, then it copies the current address to the repeat addresss register.
If the hardware finds an ADPCM header with Loop-Stop-Bit, then it copies the repeat addresss register setting to the current address; that, \<after> playing the current ADPCM block.

Normally, repeat works automatically via the above start/stop bits, and software doesn't need to deal with the Repeat Address Register. However, reading from it may be useful to sense if the hardware has reached a start bit, and writing may be also useful in some cases, eg. to redirect a one-shot sample (with stop-bit, but without any start-bits) to a silent-loop located elsewhere in memory.

Sample Data (SPU-ADPCM)

Samples consist of one or more 16-byte blocks:

Flag Bits (in 2nd byte of ADPCM Header)

Possible combinations for Bit0-1 are:

Looped and One-shot Samples

The Loop Start/End flags in the ADPCM Header allow to play one or more sample block(s) in a loop, that can be either all block(s) endless repeated, or only the last some block(s) of the sample.
There's no way to stop the output, so a one-shot sample must be followed by dummy block (with Loop Start/End flags both set, and all data nibbles set to zero; so that the block gets endless repeated, but doesn't produce any sound).

SPU-ADPCM vs XA-ADPCM

The PSX supports two ADPCM formats: SPU-ADPCM (as described above), and XA-ADPCM. XA-ADPCM is decompressed by the CDROM Controller, and sent directly to the sound mixer, without needing to store the data in SPU RAM, nor needing to use a Voice

channel.
The actual decompression algorithm is the same for both formats. However, the XA nibbles are arranged in different order, and XA uses 2x28 nibbles per block (instead of 2x14), XA blocks can contain mono or stereo data, XA supports only two sample rates, and, XA doesn't support looping.

7.3 SPU ADPCM Pitch

1F801C04h+N*10h - Voice 0..23 ADPCM Sample Rate (R/W) (VxPitch)

Defines the ADPCM sample rate (1000h = 44100Hz). This register (and PMON) does affect only the ADPCM sample frequency (but not on the Noise frequency, which is defined - and shared for all voices - in the SPUCNT register).

1F801D90h - Voice 0..23 Pitch Modulation Enable Flags (PMON)

Pitch modulation allows to generate "Frequency Sweep" effects by mis-using the amplitude from channel (x-1) as pitch factor for channel (x).

For example, output a very loud 1Hz sine-wave on channel 4 (with ADSR volume 4000h, and with Left/Right volume=0; unless you actually want to output it to the speaker). Then additionally output a 2kHz sine wave on channel 5 with PMON.Bit5 set. The "2kHz" sound should then repeatedly sweep within 1kHz..3kHz range (or, for a more decent sweep in 1.8kHz..2.2kHz range, drop the ADSR volume of channel 4).

Pitch Counter

The pitch counter is adjusted at 44100Hz rate as follows:

Counter.Bit12 and up indicates the current sample (within a ADPCM block).
Counter.Bit3..11 are used as 8bit gaussian interpolation index.

Maximum Sound Frequency

The Mixer and DAC supports a 44.1kHz output rate (allowing to produce max 22.1kHz tones). The Reverb unit supports only half the frequency.
The pitch counter supports sample rates up to 176.4kHz. However, exceeding the 44.1kHz limit causes the hardware to skip samples (or actually: to apply incomplete interpolation on the 'skipped' samples).
VxPitch can be theoretically 0..FFFFh (max 705.6kHz), normally 4000h..FFFFh are simply clipped to max=4000h (176.4kHz). Except, 4000h..FFFFh could be used with pitch modulation (as they are multiplied by 0.00..1.99 before clipping; in practice this works only for 4000h..7FFFh; as values 8000h..FFFFh are mistaken as signed values).

4-Point Gaussian Interpolation

Interpolation is applied on the 4 most recent 16bit ADPCM samples (new,old,older,oldest), using bit4-11 of the pitch counter as 8bit interpolation index (i=00h..FFh):

The Gauss table contains the following values (in hex):

The PSX table is a bit different as the SNES table: Values up to 3569h are smaller as on SNES, the remaining values are bigger as on SNES, and the width of the PSX table entries is 4bit higher as on SNES.
The PSX table is slightly bugged: Theoretically, each four values (gauss[000h+i], gauss[0FFh-i], gauss[100h+i], gauss[1FFh-i]) should sum up to 8000h, but in practice they do sum up to 7F7Fh..7F81h (fortunately the PSX sum doesn't exceed the 8000h limit; meaning that the PSX interpolations won't overflow, which has been a hardware glitch on the SNES).

Waveform Examples

7.4 SPU Volume and ADSR Generator

1F801C08h+N*10h - Voice 0..23 Attack/Decay/Sustain/Release (ADSR) (32bit)

The Attack phase gets started when the software sets the voice ON flag (see below), the hardware does then automatically go through Attack/Decay/Sustain, and switches from Sustain to Release when the software sets the Key OFF flag.

1F801D80h - Mainvolume left

1F801D82h - Mainvolume right

1F801C00h+N*10h - Voice 0..23 Volume Left

1F801C02h+N*10h - Voice 0..23 Volume Right

Fixed Volume Mode (when Bit15=0):

Sweep Volume Mode (when Bit15=1):

Sweep is another Volume envelope, additionally to the ADSR volume envelope (unlike ADSR, sweep can be used for stereo effects, such like blending from left to right). Sweep starts at the current volume (which can be set via Bit15=0, however, caution - the Bit15=0 setting isn't applied until the next 44.1kHz cycle; so setting the initial level with Bit15=0, followed by the sweep parameter with Bit15=1 works only if there's a

suitable delay between the two operations). Once when sweep is started, the current volume level increases to +7FFFh, or decreases to 0000h.
Sweep Phase should be equal to the sign of the current volume (not yet tested, in the negative mode it does probably "increase" to -7FFFh?). The Phase bit seems to have no effect in Exponential Decrease mode.

1F801DB0h - CD Audio Input Volume (for normal CD-DA, and compressed XA-ADPCM)

1F801DB4h - External Audio Input Volume

Note: The CDROM controller supports additional CD volume control (including ability to convert stereo CD output to mono, or to swap left/right channels).

Envelope Operation depending on Shift/Step/Mode/Direction

Exponential Increase is a fake (simply changes to a slower linear increase rate at higher volume levels).

1F801C0Ch+N*10h - Voice 0..23 Current ADSR volume (R/W)

Reportedly Release can go down to -1 (FFFFh), but that isn't true; and release ends at 0... or does THAT depend on an END flag found in the sample-data?
The register is read/writeable, writing allows to let the ADSR generator to "jump" to a specific volume level. But, ACTUALLY, the ADSR generator does overwrite the setting (from another internal register) whenever applying a new Step?!

1F801DB8h - Current Main Volume Left/Right

1F801E00h+voice*04h - Voice 0..23 Current Volume Left/Right

These are internal registers, normally not used by software (the Volume settings are usually set via Ports 1F801D80h and 1F801C00h+N*10h).

Note

Negative volumes are phase inverted, otherwise same as positive.

7.5 SPU Voice Flags

1F801D88h - Voice 0..23 Key ON (Start Attack/Decay/Sustain) (KON) (W)

Starts the ADSR Envelope, and automatically initializes ADSR Volume to zero, and copies Voice Start Address to Voice Repeat Address.

1F801D8Ch - Voice 0..23 Key OFF (Start Release) (KOFF) (W)

For a full ADSR pattern, OFF would be usually issued in the Sustain period, however, it can be issued at any time (eg. to abort Attack, skip the Decay and Sustain periods, and switch immediately to Release).

1F801D9Ch - Voice 0..23 ON/OFF (status) (ENDX) (R)

The bits get CLEARED when setting the corresponding KEY ON bits.
The bits get SET when reaching an LOOP-END flag in ADPCM header.bit0.

R/W

Key On and Key Off should be treated as write-only (although, reading returns the most recently 32bit value, this doesn't doesn't provide any status information about whether sound is on or off).
The on/off (status) (ENDX) register should be treated read-only (writing is possible in so far that the written value can be read-back for a short moment, however, thereafter the hardware is overwriting that value).

7.6 SPU Noise Generator

1F801D94h - Voice 0..23 Noise mode enable (NON)

SPU Noise Generator

The signed 16bit output Level is calculated as so (repeated at 44.1kHz clock):

Note that the Noise frequency is solely controlled by the Shift/Step values in SPUCNT register (the ADPCM Sample Rate has absolutely no effect on noise), so when using noise for multiple voices, all of them are forcefully having the same frequency; the only workaround is to store a random ADPCM pattern in SPU RAM, which can be then used with any desired sample rate(s).

7.7 SPU Control and Status Register

1F801DAAh - SPU Control Register (SPUCNT)

Changes to bit0-5 aren't applied immediately; after writing to SPUCNT, it'd be usually recommended to wait until the LSBs of SPUSTAT are updated accordingly. Before setting a new Transfer Mode, it'd be recommended first to set the "Stop" mode (and, again, wait until Stop is applied in SPUSTAT).

1F801DAEh - SPU Status Register (SPUSTAT) (R)

When switching SPUCNT to DMA-read mode, status bit9 and bit7 aren't set immediately (apparently the SPU is first internally collecting the data in the Fifo, before transferring it).
Bit11 indicates if data is currently written to the first or second half of the four 1K-byte capture buffers (for CD Audio left/right, and voice 1/3). Note: Bit11 works only if Bit2 and/or Bit3 of Port 1F801DACh are set.
The SPUSTAT register should be treated read-only (writing is possible in so far that the written value can be read-back for a short moment, however, thereafter the hardware is overwriting that value).

7.8 SPU Memory Access

1F801DA6h - Sound RAM Data Transfer Address

Used for manual write and DMA read/write SPU memory. Writing to this registers stores the written value in 1F801DA6h, and does additional store the value (multiplied by 8) in another internal "current address" register (that internal register does increment during transfers, whilst the 1F801DA6h value DOESN'T increment).

1F801DA8h - Sound RAM Data Transfer Fifo

Used for manual-write. Not sure if it can be also used for manual read?

1F801DACh - Sound RAM Data Transfer Control (should be 0004h)

The Transfer Type selects how data is forwarded from Fifo to SPU RAM:

Rep2 skips the 2nd halfword, Rep4 skips 2nd..4th, Rep8 skips 1st..7th.
Fill uses only the LAST halfword in Fifo, that might be useful for memfill purposes, although, the length is probably determined by the number of writes to the Fifo (?) so one must still issue writes for ALL halfwords...?
Note:
The above rather bizarre results apply to WRITE mode. In READ mode, the register causes the same halfword to be read 2/4/8 times (for rep2/4/8).

SPU RAM Manual Write

- Be sure that [1F801DACh] is set to 0004h
- Set SPUCNT to "Stop" (and wait until it is applied in SPUSTAT)
- Set the transfer address
- Write 1..32 halfword(s) to the Fifo

- Set SPUCNT to "Manual Write" (and wait until it is applied in SPUSTAT)
- Wait until Transfer Busy in SPUSTAT goes off (that, AFTER above apply-wait)
  For multi-block transfers: Repeat the above last three steps (that is rarely done by any games, but it is done by the BIOS intro; observe that waiting for SPUCNT writes being applied in SPUSTAT won't work in that case (since SPUCNT was already in manual write mode from previous block), so one must instead use some hardcoded delay of at least 300h cycles; the BIOS is using a much longer bizarre delay though).

SPU RAM DMA-Write

- Be sure that [1F801DACh] is set to 0004h
- Set SPUCNT to "Stop" (and wait until it is applied in SPUSTAT)
- Set the transfer address
- Set SPUCNT to "DMA Write" (and wait until it is applied in SPUSTAT)
- Start DMA4 at CPU Side (blocksize=10h, control=01000201h)
- Wait until DMA4 finishes (at CPU side)

SPU RAM Manual-Read

As by now, there's no known method for reading SPU RAM without using DMA.

SPU RAM DMA-Read (stable reading, with [1F801014h].bit24-27 = nonzero)

- Be sure that [1F801014h] is set to 220931E1h (bit24-27 MUST be nonzero)
- Be sure that [1F801DACh] is set to 0004h
- Set SPUCNT to "Stop" (and wait until it is applied in SPUSTAT)
- Set the transfer address
- Set SPUCNT to "DMA Read" (and wait until it is applied in SPUSTAT)
- Start DMA4 at CPU Side (blocksize=10h, control=01000200h)
- Wait until DMA4 finishes (at CPU side)

SPU RAM DMA-Read (unstable reading, with [1F801014h].bit24-27 = zero)

Below describes some dirt effects and some trickery to get around those dirt effects.

With [1F801014h].bit24-27=zero, reading SPU RAM via DMA works glitchy:
The first received halfword within each block is FFFFh. So with a DMA blocksize of 10h words (=20h halfwords), the following is received:

that'd theoretically match the SPU Fifo Size, but, because of the inserted FFFFh value, the last Fifo entry isn't received, ie. halfword[1Fh,3Fh] are lost. As a workaround, one can increase the DMA blocksize to 11h words, and then the following is received:

this time, all data is received, but after the transfer one must still remove the FFFFh values, and the duplicated halfwords by software. Aside from the \<inserted> FFFFh values there are occassionaly some unstable halfwords ORed by FFFFh (or ORed by other garbage values), this can be fixed by using "rep2" mode, which does then receive:

again, remove the first halfword (FFFFh) and the last halfword, and, take the duplicated halfwords ANDed together. Unstable values occur only every 32 halfwords or so (probably when the SPU is simultaneously reading ADPCM data), but do never occur on two continous halfwords, so, even if one halfword was ORed by garbage, the other halfword is always correct, and the result of the ANDed halfwords is 100% stable.
Note: The unstable reading does NOT occur always, when resetting the PSX a couple of times it does occassionally boot-up with totally stable reading, since there is no known way to activate the stable "mode" via I/O ports, the stable/unstable behaviour does eventually depend on internal clock dividers/multipliers, and whether they are starting in

sync with the CPU or not.
Caution: The "rep2" trick cannot be used in combination with reverb (reverb seems to be using the Port 1F801DACh Sound RAM Data Transfer Control, too).

7.9 SPU Interrupt

1F801DA4h - Sound RAM IRQ Address (IRQ9)

See also: SPUCNT (IRQ enable/disable/acknowledge) and SPUSTAT (IRQ flag).

Voice Interrupt

Triggers an IRQ when a voice reads ADPCM data from the IRQ address.
Mind that ADPCM cannot be stopped (uh, except, probably they CAN be stopped, by setting the sample rate to zero?), all voices are permanently reading data from SPU RAM - even in Noise mode, even if the Voice Volume is zero, and even if the ADSR pattern has finished the Release period - so even inaudible voices can trigger IRQs. To prevent unwanted IRQs, best set all unused voices to an endless looped dummy ADPCM block.
For stable IRQs, the IRQ address should be aligned to the 16-byte ADPCM blocks. If if the IRQ address is in the middle of a 16-byte ADPCM block, then the IRQ doesn't seem to trigger always (unknown why, but it seems to occassionally miss IRQs, even if the block gets repeated several times).

Capture Interrupt

Setting the IRQ address to 0000h..01FFh (aka byte address 00000h..00FFFh) will trigger IRQs on writes to the four capture buffers. Each of the four buffers contains 400h bytes (=200h samples), so the IRQ rate will be around 86.13Hz (44100Hz/200h).
CD-Audio capture is always active (even CD-Audio output is disabld in SPUCNT, and even if the drive door is open). Voice capture is (probably) also always active (even if the corresponding voice is off).
Capture IRQs do NOT occur if 1F801DACh.bit3-2 are both zero.

Reverb Interrupt

Reverb is also triggering interrupts if the IRQ address is located in the reverb buffer area. Unknown \<which> of the various reverb read(s) and/or reverb write(s) are triggering interrupts.

Data Transfers

Data Transfers (usually via DMA4) to/from SPU-RAM do also trap SPU interrupts.

Note

The IRQ Address is used in the following games (not exhaustive): Metal Gear Solid: Dialogue and Konami intro. Legend of Mana Hercules: the memory card loading screen's lip sync. Tokimeki Memorial 2 Crash Team Racing: Lip sync, requires capture buffers. The Misadventures of Tron Bonne: Dialogues. Need For Speed 3: (somewhat?).

7.10 SPU Reverb Registers

Reverb Volume and Address Registers (R/W)

All volume registers are signed 16bit (range -8000h..+7FFFh).
All src/dst/disp/base registers are addresses in SPU memory (divided by 8), src/dst are relative to the current buffer address, the disp registers are relative to src registers, the base register defines the start address of the reverb buffer (the end address is fixed, at 7FFFEh). Writing a value to mBASE does additionally set the current buffer address to that value.

1F801D98h - Voice 0..23 Reverb mode aka Echo On (EON) (R/W)

Sets reverb for the channel. As soon as the sample ends, the reverb for that channel is turned off... that's fine, but WHEN does it end?
In Reverb mode, the voice seems to output BOTH normal (immediately) AND via Reverb (delayed).

Reverb Bits in SPUCNT Register (R/W)

The SPUCNT register contains a Reverb Master Enable flag, and Reverb Enable flags for External Audio input and CD Audio input.
When the Reverb Master Enable flag is cleared, the SPU stops to write any data to the Reverb buffer (that is useful when zero-filling the reverb buffer; ensuring that already-zero values aren't overwritten by still-nonzero values).
However, the Reverb Master Enable flag does not disable output from Reverb buffer to the speakers (that might be useful to output uncompressed 22050Hz samples) (otherwise, to disable the buffer output, set the Reverb Output volume to zero and/or zerofill the reverb buffer).

7.11 SPU Reverb Formula

Reverb Formula

Notes

The values written to memory are saturated to -8000h..+7FFFh.
The multiplication results are divided by +8000h, to fit them to 16bit range.
All memory addresses are relative to the current BufferAddress, and wrapped within mBASE..7FFFEh when exceeding that region.
All data in the Reverb buffer consists of signed 16bit samples. The Left and Right Reverb Buffer addresses should be choosen so that one half of the buffer contains Left samples, and the other half Right samples (ie. the data is L,L,L,L,... R,R,R,R,...; it is NOT interlaced like L,R,L,R,...), during operation, when the buffer address increases, the Left half will overwrite the older samples of the Right half, and vice-versa.
The reverb hardware spends one 44100h cycle on left calculations, and the next 44100h cycle on right calculations (unlike as shown in the above formula, where left/right are shown simultaneously at 22050Hz).

Reverb Disable

SPUCNT.bit7 disables writes to reverb buffer, but reads from reverb buffer do still occur. If vAPF2 is zero then it does simply read "Lout=[mLAPF2-dAPF2]" and "Rout=[mRAPF2-dAPF2]". If vAPF2 is nonzero then it does additionally use data from APF1, if vAPF1 and vAPF2 are both nonzero then it's also using data from COMB. However, the SAME/DIFF stages aren't used when reverb is disabled.

Bug

vIIR works only in range -7FFFh..+7FFFh. When set to -8000h, the multiplication by -8000h is still done correctly, but, the final result (the value written to memory) gets negated (this is a pretty strange feature, it is NOT a simple overflow bug, it does affect the "+[mLSAME-2]" addition; although that part normally shouldn't be affected by the "*vIIR" multiplication). Similar effects might (?) occur on some other volume registers when they are set to -8000h.

Speed of Sound

The speed of sound is circa 340 meters per second (in dry air, at room temperature). For example, a voice that travels to a wall at 17 meters distance, and back to its origin, should have a delay of 0.1 seconds.

7.12 SPU Reverb Examples

Reverb Examples

Below are some Reverb examples, showing the required memory size (ie. set Port 1F801DA2h to "(80000h-size)/8"), and the Reverb register settings for Port 1F801DC0h..1F801DFFh, ie. arranged like so:

Also, don't forget to initialize Port 1F801D84h, 1F801D86h, 1F801D98h, and SPUCNT, and to zerofill the Reverb Buffer (so that no garbage values are output when activating reverb). For whatever reason, one MUST also initialize Port 1F801DACh (otherwise reverb stays off).

Room (size=26C0h bytes)

Studio Small (size=1F40h bytes)

Studio Medium (size=4840h bytes)

Studio Large (size=6FE0h bytes)

Hall (size=ADE0h bytes)

Half Echo (size=3C00h bytes)

Space Echo (size=F6C0h bytes)

Chaos Echo (almost infinite) (size=18040h bytes)

Delay (one-shot echo) (size=18040h bytes)

Reverb off (size=10h dummy bytes)

Note that the memory offsets should be 0001h here (not 0000h), otherwise zerofilling the reverb buffer seems to fail (maybe because zero memory offsets somehow cause the fill-value to mixed with the old value or so; that appears even when reverb master enable is zero). Also, when not using reverb, Port 1F801D84h, 1F801D86h, 1F801D98h, and the SPUCNT reverb bits should be set to zero.

7.13 SPU Unknown Registers

1F801DA0h - Some kind of a read-only status register.. or just garbage..?

Usually 9D78h, occassionaly changes to 17DAh or 108Eh for a short moment.
Other day: Usually 9CF8h, or occassionally 9CFAh.
Another day: Usually 0000h, or occassionally 4000h.

1F801DBCh - 4 bytes - Unknown? (R/W)

Other day (dots = same as above):

1F801E60h - 32 bytes - Unknown? (R/W)

Other day (dots = same as above):

The bytes at 1F801DBCh and 1F801E60h usually have the above values on cold-boot. The registers are read/write-able, although writing any values to them doesn't seem to have any effect on sound output. Also, the SPU doesn't seem to modify the registers at any time during sound output, nor reverb calculations, nor activated external audio input... the registers seem to be just some kind of general-purpose RAM.

7.14 SPU Internal State Machine from SPU RAM Timing

7.14.1 Introduction

The 33.8 Mhz clock of the PSX is a well chosen value. It is exactly 768 x 44.1 Khz = For each audio sample in CD quality, there are 768 cycles of system clock. So, the state machine has to repeat its complete cycle every 768 system clock cycles.

Now the full job to do within those 768 cycles: - 24 channels to process. - Reverb to compute and write back. - Write back to voice 1 / 3, audio CD L/R. - Do transfer from/to CPU bus of SPU RAM data if asked.

7.14.2 First look at the data from logic analyzer.

By looking at the signal of the SPU RAM chip, it is possible to figure out what it is reading and writing. - A read or a write to the SPU Ram is happening in 8 clock cycles. (Did not check in detail, but probably allow refresh and everything) - Each channel is using 24 cycles. (3 operations of 8 cycles) - Has TWO read for the current ADPCM block : one to the header of the currently played ADPCM block, one to the current 16 bit of the ADPCM. - A unrelated READ (see later) - 8 Cycle for each operation : WRITE CD Left, WRITE CD Right, Voice 1 WRITE, Voice 3 WRITE. - Reverb operations : 14 memory operations of 8 cycles.

7.14.3 Sequence of work

When doing the analysis from data, it is possible to figure out what are the operations, in what order they are done. But it is not possible to figure out what is the FIRST operation in the loop. So we arbitrarely decide to start the loop at 'Voice 1' (voice being from 1 to 24).

- Voice 1
- Write CD Left
- Write CD Right
- Write Voice 1
- Write Voice 3
- Reverb
- Voice 2
- Voice 3
- Voice 4
- ...
- ...
- Voice 23
- Voice 24

As written earlier, each Voice is 3x RAM access (one unrelated), reverb is 14x RAM access, then 4x RAM access for the all write.

7.14.4 What we can guess from those information.

- If system wants to keep reverb done in the end, and write in sync against Voice 1 and 3, then the loop would most likely start at Voice 2.
- ADPCM decoder has to keep ADPCM decoder internal state about the samples. As the algorithm depends on the previous value inside a block, it can't do a direct access to a given sample in the block.
- We also understand how reverb is using 22 Khz because of the lack of bandwidth to do everything in 768 cycles if done at 44.1 Khz.
- Even when voices are not active, they always read something. It is possible to guess that the sample is simply ignored at some point in the data path (volume set to zero internally or mux not selecting the value). Interestingly, it may be possible if garbage is introduced in those read, to know how it is cancelled (enabling suddenly the channel and reading the sample out of the channel 1 or 3) -> DSP keeps history of sample for Gaussian Interpolation.

7.14.5 Reverb Computation Order

We anticipate that the easiest way in hardware to disable/enable the REVERB function would be to switch those WRITE into READ.

7.14.6 Voices

7.14.7 Notes

- Remaining cycles.
- With 24x8 + 4x8 + 14x8 = 720 cycles out of 768 cycles.
- That would mean 6 READ/WRITE should still be possible.
- UNRELATED READ in voices : probably used for transfer from [CPU->SPU RAM] or [SPU RAM->CPU]
- That would equate to a transfer performance of 24 x 2 byte x 44100 Khz = 2,116,800 bytes/sec
- The fixed READ timing would explain also why CPU can't read directly SPU RAM. As the SPU need to be the master to push the data.
- It only works with DMA waiting for the data to be sent.

Everything is not fully clear yet, testing of SPU with proper tests to validate/invalidate various assumption. Our finding are based on a logic analyzer log using the PSX boot sounds, knowing the values of the registers thanks to emulators.

8. Interrupts

1F801070h I_STAT - Interrupt status register (R=Status, W=Acknowledge)

1F801074h I_MASK - Interrupt mask register (R/W)

Status: Read I_STAT (0=No IRQ, 1=IRQ)
Acknowledge: Write I_STAT (0=Clear Bit, 1=No change)
Mask: Read/Write I_MASK (0=Disabled, 1=Enabled)

Secondary IRQ10 Controller (Port 1F802030h)

EXP2 DTL-H2000 I/O Ports

Interrupt Request / Execution

The interrupt request bits in I_STAT are edge-triggered, ie. the get set ONLY if the corresponding interrupt source changes from "false to true".
If one or more interrupts are requested and enabled, ie. if "(I_STAT AND I_MASK)=nonzero", then cop0r13.bit10 gets set, and when cop0r12.bit10 and cop0r12.bit0 are set, too, then the interrupt gets executed.

Interrupt Acknowledge

To acknowledge an interrupt, write a "0" to the corresponding bit in I_STAT. Most interrupts (except IRQ0,4,5,6) must be additionally acknowledged at the I/O port that has caused them (eg. JOY_CTRL.bit4).

Observe that the I_STAT bits are edge-triggered (they get set only on High-to-Low, or False-to-True edges). The correct acknowledge order is:

When doing it vice-versa, the hardware may miss further IRQs (eg. when first setting JOY_CTRL.4=1, then a new IRQ may occur in JOY_STAT.4 within a single clock cycle, thereafter, setting I_STAT.7=0 would successfully reset I_STAT.7, but, since JOY_STAT.4 is already set, there'll be no further edge, so I_STAT.7 won't be ever set in future).

COP0 Interrupt Handling

Relevant COP0 registers are cop0r13 (CAUSE, reason flags), and cop0r12 (SR, control flags), and cop0r14 (EPC, return address), and, cop0cmd=10h (aka RFE opcode) is used to prepare the return from interrupts. For more info, see
COP0 - Exception Handling

PSX specific COP0 Notes

COP0 has six hardware interrupt bits, of which, the PSX uses only cop0r13.bit10 (the other ones, cop0r13.bit11-15 are always zero). cop0r13.bit10 is NOT a latch, ie. it gets automatically cleared as soon as "(I_STAT AND I_MASK)=zero", so there's no need to do an acknowledge at the cop0 side. COP0 additionally has two software interrupt bits, cop0r13.bit8-9, which do exist in the PSX, too, these bits are read/write-able latches which can be set/cleared manually to request/acknowledge exceptions by software.

Halt Function (Wait for Interrupt)

The PSX doesn't have a HALT opcode, so, even if the program is merely waiting for an interrupt to occur, the CPU is always running at full speed, which is resulting in high power consumption, and, in case of emulators, high CPU emulation load. To save energy, and to make emulation smoother on slower computers, I've added a Halt function for use in emulators:
EXP2 Nocash Emulation Expansion

9. DMA Channels

DMA Register Summary

These ports control DMA at the CPU-side. In most cases, you'll additionally need to initialize an address (and transfer direction, transfer enabled, etc.) at the remote-side (eg. at the GPU-side for DMA2).

1F801080h+N*10h - D#_MADR - DMA base address (Channel 0..6) (R/W)

In SyncMode=0, the hardware doesn't update the MADR registers (it will contain the start address even during and after the transfer) (unless Chopping is enabled, in that case it does update MADR, same does probably also happen when getting interrupted by a higher priority DMA channel).
In SyncMode=1 and SyncMode=2, the hardware does update MADR (it will contain the start address of the currently transferred block; at transfer end, it'll hold the end-address in SyncMode=1, or the 00FFFFFFh end-code in SyncMode=2)
Note: Address bit0-1 are writeable, but any updated current/end addresses are word-aligned with bit0-1 forced to zero.

1F801084h+N*10h - D#_BCR - DMA Block Control (Channel 0..6) (R/W)

For SyncMode=0 (ie. for OTC and CDROM):

For SyncMode=1 (ie. for MDEC, SPU, and GPU-vram-data):

For SyncMode=2 (ie. for GPU-command-lists):

BC/BS/BA can be in range 0001h..FFFFh (or 0=10000h). For BS, take care not to set the blocksize larger than the buffer of the corresponding unit can hold. (GPU and SPU both have a 16-word buffer). A larger blocksize means faster transfer.
SyncMode=1 decrements BA to zero, SyncMode=0 with chopping enabled decrements BC to zero (aside from that two cases, D#_BCR isn't changed during/after transfer).

1F801088h+N*10h - D#_CHCR - DMA Channel Control (Channel 0..6) (R/W)

The Start/Trigger bit is automatically cleared upon BEGIN of the transfer, this bit needs to be set only in SyncMode=0 (setting it in other SyncModes would force the first block to be transferred instantly without DRQ, which isn't desired).
The Start/Busy bit is automatically cleared upon COMPLETION of the transfer, this bit must be always set for all SyncModes when starting a transfer.
For DMA6/OTC there are some restrictions, D6_CHCR has only three read/write-able

bits: Bit24,28,30. All other bits are read-only: Bit1 is always 1 (step=backward), and the other bits are always 0.

1F8010F0h - DPCR - DMA Control Register (R/W)

Initial value on reset is 07654321h. If two or more channels have the same priority setting, then the priority is determined by the channel number (DMA0=Lowest, DMA6=Highest).

1F8010F4h - DICR - DMA Interrupt Register (R/W)

IRQ flags in Bit(24+n) are set upon DMAn completion - but caution - they are set ONLY if enabled in Bit(16+n).
Bit31 is a simple readonly flag that follows the following rules:

Upon 0-to-1 transition of Bit31, the IRQ3 flag (in Port 1F801070h) gets set.
Bit24-30 are acknowledged (reset to zero) when writing a "1" to that bits (and, additionally, IRQ3 must be acknowledged via Port 1F801070h).

1F8010F8h (usually 7FFAC68Bh? or 0BFAC688h)

1F8010FCh (usually 00FFFFF7h) (...maybe OTC fill-value)

Contains strange read-only values (but not the usual "Garbage").
Not yet tested during transfer, might be remaining length and address?

Commonly used DMA Control Register values for starting DMA transfers

XXX: DMA2 values 01000201h (VramWrite), 01000401h (List) aren't 100% confirmed to be used by ALL existing games. All other values are always used as listed above.

DMA Transfer Rates

MDEC decompression time is still unknown (may vary on RLE and color/mono).
GPU polygon rendering time is unknown (may be quite slow for large polys).
GPU vram read/write time is unknown (may vary on horizontal screen resolution).
CDROM BIOS default is 24 clks, for some reason most games change it to 40 clks.
SPU transfer is unknown (may have some extra delays).
XXX is SPU really only 4 clks (theoretically SPU access should be slower)?
PIO isn't used by any games (and if used: could be configured to other rates)

OTC is just writing to RAM without extra overload.
CDROM/SPU/PIO timings can be configured via Memory Control registers.

DRAM Hyper Page mode

DMA is using DRAM Hyper Page mode, allowing it to access DRAM rows at 1 clock cycle per word (effectively around 17 clks per 16 words, due to required row address loading, probably plus some further minimal overload due to refresh cycles). This is making DMA much faster than CPU memory accesses (CPU DRAM access takes 1 opcode cycle plus 6 waitstates, ie. 7 cycles in total)

CPU Operation during DMA

Basically, the CPU is stopped during DMA (theoretically, the CPU could be kept running when accessing only cache, scratchpad and on-chip I/O ports like DMA registers, and during the CDROM/SPU/PIO waitstates it could even access Main RAM, but these situations aren't supported).
However, the CPU operation resumes during periods when DMA gets interrupted (ie. after SyncMode 1 blocks, after SyncMode 2 list entries) (or in SyncMode 0 with Chopping enabled).

10. Timers

1F801100h+N*10h - Timer 0..2 Current Counter Value (R/W)

This register is automatically incrementing. It is write-able (allowing to set it to any value). It gets forcefully reset to 0000h on any write to the Counter Mode register, and on counter overflow (either when exceeding FFFFh, or when exceeding the selected target value).

1F801104h+N*10h - Timer 0..2 Counter Mode (R/W)

In one-shot mode, the IRQ is pulsed/toggled only once (one-shot mode doesn't stop the counter, it just suppresses any further IRQs until a new write to the Mode register occurs; if both IRQ conditions are enabled in Bit4-5, then one-shot mode triggers only

one of those conditions; whichever occurs first).
Normally, Pulse mode should be used (Bit10 is permanently set, except for a few clock cycles when an IRQ occurs). In Toggle mode, Bit10 is set after writing to the Mode register, and becomes inverted on each IRQ (in one-shot mode, it remains zero after the IRQ) (in repeat mode it inverts Bit10 on each IRQ, so IRQ4/5/6 are triggered only each 2nd time, ie. when Bit10 changes from 1 to 0).
The "free run" mode is simply saying that the counter will not reset at a given threshold value.

1F801108h+N*10h - Timer 0..2 Counter Target Value (R/W)

When the Target flag is set (Bit3 of the Control register), the counter increments up to (including) the selected target value, and does then restart at 0000h.

Dotclock/Hblank

For more info on dotclock and hblank timings, see:
GPU Timings
Caution: Reading the Current Counter Value can be a little unstable (when using dotclk or hblank as clock source); the GPU clock isn't in sync with the CPU clock, so the timer may get changed during the CPU read cycle. As a workaround: repeat reading the timer until the received value is the same (or slightly bigger) than the previous value.

11. CDROM Drive

Playstation CDROM I/O Ports

Playstation CDROM Commands

General CDROM Disk Format

Playstation CDROM Protection

General CDROM Disk Images

Playstation CDROM Coprocessor

11.1 CDROM Controller I/O Ports

	1F801800h	1F801801h	1F801802h	1F801803h
0	Index/Status (RW)	Response FIFO (R) (Mirror) Command Register (W)	Data FIFO (R) Parameter FIFO (W)	Interrupt Enable Register (R) Request Register (W)
1	Index/Status (RW)	Response FIFO (R) Sound Map Data Out (W)	Data FIFO (R) Interrupt Enable Register (W)	Interrupt Flag Register (RW)
2	Index/Status (RW)	Response FIFO (R) (Mirror) Sound Map Coding Info (W)	Data FIFO (R) Left-CD to Left-SPU Volume (W)	Interrupt Enable Register (R) (Mirror) Left-CD to Right-SPU Volume (W)
3	Index/Status (RW)	Response FIFO (R) (Mirror) Right-CD to Right-SPU Volume (W)	Data FIFO (R) Right-CD to Left-SPU Volume (W)	Interrupt Flag Register (R) (Mirror) Audio Volume Apply Changes (W)

1F801800h - Index/Status Register (Bit0-1 R/W) (Bit2-7 Read Only)

Bit3,4,5 are bound to 5bit counters; ie. the bits become true at specified amount of reads/writes, and thereafter once on every further 32 reads/writes.

1F801801h.Index0 - Command Register (W)

Writing to this address sends the command byte to the CDROM controller, which will then read-out any Parameter byte(s) which have been previously stored in the Parameter Fifo. It takes a while until the command/parameters are transferred to the controller, and until the response bytes are received; once when completed, interrupt INT3 is generated (or INT5 in case of invalid command/parameter values), and the response (or error code) can be then read from the Response Fifo. Some commands additionally have a second response, which is sent with another interrupt.

1F801802h.Index0 - Parameter Fifo (W)

Before sending a command, write any parameter byte(s) to this address.

1F801803h.Index0 - Request Register (W)

1F801802h.Index0..3 - Data Fifo - 8bit/16bit (R)

After ReadS/ReadN commands have generated INT1, software must set the Want Data bit (1F801803h.Index0.Bit7), then wait until Data Fifo becomes not empty (1F801800h.Bit6), the datablock (disk sector) can be then read from this register.

The PSX hardware allows to read 800h-byte or 924h-byte sectors, indexed as [000h..7FFh] or [000h..923h], when trying to read further bytes, then the PSX will repeat the byte at index [800h-8] or [924h-4] as padding value.
Port 1F801802h can be accessed with 8bit or 16bit reads (ie. to read a 2048-byte sector, one can use 2048 load-byte opcodes, or 1024 load halfword opcodes, or, more conventionally, a 512 word DMA transfer; the actual CDROM databus is only 8bits wide, so CPU/DMA are apparently breaking 16bit/32bit reads into multiple 8bit reads from 1F801802h).

1F801801h.Index1 - Response Fifo (R)

1F801801h.Index0,2,3 - Response Fifo (R) (Mirrors)

The response Fifo is a 16-byte buffer, most or all responses are less than 16 bytes, after reading the last used byte (or before reading anything when the response is 0-byte long), Bit5 of the Index/Status register becomes zero to indicate that the last byte was received.
When reading further bytes: The buffer is padded with 00h's to the end of the 16-bytes, and does then restart at the first response byte (that, without receiving a new response, so it'll always return the same 16 bytes, until a new command/response has been sent/received).

1F801802h.Index1 - Interrupt Enable Register (W)

1F801803h.Index0 - Interrupt Enable Register (R)

1F801803h.Index2 - Interrupt Enable Register (R) (Mirror)

XXX WRITE: bit5-7 unused should be 0 // READ: bit5-7 unused

1F801803h.Index1 - Interrupt Flag Register (R/W)

1F801803h.Index3 - Interrupt Flag Register (R) (Mirror)

Writing "1" bits to bit0-4 resets the corresponding IRQ flags; normally one should write 07h to reset the response bits, or 1Fh to reset all IRQ bits. Writing values like 01h is possible (eg. that would change INT3 to INT2, but doing that would be total nonsense). After acknowledge, the Response Fifo is made empty, and if there's been a pending command, then that command gets send to the controller.
The lower 3bit indicate the type of response received,

The other 2bit indicate something else,

The response interrupts are queued, for example, if the 1st response is INT3, and the second INT5, then INT3 is delivered first, and INT5 is not delivered until INT3 is acknowledged (ie. the response interrupts are NOT ORed together to produce INT7 or so). The upper bits however can be ORed with the lower bits (ie. Command Start INT10h and 1st Response INT3 would give INT13h).

1F801802h.Index2 - Audio Volume for Left-CD-Out to Left-SPU-Input (W)

1F801803h.Index2 - Audio Volume for Left-CD-Out to Right-SPU-Input (W)

1F801801h.Index3 - Audio Volume for Right-CD-Out to Right-SPU-Input (W)

1F801802h.Index3 - Audio Volume for Right-CD-Out to Left-SPU-Input (W)

Allows to configure the CD for mono/stereo output (eg. values "80h,0,80h,0" produce normal stereo volume, values "40h,40h,40h,40h" produce mono output of equivalent volume).
When using bigger values, the hardware does have some incomplete saturation support; the saturation works up to double volume (eg. overflows that occur on "FFh,0,FFh,0" or "80h,80h,80h,80h" are clipped to min/max levels), however, the saturation does NOT work properly when exceeding double volume (eg. mono with quad-volume "FFh,FFh,FFh,FFh").

After changing these registers, write 20h to 1F801803h.Index3.
Unknown if any existing games are actually supporting mono output. Resident Evil 2 uses these ports to produce fade-in/fade-out effects (although, for that purpose, it should be much easier to use Port 1F801DB0h).

1F801803h.Index3 - Audio Volume Apply Changes (by writing bit5=1)

1F801801h.Index1 - Sound Map Data Out (W)

This register seems to be restricted to 8bit bus, unknown if/how the PSX DMA controller can write to it (it might support only 16bit data for CDROM).

1F801801h.Index2 - Sound Map Coding Info (W)

Command Execution

Command/Parameter transmission is indicated by bit7 of 1F801800h.
When that bit gets zero, the response can be read immediately (immediately for MOST commands, but not ALL commands; so better wait for the IRQ).
Alternately, you can wait for an IRQ (which seems to take place MUCH later), and then read the response.
If there are any pending cdrom interrupts, these MUST be acknowledged before sending the command (otherwise bit7 of 1F801800h will stay set forever).

Command Busy Flag - 1F801800h.Bit7

Indicates ready-to-send-new-command,

Trying to send a new command in the Busy-phase causes malfunction (the older command will be dropped, the newer command executes and returns its results and triggers an interrupt).
When the Busy-flag goes off, a new command can be send immediately (even if the response from the previous command wasn't received yet), however, the new command stays in the Busy-phase until the IRQ from the previous command is acknowledged, at that point the actual transmission of the new command starts, and the Busy-flag goes off (once when the transmission completes).

Will not drop any of the two commands, thus execute sequentially.

Will drop the second response of Stop(), and then execute the next command.

Misc

Trying to do a 32bit read from 1F801800h returns the 8bit value at 1F801800h multiplied by 01010101h.

To init the CD

Seek-Busy Phase

Warning: most or all of the info in the sentence below appear to incorrect (either that, or I didn't understand that rather confusing sentence).
REPORTEDLY:
"You should not send some commands while the CD is seeking (ie. Getstat returns with bit6 set). Thing is that stat only gets updated after a new command. I haven't tested this for other command, but for the play command (03h) you can just keep repeating the [which?] command and checking stat returned by that, for bit6 to go low (and bit7 to go high in this case). If you don't and try to do a getloc [GetlocP and/or GetlocL?] directly after the play command reports it's done [what done? meaning sending start-to-play was "done"? or meaning play reached end-of-disc?], the CD will stop. (I guess the CD can't get it's current location while it's seeking, so the logic stops the seek to get an exact fix, but never restarts..)"

Sound Map Flowchart

Sound Map mode allows to output XA-ADPCM from Main RAM (rather than from CDROM).

Sound Map mode may be very useful for testing XA-ADPCM directly from within an exe file (without needing a cdrom with ADPCM sectors). And, Sound Map supports both 4bit and 8bit compression (the SPU supports only 4bit).
Caution: If ADPCM wasn't playing, and one sends one 900h-byte block, then it will get stored in one of three 900h-byte slots in SRAM, and one would expect that that slot to be played when the ADPCM output starts - however, actually, the hardware will more or less randomly play one of the three slots; not necessarily the slot that was updated most recently.

11.2 CDROM Controller Command Summary

Command Summary

E = Error 80h appears on some commands (02h..09h, 0Bh..0Dh, 10h..16h, 1Ah, 1Bh?, and 1Dh) when the disk is missing, or when the drive unit is disconnected from the mainboard.

sub_function numbers (for command 19h)

Test commands are invoked with command number 19h, followed by a sub_function number as first parameter byte. The Kernel seems to be using only sub_function 20h (to detect the CDROM Controller version).

* sub_functions 06h..08h, 30h..31h, and 4xh are supported only in vC0 and vC1.
** sub_function 51h is supported only in BIOS version vC2 and up.
*** sub_functions 22h..25h, 71h..76h supported only in BIOS version vC1 and up.

Unsupported GetQ,VCD,SecretUnlock (command 1Dh,1Fh,5xh)

INT5 will be returned if the command is unsupported. That, WITHOUT removing the Parameters from the FIFO, so the parameters will be accidently passed to the NEXT command. To avoid that: clear the parameter FIFO via [1F801803h.Index1]=40h after receiving the INT5 error.

11.3 CDROM - Control Commands

Sync - Command 00h --> INTx(stat+1,40h) (?)

Reportedly "command does not succeed until all other commands complete. This can be used for synchronization - hence the name."
Uh, actually, returns error code 40h = Invalid Command...?

Setfilter - Command 0Dh,file,channel --> INT3(stat)

Automatic ADPCM (CD-ROM XA) filter ignores sectors except those which have the same channel and file numbers in their subheader. This is the mechanism used to select which of multiple songs in a single .XA file to play.
Setfilter does not affect actual reading (sector reads still occur for all sectors).
XXX err... that is... does not affect reading of non-ADPCM sectors (normal "data" sectors are kept received regardless of Setfilter).

Setmode - Command 0Eh,mode --> INT3(stat)

The "Ignore Bit" does reportedly force a sector size of 2328 bytes (918h), however, that doesn't seem to be true. Instead, Bit4 seems to cause the controller to ignore the sector size in Bit5 (instead, the size is kept from the most recent Setmode command which didn't have Bit4 set). Also, Bit4 seems to cause the controller to ignore the \<exact> Setloc position (instead, data is randomly returned from the "Setloc position minus 0..3 sectors"). And, Bit4 causes INT1 to return status.Bit3=set (IdError). Purpose of Bit4 is unknown?

Init - Command 0Ah --> INT3(stat) --> INT2(stat)

Multiple effects at once. Sets mode=00h (or not ALL bits cleared?), activates drive motor, Standby, abort all commands.

Reset - Command 1Ch,(...) --> INT3(stat) --> Delay(1/8 seconds)

Resets the drive controller, reportedly, same as opening and closing the drive door. The command executes no matter if/how many parameters are used (tested with 0..7 params). INT3 indicates that the command was started, but there's no INT that would indicate when the command is finished, so, before sending any further commands, a delay of 1/8 seconds (or 400000h clock cycles) must be issued by software.
Note: Executing the command produces a click sound in the drive mechanics, maybe it's just a rapid motor on/off, but it might something more serious, like ignoring the /POS0 signal...?

MotorOn - Command 07h --> INT3(stat) --> INT2(stat)

Activates the drive motor, works ONLY if the motor was off (otherwise fails with INT5(stat,20h); that error code would normally indicate "wrong number of parameters", but means "motor already on" in this case).
Commands like Read, Seek, and Play are automatically starting the Motor when needed (which makes the MotorOn command rather useless, and it's rarely used by any games).
Myth: Older homebrew docs are referring to MotorOn as "Standby", claiming that it would work similar as "Pause", that is wrong: the command does NOT pause anything (if the motor is on, then it does simply trigger INT5, but without pausing reading or playing).
Note: The game "Nightmare Creatures 2" does actually attempt to use MotorOn to "pause" after reading files, but the hardware does simply ignore that attempt (aside from doing the INT5 thing).

Stop - Command 08h --> INT3(stat) --> INT2(stat)

Stops motor with magnetic brakes (stops within a second or so) (unlike power-off where it'd keep spinning for about 10 seconds), and moves the drive head to the begin of the first track. Official way to restart is command 0Ah, but almost any command will restart it.
The first response returns the current status (this already with bit5 cleared), the second response returns the new status (with bit1 cleared).

Pause - Command 09h --> INT3(stat) --> INT2(stat)

Aborts Reading and Playing, the motor is kept spinning, and the drive head maintains the current location within reasonable error.

The first response returns the current status (still with bit5 set if a Read command was active), the second response returns the new status (with bit5 cleared).

Data/ADPCM Sector Filtering/Delivery

The PSX CDROM BIOS is first trying to send sectors to the ADPCM decoder, and, if that didn't work out, then it's trying to send them to the main CPU (and if that didn't work out either, then it's silently ignoring the sector).

BUG: Note that the data delivery is done in two different attempts: The first one regardless of file/channel, and the second one only on matching file/channel (if filtering is enabled).

11.4 CDROM - Seek Commands

Setloc - Command 02h,amm,ass,asect --> INT3(stat)

Sets the seek target - but without yet starting the seek operation. The actual seek is invoked by certain commands: SeekL (Data) and SeekP (Audio) are doing plain seeks (and do Pause after completion). ReadN/ReadS are similar to SeekL (and do start reading data after the seek operation). Play is similar to SeekP (and does start playing audio after the seek operation).
The amm,ass,asect parameters refer to the entire disk (not to the current track). To seek to a specific location within a specific track, use GetTD to get the start address of the track, and add the desired time offset to it.

SeekL - Command 15h --> INT3(stat) --> INT2(stat)

Seek to Setloc's location in data mode (using data sector header position data, which works/exists only on Data tracks, not on CD-DA Audio tracks).
After the seek, the disk stays on the seeked location forever (namely: when seeking sector N, it does stay at around N-8..N-0 in single speed mode, or at around N-5..N+2 in double speed mode). This command will stop any current or pending ReadN or ReadS.
Trying to use SeekL on Audio CDs passes okay on the first response, but (after two seconds or so) the second response will return an error (stat+4,04h), and stop the drive motor... that error doesn't appear ALWAYS though... works in some situations... such like when previously reading data sectors or so...?

SeekP - Command 16h --> INT3(stat) --> INT2(stat)

Seek to Setloc's location in audio mode (using the Subchannel Q position data, which works on both Audio on Data disks).
After the seek, the disk stays on the seeked location forever (namely: when seeking sector N, it does stay at around N-9..N-1 in single speed mode, or at around N-2..N in double speed mode). This command will stop any current or pending ReadN or ReadS.
Note: Some older docs claim that SeekP would recurse only "MM:SS" of the "MM:SS:FF" position from Setloc - that is wrong, it does seek to MM:SS:FF (verified on a PSone).
After the seek, status is stat.bit7=0 (ie. audio playback off), until sending a new Play command (without parameters) to start playback at the seeked location.

SetSession - Command 12h,session --> INT3(stat) --> INT2(stat)

Seeks to session (ie. moves the drive head to the session, with stat bit6 set during the seek phase).
When issued during active-play, the command returns error code 80h.
When issued during play-spin-up, play is aborted.

after seek error --> disk stops spinning at 2nd response, then restarts spinning for 1 second or so, then stops spinning forever... and following gettn/gettd/getid/getlocl/

getlocp fail with error 80h...
The command does automatically read the TOC of the new session.
There seems to be no way to determine the current sessions number (via Getparam or so), and more important, no way to determine if the disk is a multi-session disk or not... except by trial... which would stop the drive motor on seek errors on single-session disks...?
For setloc, one must probably specifiy minutes within the 1st track of the new session (the 1st track of 1st session usually/always starts at 00:02:00, but for other sessions one would need to use GetTD)...?

11.5 CDROM - Read Commands

ReadN - Command 06h --> INT3(stat) --> INT1(stat) --> datablock

Read with retry. The command responds once with "stat,INT3", and then it's repeatedly sending "stat,INT1 --> datablock", that is continued even after a successful read has occured; use the Pause command to terminate the repeated INT1 responses.
Unknown which responses are sent in case of read errors?

====

ReadN and ReadS cause errors if you're trying to read an unlicensed CD or CD-R without a mod chip. Sectors on Audio CDs can be read only when CDDA is enabled via Setmode (otherwise error code 40h is returned).

====

Actually, Read seems to work on unlicensed CD-R's, but the returned data is the whole sector or so (the 2048 data bytes preceeded by a 12byte header, and probably/maybe followed by error-correction info; in fact the total received data in the Data Fifo is 4096 bytes; the last some bytes probably being garbage) (however error correction is NOT performed by hardware, so the 2048 data bytes may be trashy) (however, if the error correction info IS received, then error correction could be performed by software) (also Setloc doesn't seem to work accurately on unlicensed CD-R's).

====

After receiving INT1, the Kernel does,

and then,

then,

thereafter,

ReadS - Command 1Bh --> INT3(stat) --> INT1(stat) --> datablock

Read without automatic retry. Not sure what that means... does WHAT on errors? Maybe intended for continous streaming video output (to skip bad frames, rather than to interrupt the stream by performing read-retrys).

ReadN/ReadS

Both ReadN/ReadS are reading data sequentially, starting at the sector specified with Setloc, and then automatically reading the following sectors.

CDROM Incoming Data / Buffer Overrun Timings

The Read commands are continously receiving 75 sectors per second (or 150 sectors at double speed), and, basically, the software must be fast enough to process that amount of incoming data. However, the PSX hardware includes a buffer that can hold up to a handful (exact number is unknown?) of sectors, so, occasional delays of more than 1/75 seconds between processing two sectors aren't causing lost sectors, unless the delay(s) are summing up too much. The relevant steps for receiving data are:

The Data Request accepts the data for the currently pending interrupt, it should be usually issued between receiving/acknowledging INT1 (however, it can be also issued shortly after the acknowledge; even if there are further sectors in the buffer, there seems to be a small delay between the acknowledge and the next interrupt, and Data Requests during that period are still treated to belong to the old interrupt).
If a buffer overrun has occured \<before> issuing the Data Request, then wrong data will be received, ie. some sectors will be skipped (the hardware doesn't seem to support a buffer-overrun error flag? Anyways, see GetlocL description for a possible way to detect buffer-overruns).
If a buffer overrun occurs \<after> issuing the Data Request, then the requested data can be still read via I/O or DMA intactly, ie. the requested data is "locked", and the overrun will affect only the following sectors.

ReadTOC - Command 1Eh --> INT3(stat) --> INT2(stat)

Reread the Table of Contents of current session without reset. The command is rather slow, the second response appears after about 1 second delay. The command itself returns only status information (to get the actual TOC info, use GetTD and GetTN commands).
Note: The TOC contains information about the tracks on the disk (not file names or so, that kind of information is obtained via Read commands). The TOC is read automatically on power-up, when opening/closing the drive door, and when changing sessions (so, normally, it isn't required to use this command).

Setloc, Read, Pause

A normal CDROM access (such like reading a file) consists of three commands:

Normally one shouldn't mess up the ordering of those commands, but if one does, following rules do apply:
Setloc is memorizing the wanted target, and marks it as unprocessed, and has no other effect (it doesn't start reading or seeking, and doesn't interrupt or redirect any active reads).
If Read is issued with an unprocessed Setloc, then the drive is automatically seeking the Setloc location (and marks Setloc as processed).
If Read is issued without an unprocessed Setloc, the following happens: If reading is already in progress then it just continues reading. If Reading was Paused, then reading resumes at the most recently received sector (ie. returning that sector once another time).

11.6 CDROM - Status Commands

Status code (stat)

The 8bit status code is returned by Getstat command (and many other commands), the meaning of the separate stat bits is:

If the shell is closed, then bit4 is automatically reset to zero after reading stat with the Getstat command (most or all other commands do not reset that bit after reading). If stat bit0 or bit2 is set, then the normal respons(es) and interrupt(s) are not send, and, instead, INT5 occurs, and an error-byte is send as second response byte, with the following values:

80h appears on some commands (02h..09h, 0Bh..0Dh, 10h..16h, 1Ah, 1Bh?, and 1Dh) when the disk is missing, or when the drive unit is disconnected from the mainboard.

When the shell is opened, INT5 is triggered regardless of whether a command was executing or not. When this happens, all bits except shell open and error are cleared in the status register. The error byte in the INT5 is set to 08h.

Some games send a Stop command before changing discs, but others just wait for the user to open the shell, causing the disc to stop. The game can then send GetStat commands, looping until bit 4 is cleared to detect when the new disc has been inserted.

Stat Seek/Play/Read bits

There's is only max ONE of the three Seek/Play/Read bits set at a time, ie. during Seek, ONLY the seek bit is set (and Read or Play doesn't get until seek completion), that is important for Gran Turismo 1, which checks for seek completion by waiting for READ getting set (rather than waiting for SEEK getting cleared).

Getstat - Command 01h --> INT3(stat)

Returns stat (like many other commands), and additionally does reset the shell open flag (for the following commands; unless the shell is still opened). This is different as for most or all other commands (which may return stat, but which do not reset the shell open flag).
In other docs, the command is eventually referred to as "Nop", believing that it does nothing than returning stat (ignoring the fact that it's having the special shell open reset feature).

Getparam - Command 0Fh --> INT3(stat,mode,null,file,channel)

Returns stat (see Getstat above), mode (see Setmode), a null byte (always 00h), and file/channel filter values (see Setfilter).

GetlocL - Command 10h --> INT3(amm,ass,asect,mode,file,channel,sm,ci)

Retrieves 4-byte sector header, plus 4-byte subheader of the current sector. GetlocL can be send during active Read commands (but, mind that the GetlocL-INT3-response can't be received until any pending Read-INT1's are acknowledged).
The PSX hardware can buffer a handful of sectors, the INT1 handler receives the \<oldest> buffered sector, the GetlocL command returns the header and subheader of the \<newest> buffered sector. Note: If the returned \<newest> sector number is much bigger than the expected \<oldest> sector number, then it's likely that a buffer overrun has occured.
GetlocL fails (with error code 80h) when playing Audio CDs (or Audio Tracks on Data CDs). These errors occur because Audio sectors don't have any header/subheader (instead, equivalent data is stored in Subchannel Q, which can be read with GetlocP).
GetlocL also fails (with error code 80h) when the drive is in Seek phase (such like shortly after a new ReadN/ReadS command). In that case one can retry issuing GetlocL (until it passes okay, ie. until the seek has completed). During Seek, the drive seems to decode only Subchannel position data (but no header/subheader data), accordingly GetlocL won't work during seek (however, GetlocP does work during Seek).

GetlocP - Command 11h - INT3(track,index,mm,ss,sect,amm,ass,asect)

Retrieves 8 bytes of position information from Subchannel Q with ADR=1. Mainly intended for displaying the current audio position during Play. All results are in BCD.

Note: GetlocP is also used for reading the LibCrypt protection data:
CDROM Protection - LibCrypt

GetTN - Command 13h --> INT3(stat,first,last) ;BCD

Get first track number, and last track number in the TOC of the current Session. The number of tracks in the current session can be calculated as (last-first+1). The first track number is usually 01h in the first (or only) session, and "last track of previous session plus 1" in further sessions.

GetTD - Command 14h,track --> INT3(stat,mm,ss) ;BCD

For a disk with NN tracks, parameter values 01h..NNh return the start of the specified track, parameter value 00h returns the end of the last track, and parameter values bigger than NNh return error code 10h.
The GetTD values are relative to Index=1 and are rounded down to second boundaries (eg. if track=N Index=0 starts at 12:34:56, and Track=N Index=1 starts at 12:36:56, then GetTD(N) will return 12:36, ie. the sector number is truncated, and the Index=0 region is skipped).

GetQ - Command 1Dh,adr,point --> INT3(stat) --> INT2(10bytesSubQ,peak_lo)

Allows to read 10 bytes from Subchannel Q in Lead-In (see CDROM Subchannels chapter for details). Unlike GetTD, this command allows to receive the exact MM:SS:FF address of the point'ed Track (GetTD reads a memorized MM:SS value from RAM, whilst GetQ reads the full MM:SS:FF from the disk, which is slower than GetTD, due to the disk-access).
With ADR=1, point can be a any point number for ADR=1 in Lead-in (eg. 01h.. 99h=Track N, A2h=Lead-Out). The returned 10 bytes are raw SubQ data (starting with the ADR/Control value; of which the lower 4bits are always ADR=1).
The 11th returned byte is the Peak LSB (similar as in Play+Report, but in this case only the LSB is transferred, which is apparently a bug in CDROM BIOS, the programmer probably wanted to send 10 bytes without peak, or 12 bytes with full peak; although peak wouldn't be too useful, as it should always zero during Lead-In... but some discs do seem return non-zero values for whatever reason).
Aside from ADR=1, a value of ADR=5 can be used on multisession disks (eg. with point B0h, C0h). Not sure if any other ADR values can be used (ADR=3, ISRC is usually not in the Lead-In, ADR=2, EAN may be in the lead-in, but one may need to specify point equal to the first EAN byte).

If the ADR/Point combination isn't found, then a timeout occurs after circa 6 seconds (to avoid this, use GetTN to see which tracks/points exist). After the timeout, the command starts playing track 1. If the controller wasn't already in audio mode before sending the command, then it does switch off the drive motor for a moment (that, after the timeout, and before starting playback).
In case of timeout, the normal INT3/INT2 responses are replaced by INT3/INT5/INT5 (INT3 at command start, 1st INT5 at timeout/stop, and 2nd INT5 at restart/play).
Note: GetQ sends scratch noise to the SPU while seeking to the Lead-In area.

GetID - Command 1Ah --> INT3(stat) --> INT2/5 (stat,flags,type,atip,"SCEx")

The status byte (ie. the first byte in the responses), may differ in some cases; values shown above are typically received when issuing GetID shortly after power-up; however, shortly after the detect-busy phase, seek-busy flag (bit6) bit may be set, and, after issuing commands like Play/Read/Stop, bit7,6,5,1 may differ. The meaning of the separate 2nd response bytes is:

The fourth letter of the "SCEx" string contains region information: "SCEI" (Japan/NTSC), "SCEA" (America/NTSC), "SCEE" (Europe/PAL). The "SCEx" string is displayed in the intro, and the PSX refuses to boot if it doesn't match up for the local region.
With a modchip installed, the same response is sent for Mode1 and Audio disks (except

for Audio disks with very short TOCs (eg. singles) because SCEX reading is aborted immediately after reading all TOC entries on Audio disks); whether it is Audio or Mode1 can be checked by examining Subchannel Q ADR/Control.Bit6 (eg. via command 19h, 60h,50h,00h).
Yaroze does return "SCEA" for SCEA discs, but, for SCEI,SCEE,SCEW discs it does return four ASCII spaces (20h).

11.7 CDROM - CD Audio Commands

To play CD-DA Audio CDs, init the following SPU Registers: CD Audio Volume, Main Volume, and SPU Control Bit0. Then send Demute command, and Play command.

Mute - Command 0Bh --> INT3(stat)

Turn off audio streaming to SPU (affects both CD-DA and XA-ADPCM).
Even when muted, the CDROM controller is internally processing audio sectors (as seen in 1F801800h.Bit2, which works as usually for XA-ADPCM), muting is just forcing the CD output volume to zero.
Mute is used by Dino Crisis 1 to mute noise during modchip detection.

Demute - Command 0Ch --> INT3(stat)

Turn on audio streaming to SPU (affects both CD-DA and XA-ADPCM). The Demute command is needed only if one has formerly used the Mute command (by default, the PSX is demuted after power-up (...and/or after Init command?), and is demuted after cdrom-booting).

Play - Command 03h (,track) --> INT3(stat) --> optional INT1(report bytes)

Starts CD Audio Playback. The parameter is optional, if there's no parameter given (or if it is 00h), then play either starts at Setloc position (if there was a pending unprocessed Setloc), or otherwise starts at the current location (eg. the last point seeked, or the current location of the current song; if it was already playing). For a disk with N songs, Parameters 1..N are starting the selected track. Parameters N+1..99h are restarting the begin of current track. The motor is switched off automatically when Play reaches the end of the disk, and INT4(stat) is generated (with stat.bit7 cleared).
The track parameter seems to be ignored when sending Play shortly after power-up (ie. when the drive hasn't yet read the TOC).
===

"Play is almost identical to CdlReadS, believe it or not. The main difference is that this does not trigger a completed read IRQ. CdlPlay may be used on data sectors. However, all sectors from data tracks are treated as 00, so no sound is played. As CdlPlay is reading, the audio data appears in the sector buffer, but is not reliable. Game Shark "enhancement CDs" for the 2.x and 3.x versions used this to get around the PSX copy protection."
Hmmm, what/where is the sector buffer... in the SPU?
And, what/who are the 2.x and 3.x versions?

Forward - Command 04h --> INT3(stat) --> optional INT1(report bytes)

Backward - Command 05h --> INT3(stat) --> optional INT1(report bytes)

After sending the command, the drive is in fast forward/backward mode, skipping every some sectors. The skipping rate is fixed (it doesn't increase after some seconds) (however, it increases when (as long as) sending the command again and again). The sound becomes (obviously) non-continous, and also rather very silent, muffled, and almost inaudible (that's making it rather useless; unless it's combined with a track/minute/second display). To terminate forward/backward, send a new Play command (with no parameters, so play starts at the "searched" location). Backward automatically switches to Play when reaching the begin of Track 1. Forward automatically Stops the drive motor with INT4(stat) when reaching the end of the last track.
Forward/Backwards work only if the drive was in Play state, and only if Play had already started (ie. not shortly/immediately after a Play command); if the drive was not in Play state, then INT5(stat+1,80h) occurs.

Setmode bits used for Play command

During Play, only bit 7,2,1 of Setmode are used, all other Setmode bits are ignored (that, including bit0, ie. during Play the drive is always in CD-DA mode, regardless of that bit).
Bit7 (double speed) should be usually off, although it can be used for a fast forward effect (with audible output). Bit2 (report) activates an optional interrupt for Play, Forward, and Backward commands (see below). Bit1 (autopause) pauses play at the end of the track.

Report --> INT1(stat,track,index,mm/amm,ss+80h/ass,sect/asect,peaklo,peakhi)

With report enabled via Setmode, the Play, Forward, and Backward commands do repeatedly generate INT1 interrupts, with eight bytes response length. The interrupt isn't generated on ALL sectors, and the response changes between absolute time, and time within current track (the latter one indicated by bit7 of ss):

The last two response bytes (peaklo,peakhi) contain the Peak value, as received from the CXD2510Q Signal Processor. That is: An unsigned absolute peak level in lower 15bit, and an L/R flag in upper bit. The L/R bit is toggled after each SUBQ read, however the PSX Report mode does usually forward SUBQ only every 10 frames (but does read SUBQ in \<every> frame), so L/R will stay stuck in one setting (but may toggle after one second; ie. after 75 frames). And, peak is reset after each read, so 9 of the 10 frames are lost.
Note: Report mode affects only CD Audio (not Data, nor XA-ADPCM sectors).

AutoPause --> INT4(stat)

Autopause can be enabled/disabled via Setmode.bit1:

End of Track is determined by sensing a track number transition in SubQ position info. After autopause, the disc stays at the \<end> of the old track, NOT at the \<begin> of the next track (so trying to resume playing by sending a new Play command without new Seek/Setloc command will instantly pause again).
Caution: SubQ track transitions may pause instantly when accidently starting to play at the end of the previous track rather than at begin of desired track (this \<might> happen due to seek inaccuracies, for example, GetTD does round down TOC entries from MM:SS:FF to MM:SS:00, which may be off by 0.99 seconds, although this error should be usually compensated by the leading 2-second pregap/index0 region at the begin of each track, unfortunately there are a few .CUE sheet files that do lack both PREGAP and INDEX 00 entries on audio tracks, which might cause problems with autopause).
AutoPause is used by Rayman and Tactics Ogre.

Playing XA-ADPCM Sectors (compressed audio data)

Aside from normal uncompressed CD Audio disks, the PSX can also play XA-ADPCM compressed sectors. XA-ADPCM sectors are organized in Files (not in tracks), and are "played" with Read command (not Play command).
To play XA-ADPCM, initialize the SPU for CD Audio input (as described above), enable ADPCM via Setmode, then select the sector via Setloc, and issue a Read command (typically ReadS).
XA-ADPCM sectors are interleaved, ie. only each Nth sector should be played (where "N" depends on the Motor Speed, mono/stereo format, and sample rate). If the "other" sectors do contain XA-ADPCM data too, then the Setfilter command (and XA-Filter enable flag in Setmode) must be used to select the desired sectors. If the "other" sectors do contain code or data (eg. MDEC video data) which is wanted to be send to the CPU, then SetFilter isn't required to be enabled (although it shouldn't disturb reading even if it is enabled).
If XA-ADPCM (and/or XA-Filter) is enabled via Setmode, then INT1 is generated only for non-ADPCM sectors.
The Setmode sector-size selection is don't care for forwarding XA-ADPCM sectors to the SPU (the hardware does always decompress all 900h bytes).

11.8 CDROM - Test Commands

11.9 CDROM - Test Commands - Version, Switches, Region, Chipset, SCEx

19h,20h --> INT3(yy,mm,dd,ver)

Indicates the date (Year-month-day, in BCD format) and version of the HC05 CDROM controller BIOS. Known/existing values are:

19h,21h --> INT3(flags)

Returns the current status of the POS0 and DOOR switches. Bit0=HeadIsAtPos0, Bit1=DoorIsOpen, Bit2-7=AlwaysZero.

19h,22h --> INT3("for Europe")

Indicates the region that console is to be used in:

The CDROMs must contain a matching SCEx string accordingly.
The string "for Europe" does also suggest 50Hz PAL/SECAM video hardware.
The Yaroze accepts any normal SCEE,SCEA,SCEI discs, plus special SCEW discs.

19h,23h --> INT3("CXD2940Q/CXD1817Q/CXD2545Q/CXD1782BR") ;Servo Amplifier

19h,24h --> INT3("CXD2940Q/CXD1817Q/CXD2545Q/CXD2510Q") ;Signal Processor

19h,25h --> INT3("CXD2940Q/CXD1817Q/CXD1815Q/CXD1199BQ") ;Decoder/FIFO

Indicates the chipset that the CDROM controller is intended to be used with. The strings aren't always precisely correct (CXD1782BR is actually CXA1782BR, ie. CXA, not CXD) (and CXD1199BQ chips exist on PU-7 boards, but later PU-8 boards do actually use CXD1815Q) (and CXD1817Q is actually CXD1817R) (and newer PSones are using CXD2938Q or possibly CXD2941R chips, but nothing called CXD2940Q).
Note: Yaroze responds by CXD1815BQ instead of CXD1199BQ (but not by CXD1815Q).

19h,04h --> INT3(stat) ;Read SCEx string (and force motor on)

Resets the total/success counters to zero, and does then try to read the SCEx string from the current location (the SCEx is stored only in the Lead-In area, so, if the drive head is elsewhere, it will usually not find any strings, unless a modchip is permanently simulating SCEx strings).
This is a raw test command (the successful or unsuccessful results do not lock/unlock the disk). The results can be read with command 19h,05h (which will terminate the SCEx reading), or they can be read from RAM with command 19h,60h,lo,hi (which doesn't stop reading). Wait 1-2 seconds before expecting any results.
Note: Like 19h,00h, this command forces the drive motor to spin at standard speed (synchronized with the data on the disk), works even if the shell is open (but stops spinning after a while if the drive is empty).

19h,05h --> INT3(total,success) ;Get SCEx Counters

Returns the total number of "Sxxx" strings received (where at least the first byte did match), and the number of full "SCEx" strings (where all bytes did match). Typically, the values are "01h,01h" for Licensed PSX Data CDs, or "00h,00h" for disk missing, unlicensed data CDs, Audio CDs.

The counters are reset to zero, and SCEx receive mode is active for a few seconds after booting a new disk (on power up, on closing the drive door, on sending a Reset command, and on sub_function 04h). The disk is unlocked if the "success" counter is nonzero, the only exception is sub_function 04h which does update the counters, but does not lock/unlock the disk.

11.10 CDROM - Test Commands - Test Drive Mechanics

Signal Processor and Servo Amplifier

19h,50h,msb[,mid,[lsb[,xlo]]] --> INT3(stat)

Sends an 8bit/16bit/24bit command to the hardware, depending on number of parameters:

19h,51h,msb[,mid,[lsb]] --> INT3(stat,hi,lo) ;BIOS vC2/vC3 only

Supported by newer CDROM BIOSes only (such that use CXD2545Q or newer chips). Works same as 19h,50h, but does additionally receive a response.
The command is always sending a 24bit CX(Xxxxxx) command, but it doesn't verify the number of parameter bytes (when using more than 3 bytes: extra bytes are ignored, when using less than 3 bytes: garbage is appended, which is somewhat valid because 8bit/16bit commands can be padded to 24bit size by appending "don't care" bits).
The command can be used to send any CX(..) command, but actually it does make sense only for the get-status commands, see below "19h,51h,39h,xxh" description.

19h,51h,39h,xxh --> INT3(stat,hi,lo) ;BIOS vC2/vC3 only

Supported by newer CDROM BIOSes only (such that use CXD2545Q or newer chips). Sends CX(39xx) to the hardware, and receives a response (the response.hi byte is usually 00h for 8bit responses, or 00h..01h for 9bit responses). For example, this can be used to dump the Coefficient RAM.

19h,03h --> INT3(stat) ;force motor off

Forces the motor to stop spinning (ignored during spin-up phase).

19h,17h --> INT3(stat) ;force motor on, clockwise, super-fast

19h,01h --> INT3(stat) ;force motor on, anti-clockwise, super-fast

19h,02h --> INT3(stat) ;force motor on, anti-clockwise, super-fast

19h,10h --> INT3(stat) ;force motor on, anti-clockwise, super-fast

19h,18h --> INT3(stat) ;force motor on, anti-clockwise, super-fast

Forces the drive motor to spin at maximum speed (which is much faster than normal or double speed), in normal (clockwise), or reversed (anti-clockwise) direction. The commands work even if the shell is open. The commands do not try to synchronize the motor with the data on the disk (and do thus work even if no disk is inserted).

19h,00h --> INT3(stat) ;force motor on, clockwise (even if shell open)

This command seems to have effect only if the drive motor was off. If it was off, it does FFh-fills the TOC entries in RAM, and seek to the begin of the TOC at 98:30:00 or so (where minute=98 means minus two). From that location, it follows the spiral on the disk, although it does occassionally jump back some seconds. After clearing the TOC, the command does not write new data to the TOC buffer in RAM.
Note: Like 19h,04h, this command forces the drive motor to spin at standard speed (synchronized with the data on the disk), works even if the shell is open (but stops spinning after a while if the drive is empty).

19h,11h --> INT3(stat) ;Move Lens Up (leave parking position)

19h,12h --> INT3(stat) ;Move Lens Down (enter parking position)

19h,13h --> INT3(stat) ;Move Lens Outwards (away from center of disk)

19h,14h --> INT3(stat) ;Move Lens Inwards (towards center of disk)

Moves the laser lens. The inwards/outwards commands do move ONLY the lens (ie. unlike as for Seek commands, the overall-laser-unit remains in place, only the lens is moved).

19h,15h - if motor on: move head outwards + inwards + motor off

Moves the drive head to outer-most and inner-most position. Note that the drive doesn't have a switch that'd tell the controller when it has reached the outer-most position (so it'll forcefully hit against the outer edge) (ie. using this command too often may destroy the drive mechanics).
Note: The same destructive hit-outer-edge effect happens when using Setloc/Seek with too large values (like minute=99h).

19h,16h --> INT3(stat) ;Unknown / makes some noise if motor is on

19h,19h --> INT3(stat) ;Unknown / no effect

19h,1Ah --> INT3(stat) ;Unknown / makes some noise if motor is on

Seem to have no effect?
19h,16h seems to Move Lens Inwards, too.

19h,06h,new --> INT3(old) ;Adjust balance in RAM, and apply it via CX(30+n)

19h,07h,new --> INT3(old) ;Adjust gain in RAM, and apply it via CX(38+n)

19h,08h,new --> INT3(old) ;Adjust balance in RAM only

These commands are supported only by older CDROM BIOS versions (those with CXA1782BR Servo Amplifier).
Later BIOSes will respond with INT5(11h,20h) when trying to use these commands

(because CXD2545Q and later Servo Amplifiers don't support the CX(30/38+n) commands).

11.11 CDROM - Test Commands - Prototype Debug Transmission

Serial Debug Messages

Older CDROM BIOSes are supporting debug message transmission via serial bus, using lower 3bit of the HC05 "databus" combined with the so-called "ROMSEL" pin (which apparently doesn't refer to Read-Only-Memory, but rather something like Runtime-Output-Message, or whatever).
Data is transferred in 24bit units (8bit command/index from HC05, followed by 16bit data to/from HC05), bigger messages are divided into multiple such 24bit snippets.
There are no connectors for external debug hardware on any PSX mainboards, so the whole stuff seems to be dating back to prototypes. And it seems to be removed from later BIOSes (which appear to use "ROMSEL" as "SCLK"; for receiving status info from the new CXD2545Q chips).

19h,30h,index,dat1,dat2 --> INT3(stat) ;Prototype/Debug stuff

19h,31h,dat1,dat2 --> INT3(stat) ;Prototype/Debug stuff

19h,4xh,index --> INT3(dat1,dat2) ;Prototype/Debug stuff

These functions are supported on older CDROM BIOS only; later BIOSes respond by INT5(11h,10h).
The functions do not affect the CDROM operation (they do simple allow to transfer data between Main CPU and external debug hardware).
Sub functions 30h and 31h may fail with INT5(11h,80h) when receiving wrong signals on the serial input line.
Sub function "4xh" value can be 40h..4Fh (don't care).

INT5 Debug Messages

Alongsides to INT5 errors, the BIOS is usually also sending information via the above serial bus (the error info is divided into multiple 8bit+16bit snippets, and contains stat, error code, mode, current SubQ position, and most recently issued command).

11.12 CDROM - Test Commands - Read/Write Decoder RAM and I/O Ports

Caution: Below commands 19h,71h..76h are supported only in BIOS version vC1 and up. Not supported in vC0.

19h,71h,index --> INT3(databyte) ;Read single register

index can be 00h..1Fh, bigger values seem to be mirrored to "index AND 1Fh", with one exception: index 13h in NOT mirrored, instead, index 33h, 53h, 93h, B3h, D3h, F3h return INT5(stat+1,10h), and index 73h returns INT5(stat+1,20h).
Aside from returning a value, the commands seem to DO something (like moving the drive head when a disk is inserted). Return values are usually:

19h,72h,index,databyte --> INT3(stat) ;Write single register

19h,73h,index,len --> INT3(databytes...) ;Read multiple registers (bugged)

19h,74h,index,len,databytes --> INT3(stat) ;Write multiple registers (bugged)

Same as read/write single register, but trying to transfer multiple registers at once.
BUG: The transfer should range from 00h to len-1, but the loop counter is left uninitialized (set to X=48h aka "command number 19h-minus-1-mul-2" instead of X=00h). Causing to the function to read/write garbage at index 48h..FFh, it does then wrap to 00h and do the correct intended transfer, but the preceeding bugged part may have smashed RAM or I/O ports.

19h,75h --> INT3(remain.lo,remain.hi,addr.lo,addr.hi) ;Get Host Xfer Info

Returns a 4-byte value. In my early tests, on the first day it returned B1h,CEh,4Ch,01h, on the next day 2Ch,E4h,95h,D5h, and on all following days 00h,C0h,00h,00h (no idea why/where the earlier values came from).
The first byte seems to be always 00h; no matter of [1F0h].
The second byte seems to be always C0h; no matter of [1F1h].
The third,fourth bytes are [1F2h,1F3h].
That two bytes are 0Ch,08h after Read commands.

19h,76h,len_lo,len_hi,addr_lo,addr_hi --> INT3(stat) ;Prepare SRAM Transfer

Prepare Transfer to/from 32K SRAM.
After INT3, data can be read (same way as sector data after INT1).

11.13 CDROM - Test Commands - Read HC05 SUB-CPU RAM and I/O Ports

19h,60h,addr_lo,addr_hi --> INT3(data) ;Read one byte from Drive RAM or I/O

Reads one byte from the controller's RAM or I/O area, see the memory map below for more info. Among others, the command allows to read Subchannel Q data, eg. at [200h..209h], including ADR=2/UPC/EAN and ADR=3/ISRC values (which are suppressed by GetlocP). Eg. wait for ADR\<>2, then for ADR=2, then read the remaining 9 bytes (because of the delayed IRQs, this works only at single speed) (at double speed one can read only 5 bytes before the values get overwritten by new data). Unknown if older boards (with 4.00MHz oscillators) are fast enough to read all 10 SubQ bytes.

CDROM Controller I/O Area and RAM Memory Map

First 40h bytes are I/O ports (as in MC68HC05 datasheet):

Next 200h bytes are RAM:

Other/invalid addresses are:

Writing to RAM

There is no command for writing to RAM. Except that, one can write to the command/parameter buffer at 1E0h and up. Normally, the longest known command should have 6 bytes (19h,76h,a,b,c,d), and longer commands results in "bad number of parameters" response - however, despite of that error message, the controller does still store ALL parameter bytes in RAM (at address 1E1h..2E0h, then wrapping back to 1E1h). Whereas, writing more than 16 bytes (FIFO storage size) will mirror the FIFO content twice, and more than 32 bytes (FIFO counter size) will work only when feeding extra data into the FIFO during transmission. Anyways, writing to 1E1h and up doesn't allow to do interesting things (such like manipulating the stack and executing custom code on the CPU).

Subchannel Q Notes

The "adjusted position values" at 050h, 210h, 310h contain only position information (with ADR=1) (the PSX seems to check only the lower 2bit of the 4bit ADR value, so it also treats ADR=5 as ADR=1, too). Additionally, during Lead-In, bytes 7..9 are overwritten by the position value from bytes 3..5. The "real values" contain unadjusted data, including ADR=2 and ADR=3 etc.

11.14 CDROM - Secret Unlock Commands

SecretUnlockPart1 - Command 50h --> INT5(11h,40h)

SecretUnlockPart2 - Command 51h,"Licensed by" --> INT5(11h,40h)

SecretUnlockPart3 - Command 52h,"Sony" --> INT5(11h,40h)

SecretUnlockPart4 - Command 53h,"Computer" --> INT5(11h,40h)

SecretUnlockPart5 - Command 54h,"Entertainment" --> INT5(11h,40h)

SecretUnlockPart6 - Command 55h,\<region> --> INT5(11h,40h)

SecretUnlockPart7 - Command 56h --> INT5(11h,40h)

Sending these commands with the correct strings (in order 50h through 56h) does disable the "SCEx" protection. The region can be detected via test command 19h,22h, and must be translated to the following \<region> string:

In the unlocked state, ReadN/ReadS are working for unlicensed CD-Rs, and for imported CDROMs from other regions (both without needing modchips). However there are some cases which may still cause problems: The GetID command (1Ah) does still identify the disc as being unlicensed, same for the Get SCEx Counters test command (19h,05h). And, if a game should happen to send the Reset command (1Ch) for some weird reason, then the BIOS would forget the unlocking, same for games that set the "HCRISD" I/O port bit. On the contrary, opening/closing the drive door does not affect the unlocking

state.
The commands have been discovered in September 2013, and appear to be supported by all CDROM BIOS versions (from old PSXes up to later PSones).
Note that the commands do always respond with INT5 errors (even on successful unlocking).
Japanese consoles are internally containing code for processing the Secret Unlock commands, but they are not actually executing that code, and even if they would do so: they are ignoring the resulting unlocking flag, making the commands nonfunctional in Japan/Asia regions.

SecretLock - Command 57h --> INT5(11h,40h)

Undoes the unlocking and restores the normal locked state (same happens when sending the Unlocking commands in wrong order or with wrong parameters).

SecretCrash - Command 58h..5Fh --> Crash

Jumps to a data area and executes random code. Results are more or less unpredictable (as they involve executing undefined opcodes). Eventually the CPU might hit a RET opcode and recover from the crash.

11.15 CDROM - Video CD Commands

VideoCdSio - Cmd 1Fh,01h,JoyL,JoyH,State,Task,0 --> INT3(stat,req,mm,ss,ff,x)

The JoyL/JoyH bytes contain 16bit button (and drive door) bits:

The State byte can be:

The Task byte can be:

The req byte in the INT3 response can be:

VideoCdSwitch - Cmd 1Fh,02h,flag,x,x,x,x --> INT3(stat,0,0,x,x,x)

Some findings on the SC430924 firmware...

The version/date is "15 Aug 1996, version C2h", although the "C2h" is misleading: The firmware is nearly identical to version "C1h" from PU-8 boards (the stuff added in normal "C2h" versions would be for PU-18 boards with different cdrom chipset).

Compared to the original C1h version, there are only a few changes: A initialization function for initializing port F on power-up. And new command (command 1Fh, inserted in the various command tables), with two subfunctions (01h and 02h):
- Command 1Fh,01h,a,b,c,d,e --> INT3(stat,a,b,c,d,e) Serial 5-byte read-write
- Command 1Fh,02h,v,x,x,x,x --> INT3(stat,0,0,x,x,x) Toggle 1bit (port F.bit3)
Whereas,

The Port F bits are:

And that's about all. Ie. essentially, the only change is that the new command controls Port F. There is no interaction with the remaining firmware (ie. reading, seeking, and everything is working as usually, without any video-cd related changes).
The SCEx stuff is also not affected (ie. Video CDs would be seen as unlicensed discs, so the PSX couldn't read anything from those discs, aside from Sub-Q position data, of course). The SCEx region is SCEI aka "Japan" (or actually for Asia in this case).

Note

The SPU MUTE Flag (SPUCNT.14) does also affect VCD Audio (mute is applied to the final analog audio amplifier). All other SPUCNT bits can be zero for VCD.

11.16 CDROM - Mainloop/Responses

SUB-CPU Mainloop

The SUB-CPU is running a mainloop that is handling hardware events (by simple polling, not by IRQs):

There is no fixed priority: if both incoming sector and incoming command are present, then the SUB-CPU may handle either one, depending on which portion of the mainloop it is currently executing.
There is no fixed timing: if the mainloop is just checking for a specific event, then a new event may be processed immediately, otherwise it may take whole mainloop cycle until the SUB-CPU sees the event.
Whereas, the mainloop cycle execution time isn't constant: It may vary depending on various details. Especially, some maintenance stuff is only handled approximately around 15 times per second (so there are 15 slow mainloop cycles per second).

The order of steps that happen when sending a command to the CD controller look roughly like this:

Responses

The PSX can deliver one INT after another. Instead of using a real queue, it's merely using some flags that do indicate which INT(s) need to be delivered. Basically, there seem to be two flags: One for Second Response (INT2), and one for Data/Report Response (INT1). There is no flag for First Response (INT3); because that INT is generated immediately after executing a command.
The flag mechanism means that the SUB-CPU cannot hold more than one undelivered INT1. That, although the CDROM Decoder does notify the SUB-CPU about all newly received sectors, and it can hold up to eight sectors in the 32K SRAM. However, the

SUB-CPU BIOS merely sets a sector-delivery-needed flag (instead of memorizing which/how many sectors need to be delivered, and, accordingly, the PSX can use only three of the available eight SRAM slots: One for currently pending INT1, one for undelivered INT1, and one for currently/incompletly received sector).

First Response (INT3) (or INT5 if failed)

The first response is sent immediately after processing a command. In detail:
The mainloop checks for incoming commands once every some clock cycles, and executes commands under following condition:

Once when the command gets executed it will sent the first response immediately after the command execution (which may only take a few clock cycles, or some more cycles, for example Init/ReadTOC do include some time consuming initializations). Anyways, there will be no other INTs generated during command execution, so once when the command execution has started, it's guaranteed that the next INT will contain the first response.

Second Responses (INT2) (or INT5 if failed)

Some commands do send a second response after actual command execution:

In some cases (like seek or spin-up), it may take more than a second until the 2nd response is sent.
It should be highly recommended to WAIT until the second response is generated BEFORE sending a new command (it wouldn't make too much sense to send a new

command between first and second response, and results would be unknown, and probably totally unpredictable).
Error Notes: If the command has been rejected (INT5 sent as 1st response) then the 2nd response isn't sent (eg. on wrong number of parameters, or if disc missing). If the command fails at a later stage (INT5 as 2nd response), then there are cases where another INT5 occurs as 3rd response (eg. on SetSession=02h on non-multisession-disk).

Data/Report Responses (INT1)

11.17 CDROM - Response Timings

Here are some response timings, measured in 33MHz units on a PAL PSone. The CDROM BIOSes mainloop is doing some maintenance stuff once and when, meaning that the response time will be higher in such mainloop cycles (max values), and less in normal cycles (min values). The maintenance timings do also depend on whether the motor is on or off (and probably on various other factors like seeking).

First Response

The First Response interrupt is sent after processing the command (that is, when the mainloop sees a new command without any old interrupt pending). For GetStat, timings are as so:

Timings for most other commands should be similar as above. One exception is the Init command, which is doing some initialization before sending the 1st response:

The ReadTOC command is doing similar initialization, and should have similar timing as Init command. Some (rarely used) Test commands include things like serial data transfers, which may be also quite slow.

Second Response

Moreover, Seek/Play/Read/SetSession/MotorOn/Init/ReadTOC are sending second responses which depend on seek time (and spin-up time if the motor was off). The seek timings are still unknown, and they are probably quite complicated:
The CDROM BIOS seems to split seek distance somehow into coarse steps (eg. minutes) and fine steps (eg. seconds/sectors), so 1-minute seek distance may have completely different timings than 59-seconds distance.
The amount of data per spiral winding increases towards ends of the disc (so the drive head will need to be moved by shorter distance when moving from minute 59 to 60 as than moving from 00 to 01).
The CDROM BIOS contains some seek distance table, which is probably optimized for 72-minute discs (or whatever capacity is used on original PSX discs). 80-minute CDRs may have tighter spiral windings (the above seek table is probably causing the drive head to be moved too far on such discs, which will raise the seek time as the head needs to be moved backwards to compensate that error).

INT1 Rate

The INT1 rate needs to be precise for CD-DA and CD-XA Audio streaming, exact clock cycle values should be: SystemClock*930h/4/44100Hz for Single Speed (and half as much for Double Speed) (the "Average" values are AVERAGE values, not exact values).

11.18 CDROM - Response/Data Queueing

[Below are some older/outdated test cases]

Sector Buffer

The CDROM sector buffer is 32Kx8 SRAM (IC303). The buffer is apparently divided into 8 slots, theoretically allowing to buffer up to 8 sectors.
BUG: The drive controller seems to allow only 2 of those 8 sectors (the oldest sector, and the current/newest sector).
Ie. after processing the INT1 for the oldest sector, one would expect the controller to generate another INT1 for next newer sector - but instead it appears to jump directly to INT1 for the newest sector (skipping all other unprocessed sectors). There is no known way to get around that effect.
So far, the big 32Kbyte buffer is entirely useless (the two accessible sectors could have been as well stored in a 8Kbyte chip) (unless, maybe the 32Kbytes have been intended for some error-correction "read-ahead" purposes, rather than as "look-back" buffer for old sectors; one of the unused slots might be also used for XA-ADPCM sectors).
The bottom line is that one should process INT1's as soon as possible (ie. before the cdrom controller receives and skips further sectors). Otherwise sectors would be lost without notice (there appear to be absolutely no overrun status flags, nor overrun error interrupts).

Update:

This is confirmed, as found in the SCEA_BBS.pdf bulletin board archive:

Sector Buffer Test Cases

Above shows the normal flow when processing INT1's as they arise. Now, inserting delays (and not processing INT1's during that delays):

Above suggests that the CDROM buffer can hold max 2 sectors (the oldest and current one). However, using a longer delay:

Above indicates that sector buffer can hold 8 sectors (as the sector 1 slot is overwritten by sector 9). And, another test with even longer delay:

Above is a special case where sector 17 appears twice; the first one is the sector 1 slot (which was overwritten by sector 9, and apparently then half overwritten by sector 17).

Sector Buffer VS GetlocL Response Tests

Another test, with Delay BEFORE Getloc:

Another test, with Delay AFTER Getloc:

Another test, with Delay BEFORE and AFTER Getloc:

Sector Buffer VS Pause Response Tests

Another test, with Delay BEFORE Pause:

Another test, with Delay AFTER Pause:

Another test, with Delay BEFORE and AFTER Pause:

For above: Note that, despite of Pause, the CDROM is still writing to the internal buffer (and overwrites slot 1 by sector 9) (this might be because the Pause command isn't processed at all until INT1 is processed).

Double Commands (Getloc then Pause)

Another test,

Another test,

Another test,

Double Commands (Pause then Getloc)

Another test,

Another test,

Another test,

Another test,

11.19 CDROM Disk Format

Overview

The PSX uses a ISO 9660 filesystem, with data stored on CD-XA (Mode2) Sectors. ISO 9660 is standard for CDROM disks, although newer CDROMs may use extended filesystems, allowing to use long filenames and lowercase filenames, the PSX Kernel doesn't support such stuff, and, in fact, it's putting some restrictions on the ISO standard: it's limiting file names to MSDOS-style 8.3 format, and it's allowing only a limited number of files and directories per disk.

CDROM Filesystem (ISO 9660 aka ECMA-119)

CDROM Extended Architecture (CD-ROM XA aka CD-XA)

Physical Audio/CDROM Disk Format (ISO/IEC 10149 aka ECMA-130)

Available Documentation

ISO documents are commercial standards (not available for download), however, they are based on ECMA standards (which are free for download, however, the ECMA stuff is in PDF format, so one may treat it as commercial bullshit, too). CD-ROM XA is commercial only (not available for download), and, CD-XA doesn't seem to have become very popular outside of the PSX-world, so there's very little information available, portions of CD-XA are also used in the CD-i standard (which may be a little better or worse documented).

Stuff

Track.Index (stored in subchannel, in BCD format)

Multiple Tracks are usually used only on Audio Disks (one track for each song, numbered 01h and up), a few Audio Disks may also split Tracks into separate fragments with different Index values (numbered 01h and up, but most tracks have only Index 01h). A simple Data Disk would usually contain only one Track (all sectors marked Track=01h and Index=01h), although some more complex Data Disks may have multiple Data tracks and/or Audio tracks.

Minute.Second.Sector (stored in subchannel, and in Data sectors, BCD format)

The sectors on CDROMs and CD Audio disks are numbered in Minutes, Seconds, and 1/75 fragments of a second (where a "second" is referring to single-speed drives, ie. the normal CD Audio playback speed).
Minute.Second.Sector is stored twice in the subchannel (once the "absolute" time, and once the "local" time).
The "absolute" sector number (counted from the begin of the disk) is mainly relevant for Seek purposes (telling the controller if the drive head is on the desired location, or if it needs to move the head backwards or forwards).
The "local" sector number (counted from the begin of the track) is mainly relevant for

Audio Players, allowing to pass the data directly to the Minute:Second display, without needing to subtract the start address of the track.
Data disks are additionally storing the "absolute" values in their Data Areas, basically that's just the subchannel data duplicated, but more precisely assigned - the problem with the subchannel data is that the CD Audio standard seems to lack a clear definition that would assign the begin of the sub-channel block to the exact begin of a sector; so, when using only the subchannel data, some Drive Controllers may assign the begin of a new sector to another location as than other Controllers do, for Audio Disks that isn't too much of a problem, but for Data Disks it'd be fatal.

Subchannels

Each frame contains 8 subchannel bits (named P,Q,R,S,T,U,V,W). So, a sector (with 98 frames) contains 98 bits (12.25 bytes) for each subchannel.
CDROM Subchannels

Error Correction

Each Frame contains 8 bytes Error Correction information, which is mainly used for Audio Disks, but it isn't 100% fail-proof, for that reason, Data Disks are containing additional Error Correction in the 930h-byte data area (the audio correction is probably focusing on repairing the MSBs of the 16bit samples, and gives less priority on the LSBs). Error Correction is some kind of a huge complex checksum, which allows to detect the location of faulty bytes, and to fix them.

930h-Byte Sectors

The "user" area for each sector is 930h bytes (2352 bytes). That region is combined of the 24-byte data per frame (and excludes the 8-byte audio error correction info, and the 1-byte subchannel data).
Most CDROM Controllers are only giving access to this 930h-byte region (ie. there's no way to read the audio error correction info by software, and only limited access to the subchannel data, such like allowing to read only the Q-channel for showing track/minute/second in audio playback mode).
On Audio disks, the 930h bytes are plain data, on Data disks that bytes are containing headers, error correction, and usually only 800h bytes user data (for more info see Sector Encoding chapter).

Sessions

Multi-Sessions are mainly used on CDR's, allowing to append newer data at the end of the disk at a later time. First of, the old session must contain a flag indicating that there may be a newer session, telling the CDROM Controller to search if one such exists (and if that is equally flagged, to search for an even newer session, and so on until reaching the last and newest session).
Each session contains a complete new ISO Volume Descriptor, and may additionally contain new Path Tables, new Directories, and new Files. The Driver Controller is usually recursing only the Volume Descriptor of the newest session. However, the various Directory Records of the new session may refer to old files or old directories from previous sessions, allowing to "import" the older files, or to "rename" or "delete" them by assigning new names to that files, or by removing them from the directory.
The PSX is reportedly not supporting multi-session disks, but that doesn't seem to be correct, namely, the Setsession command is apparently intended for that purpose... though not sure if the PSX Kernel is automatically searching the newest session... otherwise the boot executable in the first session would need to do that manually by software, and redirect control to the boot executable in the last session.

11.20 CDROM Subchannels

Subchannel P

Subchannel P contains some kind of a Pause flag (to indicate muted areas between Audio Tracks). This subchannel doesn't have any checksum, so the data cannot be trusted to be intact (unless when sensing a longer stream of all-one's, or all zero's).
Theoretically, the 98 pause bits are somehow associated to the 98 audio frames (with 24 audio bytes each) of the sector. However, reportedly, Subchannel P does contain two sync bits, if that is true, then there'd be only 96 pause flags for 98 audio frames. Strange.
Note: Another way to indicate "paused" regions is to set Subchannel Q to ADR=1 and Index=00h.

Subchannel Q

contains the following information:

Possible values for the ADR/Control field are:

The 72bit data regions are, depending on the ADR value...

Subchannel Q with ADR=1 during Lead-In -- Table of Contents (TOC)

When Point=01h..99h (Track 1..99) or Point=A2h (Lead-Out):

When Point=A0h (First Track Number):

When Point=A1h (Last Track Number):

ADR=1 should exist in 3 consecutive lead-in sectors.

Subchannel Q with ADR=1 in Data region -- Position

ADR=1 is required to exist in at least 9 out of 10 consecutive data sectors.

Subchannel Q with ADR=1 during Lead-Out -- Position

ADR=1 should exist in 3 consecutive lead-out sectors (and may be then followed by ADR=5 on multisession disks).

Subchannel Q with ADR=2 -- Catalogue number of the disc (UPC/EAN barcode)

If the first digit of the EAN-13 number is "0", then the remaining digits are a UPC-A barcode number. Either the 13-digit EAN-13 number, or the 12-digit UPC-A number should be printed as barcode on the rear-side of the CD package.
The first some digits contain a country code (EAN only, not UPC), followed by a manufacturer code, followed by a serial number. The last digit contains a checksum, which can be calculated as 250 minus the sum of the first 12 digits, minus twice the sum of each second digit, modulated by 10.
ADR=2 isn't included on all CDs, and, many CDs do have ADR=2, but the 13 digits are all zero. Most CDROM drives do not allow to read EAN/UPC numbers.
If present, ADR=2 should exist in at least 1 out of 100 consecutive sectors. ADR=2 may occur also in Lead-in.

Subchannel Q with ADR=3 -- ISRC number of the current track

(ISO 3901 and DIN-31-621):

Most CDROM drives for PC's do not allow to read ISRC numbers (or even worse, they may accidently return the same ISRC number on every two tracks).
If present, ADR=3 should exist in at least 1 out of 100 consecutive sectors. However, reportedly, ADR=3 should not occur in Lead-in.

Subchannel Q with ADR=5 in Lead-in -- Multisession Lead-In Info

When Point=B0h:

When Point=C0h:

And, optionally, when Point=C1h:

Subchannel Q with ADR=5 in Lead-Out -- Multisession Lead-Out Info

Present in 3 consequtive sectors (3x ADR=1, 3x ADR=5, 3x ADR=1, 3x ADR=5, etc).

Subchannel Q with ADR=5 in Lead-in -- CDR/CDRW Skip Info (Audio Only)

When Point=01h..40h:

When Point=B1h:

When Point=B2h,B3h,B4h:

Note: Skip intervals are seldom written by recorders and typically ignored by readers.

Subchannel R..W

Subchannels R..W are usually unused, except for some extended formats:

Most CDROM drives do not allow to read these subchannels. CD-TEXT was designed by Sony and Philips in 1997, so it should be found only on (some) newer discs. Most CD/DVD players don't support it (the only exception is that CD-TEXT seems to be popular for car hifi equipment). Most record labels don't support CD-TEXT, even Sony seems to have discontinued it on their own records after some years (so CD-TEXT is very rare on original disks, however, CDR software does often allow to write CD-TEXT on CDRs).

Subchannel R..W, when used for CD-TEXT in the Lead-In area

CD-TEXT is stored in the six Subchannels R..W. Of the 12.25 bytes (98 bits) per subchannel, only 12 bytes are used. Together, all 6 subchannels have a capacity of 72 bytes (6x12 bytes) per sector. These 72 bytes are divided into four CD-TEXT fragments (of 18 bytes each). The format of these 18 bytes is:

ID1 - Pack Type Indicator:

ID2 - Track Number:

ID3 - Sequence Number:

ID4 - Block Number and Character Position Indicator:

Example Data (generated with CDRWIN):

Each Text string is terminated by a 00h byte (or 0000h for 16bit character set). If there's still room in the 12-byte data region, then first characters for the next Text string (for the next track) are appended after the 00h byte (if there's no further track, then the remaining bytes should be padded with 00h).
The "Blocksize" (ID1=8Fh) consists three packs with 24h bytes of data (first 0Ch bytes stored with ID2=00h, next 0Ch bytes with ID2=01h, and last 0Ch bytes with ID2=02h):

Character Set values (for ID1=8Fh, ID2=00h, DATA[0]=charset):

"In case the same character stings is used for consecutive tracks, character 09h (or 0909h for 16bit charset) may be used to indicate the same as previous track. It shall not used for the first track."

adjust_crc_16_ccitt(addr_len) ;for CD-TEXT and Subchannel Q

11.21 CDROM Sector Encoding

Audio

Mode0 (Empty)

Mode1 (Original CDROM)

Mode2/Form1 (CD-XA)

Mode2/Form2 (CD-XA)

encode_sector

calc_parity(sector,offs,len,j0,step1,step2)

calc_p_parity(sector) = calc_parity(sector,0,43,19,2*43,2)
calc_q_parity(sector) = calc_parity(sector,43*4,26,0,2*44,2*43)

adjust_edc(addr,len)

init_tables

subfunc(a,b)

11.22 CDROM XA Subheader, File, Channel, Interleave

The Sub-Header for normal data sectors is usually 00h,00h,08h,00h (some PSX sectors have 09h instead 08h, indicating the end of "something" or so?

1st Subheader byte - File Number (FN)

2nd Subheader byte - Channel Number (CN)

3rd Subheader byte - Submode (SM)

The EOR bit is set in all Volume Descriptor sectors, the last sector (ie. the Volume Descriptor Terminator) additionally has the EOF bit set. Moreover, EOR and EOF are set in the last sector of each Path Table, and last sector of each Directory, and last sector of each File.

4th Subheader byte - Codinginfo (CI)

When used for Data sectors:

When used for XA-ADPCM audio sectors:

Audio/Video Interleave (Multiple Files/Channels)

The CDROM drive mechanics are working best when continously following the data spiral on the disk, that works fine for uncompressed Audio Data at normal speed, but compressed Audio Data the disk is spinning much too fast. To avoid the drive to need to pause reading or to do permanent backwards seeking, CD-XA allows to store data interleaved in separate files/channels. With common interleave values like so:

For example, 1/8 means that the controller processes only each 8th sector (each having the same File Number and Channel Number), and ignores the next 7 sectors (which must have other File Number and/or other Channel Number). There are various ways to arrange multiple files or channels, for example,

(*) If the Audio and Video data belongs together then both should use the SAME channel.
Note: Above interleave values are assuming that PSX Game Disks are always running at double speed (that's fastest for normal data files, and ADPCM files are usually using the same speed; otherwise it'd be neccessary to change the drive speed everytime when switching between Data to ADPCM modes).
Note: The file/channel numbers can be somehow selected with the Setfilter command. No idea if the controller is automatically switching to the next channel or so when reaching the end of the file?

Real Time Streaming

With the above Interleave, files can be played continously at real time - that, unless read-errors do occur. In that case the drive controller would usually perform time-consuming error-correction and/or read-retries. For video/audio streaming the resulting delay would be tendencially more annoying as than processing or skipping the incorrect

data.
In such cases the drive controller is allowed to ignore read errors; that probably on sectors that have the Real Time (RT) flag set in their subheaders. The controller is probably doing some read-ahead buffering (so, if it has buffered enough data, then it may still perform read retries and/or error correction, as long as it doesn't affect real time playback).

11.23 CDROM XA Audio ADPCM Compression

CD-ROM XA ADPCM is used for Audio data compression. Each 16bit sample is encoded in 4bit nibbles; so the compression rate is almost 1:4 (only almost 1:4 because there are 16 header bytes within each 128-byte portion). The data is usually/always stored on 914h-byte sectors (without error correction).

Subheader

The Subheader (see previous chapter) contains important info for ADPCM: The file/ channel numbers for Interleaved data, and the codinginfo flags: mono/stereo flag, 37800Hz/18900Hz sampling rate, 4bit/8bit format, and emphasis.

ADPCM Sectors

Each sector consists of 12h 128-byte portions (=900h bytes) (the remaining 14h bytes of the sectors 914h-byte data region are 00h filled).
The separate 128-byte portions consist of a 16-byte header,

followed by twentyeight data words (4x28-bytes),

and then followed by the next 128-byte portion.
The "Copy" bytes are allowing to repair faulty headers (ie. if the CDROM controller has sensed a read-error in the header then it can eventually replace it by the copy of the header).

XA-ADPCM Header Bytes

Note: The 4bit (or 8bit) samples are expanded to 16bit by left-shifting them by 12 (or 8), that 16bit value is then right-shifted by the selected 'shift' amount. For 8bit ADPCM shift should be 0..8 (values 9..12 will cut-off the LSB(s) of the 8bit value, this works, but isn't useful). For both 4bit and 8bit ADPCM, reserved shift values 13..15 will act same as shift=9).

XA-ADPCM Data Words (32bit, little endian)

or, for 8bit ADPCM format:

decode_sector(src)

decode_28_nibbles(src,blk,nibble,dst,old,older)

Pos/neg Tables

Note: XA-ADPCM supports only four filters (0..3), unlike SPU-ADPCM which supports five filters (0..4).

Old/Older Values

The incoming old/older values are usually that from the previous part, or garbage (in case of decoding errors in the previous part), or whatever (in case there was no previous part) (ie. maybe zero on power-up?) (and maybe there's also a way to reset the values to zero at the begin of a new file, or *maybe* it's silently done automatically when issuing seek commands?).

25-point Zigzag Interpolation

The CDROM decoder is applying some weird 25-point zigzag interpolation when resampling the 37800Hz XA-ADPCM output to 44100Hz. This part is different from SPU-

ADPCM (which uses 4-point gaussian pitch interpolations). For example, XA-ADPCM interpolation applied to a square wave looks like this:

The zigzagging does produce some (inaudible) 22050Hz noise, and does produce some low-pass (?) filtering ("sinc filter"). The effect can be reproduced somewhat like so:

The above formula/table gives nearly correct results, but with small rounding errors in some cases - possibly due to actual rounding issues, or due to factors with bigger fractional portions, or due to a completely different formula...
Probably, the hardware does actually do the above stuff in two steps: first, applying a zig-zag filter (with only around 21-points) to the 37800Hz output, and then doing 44100Hz interpolation (2-point linear or 4-point gaussian or whatever) in a second step. That two-step theory would also match well for 18900Hz resampling (which has lower-pitch zigzag, and gets spread accross about fifty 44100Hz samples).

XA-ADPCM Emphasis

With XA-Emphasis enabled in Sub-header, output will appear as so:

The exact XA-Emphasis formula is unknown (maybe it's just same as for CD-DA's SUBQ emphasis). Additionally, zig-zag interpolation is applied (somewhere before or after applying the emphasis stuff).
Note: The Emphasis feature isn't used by any known PSX games.

Uninitialized Six-step Counter

The hardware does contain some six-step counter (for interpolating 37800Hz to 44100Hz, ie. to insert one extra sample after each six samples). The 900h-byte sectors contain a multiple of six samples, so the counter will be always same before & after playing a sector. However, the initial counter value on power-up is uninitialized random (and the counter will fallback to that initial random setting after each 900h-byte sector).

RIFF Headers (on PCs)

When reading files that consist of 914h-byte sectors on a PC, the PC seems to automatically insert a 2Ch-byte RIFF fileheader. Like so, for ADPCM audio files:

That RIFF stuff isn't stored on the CDROM (at least not in the file area) (however, some of that info, like the "=UXA" stuff, is stored in the directory area of the CDROM).
After the RIFF header, the normal sector data is appended, that, with the full 930h bytes per sector (ie. the 914h data bytes preceeded by sync bytes, header, subheader, and followed by the EDC value).
The Channel Interleave doesn't seem to be resolved, ie. the Channels are kept arranged as how they are stored on the CDROM. However, File Interleave \<should> be resolved, ie. other Files that "overlap" the file shouldn't be included in the file.

11.24 CDROM ISO Volume Descriptors

System Area (prior to Volume Descriptors)

The first 16 sectors on the first track are the system area, for a Playstation disk, it contains the following:

Of which, the Licence String in sector 4 is, The Playstation Logo in sectors 5..11 contains data like so,

the Logo contains a header, polygons, vertices and normals for the "PS" logo (which is displayed when booting from CDROM). Some BIOS versions are comparing these 3278h bytes against an identical copy in ROM, and refuse to boot if the data isn't 1:1 the same:
- US/ASIA BIOS always accepts changed logos.
- PAL BIOS accepts changed logos up to v3.0E (and refuses in v4.0E and up).
- JP BIOS never accepts changed logos (and/or changed license strings?).

Note: PAL BIOS with "region-patch-modchip" does behave same as US/ASIA BIOS.

Volume Descriptors (Sector 16 and up)

Playstation disks usually have only two Volume Descriptors,

Primary Volume Descriptor (sector 16 on PSX disks)

Volume Descriptor Set Terminator (sector 17 on PSX disks)

Boot Record (none such on PSX disks)

Supplementary Volume Descriptor (none such on PSX disks)

Volume Partition Descriptor (none such on PSX disks)

Reserved Volume Descriptors (none such on PSX disks)

11.25 CDROM ISO File and Directory Descriptors

The location of the Root Directory is described by a 34-byte Directory Record being located in Primary Volume Descriptor entries 09Ch..0BDh. The data therein is: Block Number (usually 22 on PSX disks), LEN_FI=01h, Name=00h, and, LEN_SU=00h (due to the 34-byte limit).

Format of a Directory Record

LEN_SU can be calculated as "LEN_DR-(33+LEN_FI+Padding)". For CD-XA disks (as used in the PSX), LEN_SU is 14 bytes:

The names are alphabetically sorted, no matter if the names refer to files or directories (ie. SUBDIR would be inserted between STRFILE.EXT and SYSFILE.EXT). The first two entries (with non-ascii names 00h and 01h) are referring to current and parent directory.

Path Tables

The Path Table contain a summary of the directory names (the same information is also stored in the directory records, so programs may either use path tables or directory records; the path tables are allowing to read the whole directory tree quickly at once, without neeeding to seek from directory to directory).
Path Table 1 is in Little-Endian format, Path Table 3 contains the same data in Big-Endian format. Path Table 2 and 4 are optional copies of Table 1 and 3. The size and

location of the tables is stored in Volume Descriptor entries 084h..09Bh. The format of the separate entries within a Path Table is,

The first entry (directory number 0001h) is the root directory, the root doesn't have a name, nor a parent (the name field contains a 00h byte, rather than ASCII text, LEN_DI is 01h, and parent is 0001h, making the root it's own parent; ignoring the fact that incest is forbidden in many countries).
The next entries (directory number 0002h and up) (if any) are sub-directories within the root (sorted in alphabetical order, and all having parent=0001h). The next entries are sub-directories (if any) of the first sub-directory (also sorted in alphabetical order, and all having parent=0002h). And so on.
PSX disks usually contain all four tables (usually on sectors 18,19,20,21).

Format of an Extended Attribute Record (none such on PSX disks)

If present, an Extended Attribute Record shall be recorded over at least one Logical Block. It shall have the following contents.

Unknown WHERE that data is located... the Directory Records can specify the Extended Attribute Length, but not the location... maybe it's meant to be located in the first some bytes or blocks of the File or Directory...?

11.26 CDROM ISO Misc

Both Byte Order

All 16bit and 32bit numbers in the ISO region are stored twice, once in Little-Endian order, and then in Big-Endian Order. For example,

Exceptions are the 16bit Permission Flags which are stored only in Little-Endian format (although the flags are four 4bit groups, so that isn't a real 16bit number), and, the Path Tables are stored in both formats, but separately, ie. one table contains only Little-Endian numbers, and the other only Big-Endian numbers.

d-characters (Filenames)

a-characters

Ie. all ASCII characters from 20h..5Fh except "#$@[]^"

SEPARATOR 1 = 2Eh (aka ".") (extension; eg. "EXT")
SEPARATOR 2 = 3Bh (aka ";") (file version; "1".."32767")

Fixed Length Strings/Filenames

The Volume Descriptors contain a number fixed-length string/filename fields (unlike the Directory Records and Path Tables which have variable lengths). These fields should be padded with SPACE characters if they are empty, or if the string is shorter than the maximum length.
Filename fields in Volume Descriptors are referring to files in the Root Directory. On PSX

disks, the filename fields are usually empty, but some disks are mis-using the Copyright Filename to store the Company Name (although no such file exists on the disk).

Volume Descriptor Timestamps

The various timestamps occupy 17 bytes each, in form of

The first 16 bytes are ASCII Date and Time digits (Year, Month, Day, Hour, Minute, Second, and 1/100 Seconds. The last byte is Offset from Greenwich Mean Time in number of 15-minute steps from -48 (West) to +52 (East); or actually: to +56 when recursing Kiribati's new timezone.
Note: PSX games manufactured in year 2000 were accidently marked to be created in year 0000.

Recording Timestamps

Occupy only 7 bytes, in non-ascii format

The year ranges from 1900+0 to 1900+255.

File Flags

If this Directory Record identifies a directory then bit 2,3,7 shall be set to ZERO.
If no Extended Attribute Record is associated with the File Section identified by this Directory Record then bit positions 3 and 4 shall be set to ZERO.

Permission Flags (in Extended Attribute Records)

This is a bit bizarre, an upper-class owner is "an owner who is a member of a group of the System class of user". An upper-class user is "any user who is a member of the group specified by the Group Identification field". The separate 4bit permission codes are:

11.27 CDROM File Formats

FILENAME.EXT

The BIOS seems to support only (max) 8-letter filenames with 3-letter extension, typically all uppercase, eg. "FILENAME.EXT". Eventually, once when the executable has started, some programs might install drivers for long filenames(?)

SYSTEM.CNF

Contains boot info in ASCII/TXT format, similar to the CONFIG.SYS or AUTOEXEC.BAT files for MSDOS. A typical SYSTEM.CNF would look like so:

The first line specifies the executable to load, from the "cdrom:" drive, "\" root directory, filename "abcd_123.45" (case-insensitive, the real name in the disk directory would be uppercase, ie. "ABCD_123.45"), and, finally ";1" is the file's version number (a rather

strange ISO-filesystem specific feature) (the version number should be usually/always 1). Additionally, "arg" may contain an optional 128-byte command line argument string, which is copied to address 00000180h, where it may be interpreted by the executable (most or all games don't use that feature).
Each line in the file should be terminated by 0Dh,0Ah characters... not sure if it's also working with only 0Dh, or only 0Ah...?

A note on the "ABCD_123.45" file:
This is a normal executable (exactly as for the .EXE files, described below), however, the filename/extension is taken from the game code (the "ABCD-12345" text that is printed on the CD cover), but, with the minus replaced by an underscore, and due to the 8-letter filename limit, the last two characters are stored in the extension region.
That "XXXX_NNN.NN" naming convention seems to apply for all official licensed PSX games, not sure if it's possible to specify something like "FILENAME.EXE" as boot-file.

XXXX_NNN.NN (Boot-Executable) (filename specified in SYSTEM.CNF)

FILENAME.EXE (General-Purpose Executable)

PSX executables are having an 800h-byte header, followed by the code/data.

The code/data is simply loaded to the specified destination address, ie. unlike as in MSDOS .EXE files, there is no relocation info in the header.

Note: In bootfiles, SP is usually 801FFFF0h (ie. not 801FFF00h as in system.cnf). When SP is 0, the unmodified caller's stack is used. In most cases (except when manually calling DoExecute), the stack values in the exeheader seem to be ignored though (eg. replaced by the SYSTEM.CNF value).
The memfill region is zerofilled by a "relative" fast word-by-word fill (so address and size must be multiples of 4) (despite of the word-by-word filling, still it's SLOW because the memfill executes in uncached slow ROM).
The reserved region at [038h-04Bh] is internally used by the BIOS to memorize the caller's RA,SP,R30,R28,R16 registers (for some bizarre reason, this information is saved in the exe header, rather than on the caller's stack).
Additionally to the initial PC,R28,SP,R30 values that are contained in the header, two parameter values are passed to the executable (in R4 and R5 registers) (however, usually these values are simply R4=1 and R5=0).
Like normal functions, the executable can return control to the caller by jumping to the incoming RA address (provided that it hasn't destroyed the stack or other important memory locations, and that it has pushed/popped all registers) (returning works only for non-boot executables; if the boot executable returns to the BIOS, then the BIOS will simply lockup itself by calling the "SystemErrorBootOrDiskFailure" function.

The PSX uses the standard CDROM ISO9660 filesystem without any encryption (ie. you can put an original PSX CDROM into a DOS/Windows computer, and view the content of the files in text or hex editors without problems).

PSX.EXE

Some games do not have a SYSTEM.CNF file, in which case the kernel will fallback on loading a file named PSX.EXE.

11.28 CDROM Protection - SCEx Strings

SCEx String

The heart of the PSX copy-protection is the four-letter "SCEx" string, encoded in the wobble signal of original PSX disks, which cannot reproduced by normal CD writers. The last letter varies depending on the region, "SCEI" for Japan, "SCEA" for America (and all other NTSC countries except Japan), "SCEE" for Europe (and all other PAL countries like Australia). If the string is missing (or if it doesn't match up for the local region) then the PSX refuses to boot. The verification is done by the Firmware inside of the CDROM

Controller (not by the PSX BIOS, so there's no way to bypass it by patching the BIOS ROM chip).

Wobble Groove and Absolute Time in Pregroove (ATIP) on CD-R's

A "blank" CDR contains a pre-formatted spiral on it. The number of windings in the spiral varies depending on the number of minutes that can be recorded on the disk. The spiral isn't made of a straight line (------), but rather a wobbled line (/\/\/), which is used to adjust the rotation speed during recording; at normal drive speed, wobble should produce a 22050Hz sine wave.
Additionally, the CDR wobble is modulated to provide ATIP information, ATIP is used for locating and positioning during recording, and contains information about the approximate laser power necessary for recording, the last possible time location that lead out can start, and the disc application code.
Wobble is commonly used only on (recordable) CDRs, ie. usually NOT on (readonly) CDROMs and Audio Disks. The copyprotected PSX CDROMs are having a short CDR-style wobble period in the first some seconds, which seems to contain the "SCEx" string instead of ATIP information.

Other Protections

Aside from the SCEx string, PSX disks are required to contain region and licence strings (in the ISO System Area, and in the .EXE file headers), and the "PS" logo (in the System Area, too). This data can be reproduced with normal CD writers, although it may be illegal to distribute unlicensed disks with licence strings.

11.29 CDROM Protection - Bypassing it

Modchips

A modchip is a small microcontroller which injects the "SCEx" signal to the mainboard, so the PSX can be booted even from CDRs which don't contain the "SCEx" string. Some modchips are additionally patching region checks contained in the BIOS ROM.
Note: Although regular PSX disks are black, the hardware doesn't verify the color of the disks, and works also with normal silver disks.

Disk-Swap-Trick

Once when the PSX has recognized a disk with the "SCEx" signal, it'll be satisfied until a new disk is inserted, which is sensed by the SHELL_OPEN switch. When having that switch blocked, it is possible to insert a CDR without the PSX noticing that the disk was changed.
Additionally, the trick requires some boot software that stops the drive motor (so the new disk can be inserted, despite of the PSX thinking that the drive door is still closed), and that does then start the boot executable on the new disk.
The boot software can be stored on a special boot-disk (that do have the "SCEx" string on it). Alternately, a regular PSX game disk could be used, with the boot software stored somewhere else (eg. on Expansion ROM, or BIOS ROM replacement, or Memory Card).

Booting via BIOS ROM or Expansion ROM

The PSX can be quite easily booted via Expansion ROM, or BIOS ROM replacements, allowing to execute code that is stored in the ROM, or that is received via whatever serial or parallel cable connection from a PC.
However, even with a BIOS replacement, the protection in the CDROM controller is still active, so the ROM can't read "clean" data from the CDROM Drive (unless the Disk-Swap trick is used).
Whereas, no "clean" data doens't mean no data at all. The CDROM controller does still seem to output "raw" data (without removing the sector header, and without handling error correction, and with only limited accuracy on the sector position). So, eventually, a customized BIOS could convert the "raw" data to "clean" data.

Secret Unlock Commands

There is an "official" backdoor that allows to disable the SCEx protection by software via secret commands (for example, sending those commands can be done via BIOS patches, nocash BIOS clone, or Expansion ROMs).
CDROM - Secret Unlock Commands

Booting via Memory Card

Some games that load data from memory cards may get confused if the save data isn't formatted as how they expect it - with some fine tuning you can get them to "crash" in a manner that they do accidently execute bootcode stored on the memory card.
Requires a tools to write to the memory card (eg. parallel port cable), and the memory

card data customized for a specific game, and an original CDROM with that specific game. Once when the memory card code is booted, the Disk-Swap trick can be used.

11.30 CDROM Protection - Modchips

Modchip Source Code

The Old Crow mod chip source code works like so:

Connection for the data/gate/sync signals:

For older PSX boards (data/gate):

For newer PSX and PSone boards (data/sync):

On the mainboard should be a big SMD capacitor (connected to the "data" pin), and a big testpoint (connected to the "sync" pin); it's easier to connect the signals to that locations than to the tiny CXD-chip pins.
gate and data must be tristate outputs, or open-collector outputs (or normal high/low outputs passed through a diode).

Note on "data" pin (all boards)

Transfers the "SCEx" data. Note that the signal produced by the modchip is looking entirly different than the signal produced by original disks, the real signal would be modulated 22050Hz wobble, while the modchip is simply dragging the signal permanently LOW throughout "1" bits, and leaves it floating for "0" bits. Anyways the "faked" signal seems to be accurate enough to work.

Note on "gate" pin (older PSX boards only)

The "gate" pin needs to be LOW only for use with original licensed disks (reportedly otherwise the SCEx string on that disks would conflict with the SCEx string from the modchip).
At the mainboard side, the "gate" signal is an input, and "data" is an inverted output of the gate signal (so dragging gate to low, would cause data to go high).

Note on "sync" pin (newer PSX and PSone boards only)

The "sync" pin is a testpoint on the mainboard, which does (at single speed) output a frequency of circa 44.1kHz/6 (of which some clock pulses seem to be longer or shorter, probably to indicate adjustments to the rotation speed).
Some modchips are connected directly to "sync" (so they are apparently synchronizing the data output with that signal; which is not implemented in the above source code).
Anyways, other modchips are using a more simplified connection: The modchip itself connects only to the "data" pin, and "sync" is required to be wired to IC723.Pin17.

Note on Multi-Region chips

Modchips that are designed to work in different regions are sending a different string (SCEA, SCEE, SCEI) in each loop cycle. Due to the slow 250bps transfer rate, it may take a while until the PSX has received the correct string, so this multi-region technique may cause a noticeable boot-delay.

Stealth (hidden modchip)

The Stealth connection is required for some newer games with anti-modchip protection, ie. games that refuse to run if they detect a modchip. The detection relies on the fact that the SCEx signal is normally received only when booting the disk, whilst older modchips were sending that signal permanently. Stealth modchips are sending the signal only on power-up (and when inserting a new disk, which can be sensed via SHELL_OPEN signal).
Modchip detection reportedly works like so (not too sure if all commands are required, some seem to be rather offtopic):

If GetSCExInfo returns nonzero values, then the console is equipped with a modchip, and if so, anti-modchip games would refuse to work (no matter if the disk is an illegal copy, or not).

NTSC-Boot BIOS Patch

Typically connects to two or three BIOS address/data lines, apparently watching that signals, and dragging a data line LOW at certain time, to skip software based region checks (eg. allowing to play NTSC games on PAL consoles).
Aside from the modchip connection, that additionally requires to adjust the video signal (in 60Hz NTSC mode, the PSX defaults to generate a NTSC video signal) (whilst most

PAL screens can handle 60Hz refresh, they can't handle NTSC colors) (on PSone boards, this can be fixed simply by grounding the /PAL pin; IC502.Pin13) (on older PSX boards it seems to be required to install an external color clock generator).

MODCHIP Connection Example

Connection for 8pin "12C508" mod chip from fatcat.co.nz for a PAL PSone with PM-41 board (ie. with 208pin SPU CXD2938Q, and 52pin IC304 "C 3060, SC430943PB"):

The chip can be used in a Basic connection (with only pin1,5,6,8 connected), or Stealth and NTSC-Boot connection (additionally pin2,3,4,7 connected). Some other modchips (such without internal oscillator) are additionally connected to a 4MHz or 4.3MHz signal on the mainboard. Some early modchips also connected to a bunch of additional pins that were reportedly for power-on timings (whilst newer chips use hardcoded power-on delays).

Nocash BIOS "Modchip" Feature

The nocash PSX bios outputs the "data" signal on the A20 address line, so (aside from the BIOS chip) one only needs to install a 1N4148 diode and two wires to unlock the CDROM:

With the "sync" connection, the SCEx signal from the disk is disabled (ie. even original licensed disks are no longer recognized, unless SCEx is output via A20 by software). For more variants, see:
CDROM Protection - Chipless Modchips

11.31 CDROM Protection - Chipless Modchips

The nocash kernel clone outputs a SCEX signal via A20 and A21 address lines, (so one won't need a separate modchip/microprocessor):

When using the clone bios as internal ROM replacement, A20 can be used with simple wires/diodes. Doing that with external expansion ROMs would cause the console to stop working when unplugging the ROM, hence needing a slightly more complex circuit with transistors/logic chips.

External Expansion ROM version, for older boards (PU-7 through PU-20):

External Expansion ROM version, for newer boards (PU-22):

Internal Kernel ROM version, for older boards (PU-7 through PU-20):

Internal Kernel ROM version, for newer boards (PU-22 through PM-41(2)):

What pin is where...

GATE on PU-18 is usually IC706.Pin7 (but IC706.Pin10 reportedly works, too).
GATE on PU-20 is usually IC706.Pin10 (but IC706.Pin7 might work, too).

11.32 CDROM Protection - LibCrypt

LibCrypt is an additional copy-protection, used by about 100 PSX games. The protection uses a 16bit decryption key, which is stored as bad position data in Subchannel Q. The 16bit key is then used for a simple XOR-decryption on certain 800h-byte sectors.

Protected sectors generation schemas

There are some variants on how the Subchannel Q data is modified:

Anyways, the relevant part is that the modified sectors have wrong CRCs (which means that the PSX cdrom controller will ignore them, and the GetlocP command will keep returning position data from the previous sector).

LibCrypt sectors

The modified sectors could be theoretically located anywhere on the disc, however, all known protected games are having them located on the same sectors:

Each bit is stored twice on Minute=03 (five sectors apart). For some reason, there is also a "backup copy" on Minute=09 (however, the libcrypt software doesn't actually support using that backup stuff, and, some discs don't have the backup at all (namely, discs with less than 10 minutes on track 1?)).
A modified sector means a "1" bit, an unmodified means a "0" bit. The 16bit keys of the existing games are always having eight "0" bits, and eight "1" bits (meaning that there are 16 modified sectors on Minute=03, and, if present, another 16 ones one Minute=09).

Example (Legacy of Kain)

Legacy of Kain (PAL) is reading the LibCrypt data during the title screen, and does then display GOT KEY!!! on TTY terminal (this, no matter if the correct 16bit key was received).
The actual protection jumps in a bit later (shortly after learning to glide, the game will hang when the first enemies appear if the key isn't okay). Thereafter, the 16bit key is kept used once and when to decrypt 800h-byte sector data via simple XORing.
The 16bit key (and some other related counters/variables) aren't stored in RAM, but rather in COP0 debug registers (which are mis-used as general-purpose storage in this case), for example, the 16bit key is stored in LSBs of the "cop0r3" register.

11.33 CDROM Disk Images CCD/IMG/SUB (CloneCD)

File.IMG - 2352 (930h) bytes per sector

Contains the sector data, recorded at 930h bytes per sector. Unknown if other sizes are also used/supported (like 800h bytes/sector, or even images with mixed sizes of 800h and 930h for different tracks).

File.SUB - 96 (60h) bytes per sector (subchannel P..W with 96 bits each)

Contains subchannel data, recorded at 60h bytes per sector.

Optionally, the .SUB file can be omitted (it's needed only for discs with non-standard subchannel data, such like copy-protected games).

File.CCD - Lead-in info in text format

Contains Lead-in info in ASCII text format. Lines should be terminated by 0Dh,0Ah. The overall CCD filestructure is:

Read on below for details on the separate sections.

[CloneCD]

[Disc]

[CDText]

[Session 1]

[Entry 0]

[Entry 0..2] are usually containing Point A0h..A2h info. [Entry 3..N] are usually TOC info for Track 1 and up.

[TRACK 1] ;-track number (non-BCD) (1..99)

Missing Sectors & Sector Size

The .CCD file doesn't define the "PreGapSize" (the number of missing sectors at begin of first track). It seems to be simply constant " PreGapSize=150". Unless one is supposed to calculate it as "PreGapSize=((PMin*60+PSec)*75+PFrame)-PLBA".
The SectorSize seems to be also constant, "SectorSize=930h".

Non-BCD Caution

All Min/Sec/Frame/Track/Index values are expressed in non-BCD, ie. they must be converted to BCD to get the correct values (as how they are stored on real CDs). Exceptions are cases where those bytes have other meanings: For example, "PSec=32" does normally mean BcdSecond=32h, but for Point A0h it would mean DiscType=20h=CD-ROM-XA).
The Point value is also special, it is expressed in hex (0xNN), but nonetheless it is non-BCD, ie. Point 1..99 are specified as 0x01..0x63, whilst, Point A0h..FFh are specified as such (ie. as 0xA0..0xFF).

Versions

Version=1 doesn't seem to exist (or it is very rare). Version=2 is quite rare, and it seems to lack the [TRACK N] entries (meaning that there is no MODE and INDEX information, except that the INDEX 1 location can be assumed to be same as PLBA). Version=3 is most common, this version includes [TRACK N] entries, but often only with INDEX=1 (and up, if more indices), but without INDEX 0 (on Track 1 it's probably missing due to pregap, on further Tracks it's missing without reason) (so, only ways to reproduce INDEX=0 would be to guess it being located 2 seconds before INDEX=1, or, to use the information from the separate .SUB file, if that file is present; note: presence of index 0 is absolutely required for some games like PSX Tomb Raider 2).

Entry & Points & Sessions

The [Entry N] fields are usually containing Point A0h,A1h,A2h, followed by Point 1..N (for N tracks). For multiple sessions: The session is terminated by Point B0h,C0h. The next session does then contain Point A0h,A1h,A2h, and Point N+1..X (for further tracks).
The INDEX values in the [TRACK N] entries are originated at the begin of the corresponding session, whilst PLBA values in [Entry N] entries are always originated at the begin of the disk.

11.34 CDROM Disk Images CDI (DiscJuggler)

Overall Format

Sector Data

Contains Sector Data for sector 00:00:00 and up (ie. all sectors are stored in the file, there are no missing "pregap" sectors).
Sector Size can be 800h..990h bytes/sector (sector size may vary per track).

Number of Sessions (1 byte)

Session Block (15-bytes)

Track/Disc Header (30h+F bytes) (used in Track Blocks and Disc Info Block)

Track Block (E4h+F+I+T bytes)

Disc Info Block (5Fh+F+V+T bytes)

Entrypoint (4 bytes) (located at "Filesize-4")

11.35 CDROM Disk Images CUE/BIN/CDT (Cdrwin)

.CUE/.BIN (CDRWIN)

CDRWIN stores disk images in two separate files. The .BIN file contains the raw disk image, starting at sector 00:02:00, with 930h bytes per sector, but without any TOC or subchannel information. The .CUE file contains additional information about the separate track(s) on the disk, in ASCII format, for example:

The .BIN file does not contain ALL sectors, as said above, the first 2 seconds are not stored in the .BIN file. Moreover, there may be missing sectors somewhere in the middle

of the file (indicated as PREGAP in the .CUE file; PREGAPs are usually found between Data and Audio Tracks).
The MM:SS:FF values in the .CUE file are logical addresses in the .BIN file, rather than physical addresses on real CDROMs. To convert the .CUE values back to real addresses, add 2 seconds to all MM:SS:FF addresses (to compensate the missing first 2 seconds), and, if the .CUE contains a PREGAP, then the pregap value must be additionally added to all following MM:SS:FF addresses.
The end address of the last track is not stored in the .CUE, instead, it can be only calculated by converting the .BIN filesize to MM:SS:FF format and adding 2 seconds (plus any PREGAP values) to it.

FILE \<filename> BINARY|MOTOTOLA..or..MOTOROLA?|AIFF|WAVE|MP3

FLAGS DCP 4CH PRE SCMS

INDEX NN MM:SS:FF

TRACK NN datatype

PREGAP MM:SS:FF

POSTGAP MM:SS:FF

Duration of silence at the begin (PREGAP) or end (POSTGAP) of a track. Even if it isn't specified, the first track will always have a 2-second pregap.
The gaps are NOT stored in the BIN file.

REM comment

Allows to insert comments/remarks (which are usually ignored). Some third-party tools are mis-using REM to define additional information.

CATALOG 1234567890123

ISRC ABCDE1234567

PERFORMER "The Band"

SONGWRITER "The Writer"

TITLE "The Title"

These entries allow to define basic CD-Text info directly in the .CUE file.
Some third-party utilites allow to define additional CD-Text info via REM lines, eg. "REM GENRE Rock".
Alternately, more complex CD-Text data can be stored in a separate .CDT file.

CDTEXTFILE "C:\LONG FILENAME.CDT"

Specifies an optional file which may contain CD-TEXT. The .CDT file consists of raw 18-byte CD-TEXT fragments (which may include any type of information, including exotic one's like a "Message" from the producer). For whatever reason, there's a 00h-byte appended at the end of the file. Alternately to the .CDT file, the less exotic types of CD-TEXT can be defined by PERFORMER, TITLE, and SONGWRITER commands in the .CUE file.

Missing

Unknown if newer CUE/BIN versions do also support subchannel data.

11.36 CDROM Disk Images MDS/MDF (Alcohol 120%)

File.MDF - Contains sector data (optionally with sub-channel data)

Contains the sector data, recorded at 800h..930h bytes per sector, optionally followed by 60h bytes subchannel data (appended at the end of each sector). The stuff seems to be start on 00:02:00 (ie. the first 150 sectors are missing; at least it is like so when "Session Start Sector" is -150).
The subchannel data (if present) consists of 8 subchannels, stored in 96 bytes (each byte containing one bit per subchannel).

The 96 bits (per subchannel) can be translated to bytes, as so:

File.MDS - Contains disc/lead-in info (in binary format)

An MDS file's structure consists of the following stuff ...

Header (58h bytes)

Session-Blocks (18h bytes)

Data-Blocks (50h bytes)

Block 0..2 are usually containing Point A0h..A2h info. Block 3..N are usually TOC info for Track 1 and up.

For Point>=A0h, below 44h bytes at [0Ch..4Fh] are zero-filled

Trackmode:

Index Blocks (usually 8 bytes per track)

Index blocks are usually 8 bytes in size (two indices per track). Maybe this block is/was intended to allow to contain more indices (although the Alcohol 120% does always store only 2 indices, even when recording a CD with more than 2 indices per track).
The MDS file does usually contain Index blocks for \<all> Data Blocks (ie. including unused dummy Index Blocks for Data Blocks with Point>=A0h).

Filename Blocks (10h bytes)

Normally all tracks are sharing the same filename block (although theoretically the tracks could use separate filename blocks; with different filenames).

Filename Strings (usually 6 bytes)

Contains the filename of the of the sector data (usually "*.mdf", indicating to use the same name as for the .mds file, but with .mdf extension).

Missing

Unknown if/how this format supports EAN-13, ISRC, CD-TEXT.
Unknown if Track/Point/Index are BCD or non-BCD.

11.37 CDROM Disk Images NRG (Nero)

.NRG (NERO)

Nero is probably the most bloated and most popular CD recording software. The first part of the file contains the disk image, starting at sector 00:00:00, with 800h..930h bytes per sector. Additional chunk-based information is appended at the end of the file, usually consisting of only four chunks: CUES,DAOI,END!,NERO (in that order).

Chunk Entrypoint (in last 8/12 bytes of file)

Cue Sheet (summary of the Table of Contents, TOC)

below EIGHT bytes repeated for each track/index,
of which, first FOUR bytes are same for both CUES and CUEX,

next FOUR bytes for CUES,

or, next FOUR four bytes for CUEX,

Caution: Above may contain two position 00:00:00 entries: one nonsense entry for Track 00 (lead-in), followed by a reasonable entry for Track 01, Index 00.

Disc at Once Information

below repeated for each track,

End of chain

Track Information (contained only in Track at Once images)

below repeated for each track,

Unknown 1 (contained only in Track at Once images)

Unknown 2 (contained only in Track at Once images)

Session Info (begin of a session) (contained only in multi-session images)

CD-Text (contained only in whatever images)

below repeated for each fragment,

Media Type? (contained only in whatever images)

Notes

Newer/older .NRG files may contain 32bit/64bit values (and use "OLD"/"NEW" chunk names) (as indicated by the "/" slashes).
CAUTION: All 16bit/32bit/64bit values are in big endian byte-order.

Missing

Unknown if newer NRG versions do also support subchannel data.

11.38 CDROM Disk Image/Containers CDZ

.CDZ is a compressed disk image container format (developed by pSX Author, and used only by the pSX emulator). The disk is split into 64kbyte blocks, which allows fast random access (without needing to decompress all preceeding sectors).
However, the compression ratio is surprisingly bad (despite of being specifically designed for cdrom compression, the format doesn't remove redundant sector headers, error correction information, and EDC checksums).

.CDZ File Structure

.CDZ Chunk Format

Chunk Header in v0 (unreleased prototype):

Chunk Header in v1 (first released version):

Chunk Header in v2 and up (later versions):

Chunk Body (same in all versions):

Chunk Footer in v0 (when above header didn't have the "ZLIB" ID):

Chunk Footer in v1 and up:

The "Compressed ZLIB Data" parts contain Deflate'd data (starting with 2-byte ZLIB header, and ending with 4-byte ZLIB/ADLER checksum), for details see:
Inflate - Core Functions
Inflate - Initialization & Tree Creation
Inflate - Headers and Checksums

.CDZ Chunks / Content

The chunk(s) have following content:

Note: cdztool doesn't actually recognize files with .ISO extension (however, one can rename them to .BIN, and then compress them as CUE-less .BIN file).

Cdztool.exe Versions

Note: v0 wasn't ever released (it's only noteworthy because later versions do have backwards compatibility for decompressing old v0 files). v1 didn't work with all operating systems (on Win98 it just says "Error: Couldn't create \<output>" no matter what one is doing, however, v1 does work on later windows versions like WinXP or so?).

11.39 CDROM Disk Image/Containers ECM

ECM (Error Code Modeler by Neill Corlett) is a utility that removes unneccessary ECC error correction and EDC error detection values from CDROM-images. This is making the images a bit smaller, but the real size reduction isn't gained until subsequently compressing the images via tools like ZIP. Accordingly, these files are extremly uncomfortable to use: One most first UNZIP them, and then UNECM them.

.EXT.ECM - Double extension

ECM can be applied to various CDROM-image formats (like .BIN, .CDI, .IMG, .ISO, .MDF, .NRG), as indicated by the double-extension. Most commonly it's applied to .BIN files (hence using extension .BIN.ECM).

Example / File Structure

Type/Length Byte(s)

Type/Length is encoded in 1..5 byte(s), with "More=1" indicating that further length byte(s) follow:

Length=FFFFFFFFh=End Indicator
The actual decompression LEN is: "LEN=Length+1"

ECM Decompression

Below is repeated LEN times (with LEN being the Length value plus 1):

Type 1-3 are reconstructing the missing bytes before saving. Type 2-3 are saving only 920h bytes, so (if the original image contained full 930h byte sectors) the missing 10h bytes must be inserted via Type 0. Type 0 can be also used for copying whole sectors as-is (eg. Audio sectors, or Data sectors with invalid Sync/Header/ECC/EDC values). And, Type 0 can be used to store non-sector data (such like the chunks at the end of .NRG or .CDI files).

Central Mistakes

There's a lot of wrong with the ECM format. The two central problems are that it doesn't support data-compression (and needs external compression tools like zip/rar), and, that it doesn't contain a sector look-up table (meaning that random access isn't possible unless when scanning the whole file until reaching the desired sector).

Worst-case Scenario

As if ECM as such wouldn't be uncomfortable enough, you may expect typical ECM users to get more things messed up. For example:

11.40 CDROM Subchannel Images

SBI (redump.org)

SBI Files start with a 4-byte FileID:

The followed by entries as so:

Note: The PSX libcrypt protection relies on bad checksums (Q10..Q11), which will cause the PSX cdrom controller to ignore Q0..Q9 (and to keep returning position data from most recent sector with intact checksum).
Ironically, the SBI format cannot store the required Q10..Q11 checksum. The trick for using SBI files with libcrypted PSX discs is to ignore the useless Q0..Q9 data, and to assume that all sectors in the SBI file have wrong Q10..Q11 checksums.

M3S (Subchannel Q Data for Minute 3) (ePSXe)

M3S files are containing Subchannel Q data for all sectors on Minute=03 (the region where PSX libcrypt data is located) (there is no support for storing the (unused) libcrypt backup copy on Minute=09). The .M3S filesize is 72000 bytes (60 seconds * 75 sectors * 16 bytes). The 16 bytes per sector are:

Unfortunately, there are at least 3 variants of the format:

The third variant is definetly corrupt (and one should ignore such zerofilled entries). The second variant is corrupt, too (but one might attempt to repair them by guessing the missing checksum: if it contains normal position values assume correct crc, if it contains uncommon values assume a libcrypted sector with bad crc).
The M3S format is intended for libcrypted PSX games, but, people seem to have also recorded (corrupted) M3S files for unprotected PSX games (in so far, more than often, the M3S files might cause problems, instead of solving them).
Note: The odd 16-byte format with 4-byte padding does somehow resemble the "P and Q Sub-Channel" format 'defined' in MMC-drafts; if the .M3S format was based on the MMC stuff: then the 16th byte might contain a Subchannel P "pause" flag in bit7.

CDROM Images with Subchannel Data

Most CDROM-Image formats can (optionally) contain subchannel recordings. The downsides are: Storing all 8 subchannels for a full CDROM takes up about 20MBytes. And, some entries may contain 'wrong' data (read errors caused by scratches cannot be automatically repaired since subchannels do not contain error correction info).
If present, the subchannel data is usually appended at the end of each sector in the main binary file (one exception is CloneCD, which stores it in a separate .SUB file instead of in the .IMG file).

Interleaved Subchannel format (eg. Alcohol .MDF files):

Non-Interleaved Subchannel format (eg. CloneCD .SUB files):

Non-Interleaved P-Q 10h-byte Subchannel format:

11.41 CDROM Disk Images Other Formats

.ISO - A raw ISO9660 image (can contain a single data track only)

Contains raw sectors without any sub-channel information (and thus it's restricted to the ISO filesystem region only, and cannot contain extras like additional audio tracks or additional sessions). The image should start at 00:02:00 (although I wouldn't be surprised if some \<might> start at 00:00:00 or so). Obviously, all sectors must have the same size, either 800h or 930h bytes (if the image contains only Mode1 or Mode2/Form1 sectors then 800h bytes would usually enough; if it contains one or more Mode2/

Form2 sectors then all sectors should be 930h bytes).
Handling .ISO files does thus require to detect the image's sector size, and to search the sector that contains the first ISO Volume Descriptor. In case of 800h byte sectors it may be additionally required to detect if it is a Mode1 or Mode2/Form1 image; for PSX images (and any CD-XA images) it'd be Mode2.

.CHD

Something used by MAME/MESS. Originally intended for compressed ROM-images, but does also support compressed CDROM-images. Fileformat and compression ratio are unknown. Also unknown if it allows random-access.
Some info can be found in MAME source code (looking at the source code for the CHDMAN tool might be a good starting point... although the actual file structure might be hidden in other source files).

.C2D

Something. Can contain compressed or uncompressed CDROM-images. Fileformat and compression ratio are unknown. Also unknown if it allows random-access.
Some info on (uncompressed) .C2D files can be found in libmirage source code.

.ISZ

This is reportedly a "zipped" .ISO file, using some unknown compression format (unrelated to pkzip .zip file format).

.MDX

Reportedly a compressed MDS/MDF file, supported by Daemon Tools. Fileformat and compression ratio are unknown. Also unknown if it allows random-access.

.CU2/.BIN

Custom format used by PSIO (an SD-card based CDROM-drive emulator connected to PSX expansion port). The .CU2 file is somewhat intended to be smaller and easier to parse than normal .CUE files, the drawback is that it's kinda non-standard, and doesn't support INDEX and ADSR information. A sample .CUE file looks as so:

All track numbers and MM:SS:FF values are decimal. The ASCII strings should be as shown above, but they are simple ignored by the PSIO firmware (eg. using "popcorn666" instead of "size" or "track02" should also work). The first track should be marked "data1", but PSIO ignores that string, too (it does always treat track 1 as data, and track 2-99 as audio; thus not supporting PSX games with multiple data tracks). The "trk end" value should be equal to the "size" value plus 4 seconds (purpose is unknown, PSIO does just ignore the "trk end" value).
CU2 creation seems to require CDROM images in "CUE/BIN redump.org format" (with separate BIN files for each track), the CUE is then converted to a CU3 file (which is used only temporarily), until the whole stuff is finally converted to a CU2 file (and with all tracks in a single BIN file). Tools like RD2PSIO (aka redump2psio) or PSIO's own SYSCON.ZIP might help on doing some of those steps automatically.
Alongsides, PSIO uses a "multidisc.lst" file... for games that require more than one CDROM disc?

CD Image File Format (Xe - Multi System Emulator)

This is a rather crude file format, used only by the Xe Emulator. The files are meant to be generated by a utility called CDR (CD Image Ripper), which, in practice merely displays an "Unable to read TOC." error message.
The overall file structure is, according to "Xe User's Manual":

The header "definition" from the "Xe User's Manual" is as unclear as this:

Unknown if MM:SS:FF values and/or First+Last Track numbers are BCD or non-BCD.
Unknown if Last track is separately defined even if there is only ONE track.
Unknown if Track 2 and up include ADR/Control (and if yes: where?).
Unknown if ADR/Control is really meant to be \<before> MM:SS:FF on Track 1.
Unknown if ADR/Control is really meant to be \<after> MM:SS:FF on Last+Lead-Out.
Unknown if this format does have a file extension (if yes: which?).
Unknown if subchannel data is meant to be interleaved or not.
The format supports only around max 62 tracks (in case each track is 4 bytes).
There is no support for "special" features like multi-sessions, cd-text.

12. Controllers and Memory Cards

Controllers/Memory Cards

Controllers

Memory Cards

Pocketstation (Memory Card with built-in LCD screen and buttons)

Pinouts

Pinouts - Controller Ports and Memory-Card Ports

12.1 Controller and Memory Card I/O Ports

1F801040h JOY_TX_DATA (W)

Writing to this register starts the transfer (if, or as soon as TXEN=1 and JOY_STAT.2=Ready), the written value is sent to the controller or memory card, and, simultaneously, a byte is received (and stored in RX FIFO if JOY_CTRL.1 or JOY_CTRL.2 is set).
The "TXEN=1" condition is a bit more complex: Writing to SIO_TX_DATA latches the current TXEN value, and the transfer DOES start if the current TXEN value OR the latched TXEN value is set (ie. if TXEN gets cleared after writing to SIO_TX_DATA, then the transfer may STILL start if the old latched TXEN value was set).

1F801040h JOY_RX_DATA (R)

A data byte can be read when JOY_STAT.1=1. Data should be read only via 8bit memory access (the 16bit/32bit "preview" feature is rather unusable, and usually there shouldn't be more than 1 byte in the FIFO anyways).

1F801044h JOY_STAT (R)

1F801048h JOY_MODE (R/W) (usually 000Dh, ie. 8bit, no parity, MUL1)

1F80104Ah JOY_CTRL (R/W) (usually 1003h,3003h,0000h)

Caution: After slot selection (via Bits 1,13), one should issue a delay before sending the the first data byte: Digital Joypads may work without delay, Dualshock and Mouse require at least some small delay, and older Analog Joypads require a huge delay (around 500 clock cycles for SCPH-1150), official kernel waits more than 2000 cycles (which is much more than needed).

1F80104Eh JOY_BAUD (R/W) (usually 0088h, ie. circa 250kHz, when Factor=MUL1)

Timer reload occurs when writing to this register, and, automatically when the Baudrate Timer reaches zero. Upon reload, the 16bit Reload value is multiplied by the Baudrate Factor (see 1F801048h.Bit0-1), divided by 2, and then copied to the 21bit Baudrate

Timer (1F801044h.Bit11-31). The 21bit timer decreases at 33MHz, and, it ellapses twice per bit (once for CLK=LOW and once for CLK=HIGH).

The default BAUD value is 0088h (equivalent to 44h cpu cycles), and default factor is MUL1, so CLK pulses are 44h cpu cycles LOW, and 44h cpu cycles HIGH, giving it a transfer rate of circa 250kHz per bit (33MHz divided by 88h cycles).
Note: The Baudrate Timer is always running; even if there's no transfer in progress.

/IRQ7 (/ACK) Controller and Memory Card - Byte Received Interrupt

Gets set after receiving a data byte - that only if an /ACK has been received from the peripheral (ie. there will be no IRQ if the peripheral fails to send an /ACK, or if there's no peripheral connected at all).

I_STAT.7 is edge triggered (that means it can be acknowledge before or after acknowledging JOY_STAT.9). However, JOY_STAT.9 is NOT edge triggered (that means it CANNOT be acknowledged while the external /IRQ input is still low; ie. one must first wait until JOY_STAT.7=0, and then set JOY_CTRL.4=1) (this is apparently a hardware glitch; note: the LOW duration is circa 100 clock cycles).

/IRQ10 (/IRQ) Controller - Lightpen Interrupt

Pin8 on Controller Port. Routed directly to the Interrupt Controller (at 1F80107xh).
There are no status/enable bits in the JOY_registers (at 1F80104xh).

RX FIFO / TX FIFO Notes

The JOY registers can hold up to 8 bytes in RX direction, and almost 2 bytes in TX direction (just like the SIO registers, see there for details), however, normally only 1 byte should be in the RX/TX registers (one shouldn't send a 2nd byte until /ACK is sensed, and, since the transfer CLK is dictated by the CPU, the amount of incoming data cannot exceed 1 byte; provided that one reads received response byte after each transfer).
Unlike SIO, the JOY status register doesn't have a RX FIFO Overrun flag.

General Notes

RXEN should be usually zero (the hardware automatically enables receive when /JOYn is low). When RXEN is set, the next transfer causes data to be stored in RX FIFO even when /JOYn is high; the hardware automatically clears RXEN after the transfer.
For existing joypads and memory cards, data should be always transferred as 8bit no parity (although the JOY registers do support parity just like SIO registers).

Plugging and Unplugging Cautions

During plugging and unplugging, the Serial Data line may be dragged LOW for a moment; this may also affect other connected devices because the same Data line is shared for all controllers and memory cards (for example, connecting a joypad in slot 1 may corrupt memory card accesses in slot 2).
Moreover, the Sony Mouse does power-up with /ACK=LOW, and stays stuck in that state until it is accessed at least once (by at least sending one 01h byte to its controller port); this will also affect other devices (as a workaround one should always access BOTH controller ports; even if a game uses only one controller, and, code that waits for /ACK=HIGH should use timeouts).

Emulation Note

After sending a byte, the Kernel waits 100 cycles or so, and does THEN acknowledge any old IRQ7, and does then wait for the new IRQ7. Due to that bizarre coding, emulators can't trigger IRQ7 immediately within 0 cycles after sending the byte.

12.2 Controller and Memory Card Misc

BIOS Functions

Controllers can be probably accessed via InitPad and StartPad functions,
BIOS Joypad Functions
Memory cards can be accessed by the filesystem (with device names "bu00:" (slot1) and "bu10:" (slot2) or so). Before using that device names, it seems to be required to call InitCard, StartCard, and _bu_init (?).

Connectors

The PlayStation has four connectors (two controllers, two memory cards),

The controller ports have 9 pins, the memory cards only 8 pins. However, there are only 10 different pins in total.

Most of these pins are shared for all 4 connectors (eg. a CLK signal meant to be sent to one device will also arrive at the other 3 devices).
The /JOYn signals are selecting BOTH the corresponding controller, and the corresponding memory card (whether it is a controller access or memory card access depends on the first byte transferred via the CMD line; this byte should be 01h=Controller, or 81h=Memory Card; or, a special case would be 21h=Yaroze Access Card).

Data In/Out

The data is transferred in units of bytes, via separate input and output lines. So, when sending byte, the hardware does simultaneously receive a response byte.
One exception is the first command byte (which selects either the controller, or the memory card) until that byte has been sent, neither the controller nor memory card are selected (and so the first "response" byte should be ignored; probably containing more or less stable high-z levels).
The other exception is, when you have send all command bytes, and still want to receive further data, then you'll need to send dummy command bytes (should be usually 00h) to receive the response bytes.

12.3 Controller and Memory Card Signals

Overview

Top command. First comminucation(device check)

X = none, - = Hi-Z

* 0x81 is memory-card, 0x01 is standard-pad at top command.
* serial data transfer is LSB-First format.
* data is down edged output, PSX is read at up edge in shift clock.
* PSX expects No-connection if not returned Acknowledge less than 100 usec.
* clock pulse is 250KHz.
* no need Acknowledge at last data.
* Acknowledge signal width is more than 2 usec.
* time is 16msec between SEL from previous SEL.
* SEL- for memory card in PAD access.

12.4 Controller and Memory Card Multitap Adaptor

SCPH-1070 (Multitap)

The Multitap is an external adaptor that allows to connect 4 controllers, and 4 memory cards to one controller port. When using two adaptors (one on each slot), up to 8 controllers and 8 memory cards can be used.

The Multitap has a physical lip that blocks access to the memory card slot above the controller port it is plugged into. This is done to prevent ambiguity/contention between the memory card plugged into Multitap slot A and the memory card slot, since both are accessed via byte 0x81.

Multitap Controller Passthrough Mode

In passthrough mode, the access byte sent to the Multitap is 0Nh, where N is the desired pad. On receiving a valid access byte, the multitap will send a 01h access byte to the controller to activate it for transfer. All following bytes are passed through normally as if the software is communicating directly with the controller. If the initial Multitap access byte is out of range (i.e. not 01h, 02h, 03h, or 04h) or a controller is not plugged into the port, the Multitap will not ACK the access.

Multitap Controller Buffered Transfer All Mode

To activate the buffered Transfer All mode, 01h must be sent as the third byte in the controller transfer sequence (so the second byte in the command halfword), like so:

Sending any value other than 00h or 01h for that byte, in either operation mode, seems to cause the Multitap to end communication at that byte (no ACK). When activating buffered Transfer All mode, the current response is not affected. Instead, it signals the Multitap to send/receive data for all the controllers on the *next* transfer sequence, like so:

With this method, the Multitap always sends 4 halfwords per slot, padded with FFh for devices like Digital Controllers and Mice which use less than 4 halfwords. For Empty slots, all 4 halfwords are filled with FFh bytes.

A number of things can cause the Transfer All mode to fail. First, the controller corresponding to the access byte must be present, or the transfer will end on the access byte, same as for a passthrough transfer. Secondly, the command halfword must be 42h 00h or 42h 01h, or transfer will end on the second byte of the halfword.

In Transfer All mode, controller response are buffered by an entire transfer sequence, so software must request another Transfer All to receive the results from the previous Transfer All, like so:

The IRQ10 line (used by the Konami Justifier) is simply wired to all four slots via small resistors without any special logic for activating/deactivating the IRQ on specific slots.

Multitap Memory Card Access

Memory card access is done like so:

There is no other known method for accessing memory cards through the Multitap.

Unverified notes from Nocash:
That's the way how its done in Silent Hill. Although for the best of confusion, it doesn't actually work in that game (probably the developer has just linked in the multitap library, without actually supporting the multitap at higher program levels).

Multitap Games

Most Multitap games supporting up to 4 or 5 controllers require the device to be plugged into Port 1, but a small number of games strangely require the device to be plugged into Port 2 instead.

Multitap Versions

The cable connects to one of the PSX controller ports (which also carries the memory card signals). The PSX memory card port is left unused (and is blocked by a small edge on the Multitap's plug).

Software-parsed Controller IDs

Halfword 0 is parsed (by the BIOS) as usually, ie. the LSB is moved to MSB, and LSB is replaced by status byte (so ID 5A80h becomes 8000h=Multitap/okay, or xxFFh=bad). Halfwords 1,5,9,13 are NOT parsed (neither by the BIOS nor by the Multitap hardware), however, some info in the internet is hinting that Sony's libraries might be parsing these IDs too (so for example 5A41h would become 4100h=DigitalPad/okay, or xxFFh=bad).

Power Supply

The Multitap is powered by the PSX controller port. Unknown if there are any power supply restrictions (up to eight controllers and eight cards may scratch some limits, especially when doing things like activating rumble on all joypads). However, the Multitap hardware itself doesn't do much on supply restrictions (+3.5V is passed through something; maybe some fuse, loop, or 1 ohm resistor or so) (and +7.5V is passed without any restrictions).

See also

Pinouts - Component List and Chipset Pin-Outs for Multitap, SCPH-1070

12.5 Controllers - Communication Sequence

Controller Communication Sequence

The TAP byte should be usually zero, unless one wants to activate Multitap (multi-player mode), for details, see
Controller and Memory Card Multitap Adaptor
The two MOT bytes are meant to control the rumble motors (for normal non-rumble controllers, that bytes should be 00h), however, the MOT bytes have no effect unless rumble is enabled via config commands, for details, see
Controllers - Rumble Configuration

Controller ID (Halfword Number 0)

Known 16bit ID values are:

12.6 Controllers - Standard Digital/Analog Controllers

Standard Controllers

Analog Mode Note

On power-up, the controllers are in digital mode (with analog inputs disabled). Analog mode can be (de-)activated manually by pushing the Analog button. Alternately, analog mode can be (de-)activated by software via rumble configuration commands (though that's supported only on newer pads; those with two rumble motors).
The analog sticks are mechanically restricted to a "circular field of motion" (most joypads can reach "min/max" values only in "straight" horizontal or vertical directions, but not in "diagonal" directions).

Analog Joypad Range

Example min/center/max values for three different pads:

Values may vary for other pads and/or different temperatures.

Dual Analog Pad in LED=Green Mode

Basically same as normal analog LED=Red mode, with following differences:

Concerning the button names, the real analog-stick does NOT have re-arranged buttons (eg. it's L1 button is in bit10), however, concerning the button locations, the analog stick's buttons are arranged completely differently as on analog pads (so it might be rather uncomfortable to play analog stick games on analog pads in LED=Red mode; the LED=Green mode is intended to solve that problem).
Might be useful for a few analog-stick games like MechWarrior 2, Ace Combat 2, Descent Maximum, and Colony Wars. In most other cases the feature is rather confusing (that's probably why the LED=Green mode wasn't implemented on the Dual Shock).

See also

Pinouts - Component List and Chipset Pin-Outs for Digital Joypad, SCPH-1080
Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-1150
Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-1200
Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-110

12.7 Controllers - Mouse

Sony Mouse Controller

Sony Mouse Hardware Bug on Power-On

On Power-on (or when newly connecting it), the Sony mouse does draw /ACK to LOW on power-on, and does then hold /ACK stuck in the LOW position.
For reference: Normal controllers and memory cards set /ACK=LOW only for around 100 clk cycles, and only after having received a byte from the console.
The /ACK pin is shared for both controllers and both memory cards, so the stuck /ACK is also "blocking" all other connected controllers/cards. To release the stuck /ACK signal: Send a command (at least one 01h byte) to both controller slots.

Sony Mouse Compatible Games

Note: There are probably many more mouse compatible games.
Plus: Dracula - The Resurrection

Sony Mouse Component List

PCB "TD-T41V/\, MITSUMI"
Component Side:

Solder/SMD Side:

Cable:

RS232 Mice

Below is some info on RS232 serial mice. That info isn't directly PSX related as the PSX normally doesn't support those mice.
With some efforts, one can upgrade the PSX SIO port to support RS232 voltages, and with such a modded console one could use RS232 mice (in case one wants to do that).
The nocash PSX bios can map a RS232 mouse to a spare controller slot (thereby simulating a Sony mouse), that trick may work with various PSX games.

Standard Serial Mouse

A serial mouse should be read at 1200 bauds, 7 data bits, no parity, 1 stop bit (7N1) with DTR and RTS on. For best compatibility, the mouse should output 2 stop bits (so it could be alternately also read as 7N2 or 8N1). When the mouse gets moved, or when a button gets pressed/released, the mouse sends 3 or 4 characters:

Additionally, the mouse outputs a detection character (when switching RTS (or DTR?) off and on:

Normally, the detection response consist of a single character (usually "M"), though some mice have the "M" followed by 11 additional characters of garbage or version information (these extra characters have bit6=0, so after detection, one should ignore all characters until receiving the first data character with bit6=1).

Mouse Systems Serial Mouse (rarely used)

Accessed at 1200 bauds, just like standard serial mouse, but with 8N1 instead 7N1, and with different data bytes.

The strange duplicated 8bit motion values are usually simply added together, ie. X=X1+X2 and Y=Y1+Y2, producing 9bit motion values.

Notes

The Sony Mouse connects directly to the PSX controller port. Alternately serial RS232 mice can be connected to the SIO port (with voltage conversion adaptor) (most or all commercial games don't support SIO mice, nor does the original BIOS do so, however, the nocash BIOS maps SIO mice to unused controller slots, so they can be used even with commercial games; if the game uses BIOS functions to read controller data).
Serial Mice (and maybe also the Sony mouse) do return raw mickeys, so effects like double speed threshold must (should) be implemented by software. Mice are rather rarely used by PSX games. The game "Perfect Assassin" includes ultra-crude mouse support, apparently without threshold, and without properly matching the cursor range to the screen resolution.

12.8 Controllers - Racing Controllers

neGcon Racing Controller (Twist) (NPC-101/SLPH-00001/SLEH-0003)

The Twist controller works like a paddle or steering wheel, but doesn't have a wheel or knob, instead, it can be twisted: To move into one direction (=maybe right?), turn its right end away from you (or its left end towards you). For the opposite direction (=maybe left?), do it vice-versa.

Namco Volume Controller (a paddle with two buttons) (SLPH-00015)

This is a cut-down variant of the neGcon, just a featureless small box. It does have the same ID value as neGcon (ID=5A23h), but, it excludes most digital, and all analog buttons.

SANKYO N.ASUKA aka Nasca Pachinco Handle (SLPH-00007)

Another cut-down variant of the neGcon (with ID=5A23h, too). But, this one seems to have only one button. Unlike Namco's volume controller it doesn't look featureless. It looks pretty much as shown in the ascii-arts image below. Seems to be supported by several irem titles. No idea what exactly it is used for, it's probably not a sewing machine controller, nor an electronic amboss.

Mad Catz Steering Wheel (SLEH-0006)

A neGcon compatible controller. The Twist-feature has been replaced by a steering wheel (can be turned by 270 degrees), and the analog I and II buttons by foot pedals. The analog L button has been replaced by a digital button (ie. in neGcon mode, the last byte of the controller data can be only either 00h or FFh). When not using the pedals, the I/II buttons on the wheel can be used (like L button, they aren't analog though).

Unlike the neGon, the controller has Select, >\< and [] buttons, and a second set of L/R buttons (at the rear-side of the wheel) (no idea if L1/R1 or L2/R2 are at front?). Aside from the neGcon mode, the controller can be also switched to Digital mode (see below for button chart).

MadCatz Dual Force Racing Wheel

Same as above, but with a new Analog mode (additionally to Digital and neGcon modes). The new mode is for racing games that support only Analog Joypads (instead of neGcon). Additionally it supports vibration feedback.

MadCatz MC2 Vibration compatible Racing Wheel and Pedals

Same as above, but with a redesigned wheel with rearranged buttons, the digital pad moved to the center of the wheel, the L/R buttons at the rear-side of the wheel have been replaced by 2-way butterfly buttons ("pull towards user" acts as normal, the new "push away from user" function acts as L3/R3).

MadCatz Button Chart

Whereas, lt/rt/up/dn=Digital Pad, UP/DN=Left Analog Pad Up/Down, LT/RT=Right Analog Pad Left/Right. Analog mode is supported only by the Dual Force and MC2 versions, L3/R3 only by the MC2 version.

Namco Jogcon (NPC-105/SLEH-0020/SLPH-00126/SLUH-00059)

Rotations of the dial are recognized by an optical sensor (so, unlike potentiometers, the dial can be freely rotated; by more than 360 degrees). The dial is also connected to a small motor, giving it a real force-feedback effect (unlike all other PSX controllers which merely have vibration feedback). Although that's great, the mechanics are reportedly rather cheap and using the controller doesn't feel too comfortable. The Jogcon is used only by Ridge Racer 4 for PS1 (and Ridge Racer 5 for PS2), and Breakout - Off the Wall Fun.
The Mode button probably allows to switch between Jogcon mode and Digital Pad mode (similar to the Analog button on other pads), not sure if the mode can be also changed by software via configuration commands...? Unknown how the motor is controlled; probably somewhat similar to vibration motors, ie. by the M1 and/or M2 bytes, but there

must be also a way to select clockwise and anticlockwise direction)...? The controller does reportedly support config command 4Dh (same as analog rumble).

12.9 Controllers - Lightguns

There are two different types of PSX lightguns (which are incompatible with each other).

Namco Lightgun (GunCon)

Namco's Cinch-based lightguns are extracting Vsync/Hsync timings from the video signal (via a cinch adaptor) (so they are working completely independed of software timings).
Controllers - Lightguns - Namco (GunCon)

Konami Lightgun (IRQ10)

Konami's IRQ10-based lightguns are using the lightgun input on the controller slot (which requires IRQ10/timings being properly handled at software side).
Controllers - Lightguns - Konami Justifier/Hyperblaster (IRQ10)
The IRQ10-method is reportedly less accurate (although that may be just due to bugs at software side).

Third-Party Lightguns

There are also a lot of unlicensed lightguns which are either IRQ10-based, or Cinch-based, or do support both.
For example, the Blaze Scorpion supports both IRQ10 and Cinch, and it does additionally

have a rumble/vibration function; though unknown how that rumble feature is accessed, and which games are supporting it).

Lightgun Games

Controllers - Lightguns - PSX Lightgun Games

Compatibilty Notes (IRQ10 vs Cinch, PAL vs NTSC, Calibration)

Some lightguns are reportedly working only with PAL or only with NTSC games (unknown which guns, and unknown what is causing problems; the IRQ10 method should be quite hardware independed, the GunCon variant, too, although theoretically, some GunCon guns might have problems to extract Vsync/Hsync from either PAL or NTSC composite signals).
Lightguns from different manufacturers are reportedly returning slightly different values, so it would be recommended to include a calibration function in the game, using at least one calibration point (that would also resolve different X/Y offsets caused by modifying GP1 display control registers).
Lightguns are needing to sense light from the cathode ray beam; as such they won't work on regions of the screen that contain too dark/black graphics.

12.10 Controllers - Lightguns - Namco (GunCon)

GunCon Cinch-based Lightguns (Namco)

Caution: The gun should be read only shortly after begin of VBLANK.

Error/Busy Codes

Coordinates X=0001h, Y=0005h indicates "unexpected light":

Coordinates X=0001h, Y=000Ah indicates "no light", this can mean either:

To avoid the BUSY error, one should read the gun shortly after begin of VBLANK (ie. AFTER rendering, but still BEFORE vsync). Doing that isn't as simple as one might think: On a NTSC console, time between VBLANK and VSYNC is around 30000 cpu clks, reading the lightgun (or analog joypads) takes around 15000 cpu clks. So, reading two controllers within that timeframe may be problematic (and reading up to eight controllers via multitaps would be absolutely impossible). As a workaround, one may arrange the read-order to read lightguns at VBLANK (and joypads at later time). If more than one lightgun is connected, then one may need to restrict reading to only one (or maybe: max two) guns per frame.

Minimum Brightness

Below are some average minimum brightness values, the gun may be unable to return position data near/below that limits (especially coordinates close to left screen border are most fragile). The exact limits may vary from gun to gun, and will also depend on the TV Set's brightness setting.

The gun does also work with mixed colors (eg. white bold text on black background works without errors, but the returned coordinates are a bit "jumpy" in that case; returning the position of the closest white pixels).
BUG: On a plain RED screen, aiming at Y>=00F0h, the gun is randomly returning either Y, or Y-80h (that error occurs in about every 2nd frame, ie. at 50% chance). It's strange... no idea what is causing that effect.

Coordinates

The coordinates are updated in all frames (as opposed to some lightguns which do update them only when pulling the trigger).
The absolute min/max coordinates may vary from TV set to TV set (some may show a few more pixels than others). The relation of the gun's Screen Coodinates to VRAM Coordinates does (obviously) depend on where the VRAM is located on the screen; ie. on the game's GP1(06h) and GP1(07h) settings.
Vertical coordinates are counted in scanlines (ie. equal to pixels). Horizontal coordinates are counted in 8MHz units (which would equal a resolution of 385 pixels; which can be, for example, converted to 320 pixel resolution as X=X*320/385).

Misinformation (from bugged homebrew source code)

Namco Lightgun Drawing

See also

Pinouts - Component List and Chipset Pin-Outs for Namco Lightgun, NPC-103

12.11 Controllers - Lightguns - Konami Justifier/Hyperblaster (IRQ10)

Overall IRQ10-Based Lightgun Access

The purpose of the "x0h" byte is probably to enable IRQ10 (00h=off, 10h=on), this would allow to access more than one lightgun (with only one per frame having the IRQ enabled).

Standard IRQ10-based Lightguns (Konami)

The Controller Data simply consists of the ID and buttons states:

The coordinates aren't part of the controller data, instead they must be read from Timer 0 and 1 upon receiving IRQ10 (see IRQ10 Notes below).

Konami Lightgun Drawing

Konami IRQ10 Notes

The PSX does have a lightgun input (Pin 8 of the controller), but, Sony did apparently "forget" to latch the current cathode ray beam coordinates by hardware when sensing the lightgun signal (quite strange, since that'd be a simple, inexpensive, and very obvious feature for a gaming console).
Instead, the lightgun signal triggers IRQ10, and the interrupt handler is intended to "latch" the coordinates by software (by reading Timer 0 and 1 values, which must configured to be synchronized with the GPU).
That method requires IRQ handling to be properly implemented in software (basically, IRQs should not be disabled for longer periods, and DMA transfers should not block the bus for longer periods). In practice, most programmers probably don't realize how to do that, to the worst, Sony seems to have delivered a slightly bugged library (libgun) to developers.
For details on Timers, see:
Timers
In some consoles, IRQ10 seems to be routed through a Secondary IRQ Controller, see:
EXP2 DTL-H2000 I/O Ports

IRQ10 Priority

For processing IRQ10 as soon as possible, it should be assigned higher priority than all other IRQs (ie. when using the SysEnqIntRP BIOS function, it should be the first/newest element in priority chain 0). The libgun stuff assigns an even higher priority by patching the BIOS exception handler, causing IRQ10 to be processed shortly before processing the priority chains (the resulting IRQ priority isn't actually higher as when using 1st element of chain 0; the main difference is that it skips some time consuming code which pushes registers R4..R30). For details on that patch, see:
BIOS Patches
Even if IRQ10 has highest priority, execution of (older) other IRQs may cause a new IRQ10 to be executed delayed (because IRQs are disabled during IRQ handling), to avoid that problem: Best don't enable any other IRQs except IRQ0 and IRQ10, or, if you need other IRQs, best have them enabled only during Vblank (there are no scanlines drawn during vblank, so IRQ10 should never trigger during that period). DMAs might also slow down IRQ execution, so best use them only during Vblank, too.

IRQ10 Timer Reading

To read the current timer values the IRQ10 handler would be required to be called \<immediately> after receiving the IRQ10 signal, which is more or less impossible; if the main program is trying to read a mul/div/gte result while the mul/div/gte operation is still busy may stop the CPU for some dozens of clock cycles, and active DMA transfers or cache hits and misses in the IRQ handler may cause different timings, moreover, timings may become completely different if IRQs are disabled (eg. while another IRQ is processed).
However, IRQ10 does also get triggered in the next some scanlines, so the first IRQ10 is used only as a notification that the CPU should watch out for further IRQ10's. Ie. the IRQ10 handler should disable all DMAs, acknowledge IRQ10, and then enter a waitloop that waits for the IRQ10 bit in I_STAT to become set again (or abort if a timeout occurs) and then read the timers, reportedly like so:

```

```

No idea why PAL/NTSC should use different factors, that factors are looking quite silly/bugged, theoretically, the pixel-to-clock ratio should be the exactly same for PAL and NTSC...?
Mind that reading Timer values in Dotclock/Hblank mode is unstable, for Timer1 this can be fixed by the read-retry method, for Timer0 this could be done too, but one would need to subtract the retry-time to get a correct coordinate; alternately Timer0 can run at system clock (which doesn't require read-retry), but it must be then converted to video clock (mul 11, div 7), and then from video clock to dot clock (eg. div 8 for 320-pixel mode).
Above can be repeated for the next some scanlines (allowing to take the medium values as result, and/or to eliminate faulty values which are much bigger or smaller than the other values). Once when you have collected enough values, disable IRQ10, so it won't trigger on further scanlines within the current frame.

IRQ10 Bugs

BUG: The "libgun" library doesn't acknowledge the old IRQ10 \<immediately> before waiting for a new IRQ10, so the timer values after sensing the new IRQ10 are somewhat random (especially for the first processed scanline) (the library allows to read further IRQ10's in further scanlines, which return more stable results).
No idea how many times IRQ10 gets typically repeated? Sporting Clays allocates a

buffer for up to 20 scanlines (which would cause pretty much of a slowdown since the CPU is just waiting during that period) (nethertheless, the game uses only the first timer values, ie. the bugged libgun-random values).

Unknown if/how two-player games (with 2 lightguns) are working with the IRQ10 method... if IRQ10 is generated ONLY after pressing the trigger button, then it may work, unless both players have Trigger pressed at the same time... and, maybe one can enable/disable the lightguns by whatever commmand being sent to the controller (presumably that "x0h" byte, see above), so that gun 1 generates IRQ10 only in each second frame, and gun 2 only in each other frame...?

12.12 Controllers - Lightguns - PSX Lightgun Games

PSX Lightgun Games

Some games are working only with IRQ10 or only with Cinch, some games support both methods:

Note: The RPG game Dragon Quest Monsters does also contain IRQ10 lightgun code (though unknown if/when/where the game does use that code).

12.13 Controllers - Rumble Configuration

Rumble (aka "Vibration Function") is basically controlled by two previously unused bytes of the standard controller Read command.
There are two methods to control the rumble motors, the old method is very simple (but supports only one motor), the new method envolves a bunch of new configuration commands (and supports two motors).

Old Method, one motor, no config commands (SCPH-1150, SCPH-1200, SCPH-110)

The SCPH-1150 doesn't support any special config commands, instead, rumble is solely done via the normal joypad read command:

The rumble motor is simply controlled by three bits in the xx/yy bytes:

The motor control is digital on/off (no analog slow/fast), recommended values would be yyxx=0140h=on, and yyxx=0000h=off.
LED state is don't care (rumble works with led OFF, RED, and GREEN). In absence of config commands, the LED can be controlled only manually (via Analog button), the current LED state is implied in the controller "id" byte.
For backwards compatibility, the above old method does also work on SCPH-1200 and SCPH-110 (for controlling the right/small motor), alternately those newer pads can use the config commands (for gaining access to both motors).

When the controller enters configuration mode, the old rumble method is disabled until a hard reset. Unknown if it's possible to re-enable the old rumble method programmatically.

New Method, two motors, with config commands (SCPH-1200, SCPH-110)

For using the new rumble method, one must unlock the new rumble mode, for that purpose Sony has invented a "slightly" overcomplicated protocol with not less than 16 new commands (the rumble relevant commands are 43h and 4Dh, also, command 44h may be useful for activating analog inputs by software, and, once when rumble is unlocked, command 42h is used to control the rumble motors). Anyways, here's the full command set...

Normal Mode

Configuration Mode

Normal Mode - Command 42h "B" - Read Buttons (and analog inputs when enabled)

The normal read command, see Standard Controller chapter for details on buttons and analog inputs. The xx/yy bytes have effect only if rumble is unlocked; use Command

43h to enter config mode, and Command 4Dh to unlock rumble. Command 4Dh has billions of combinations, among others allowing to unlock only one of the two motors, and to exchange the xx/yy bytes, however, with the default values, xx/yy are assigned like so:

The Left/Large motor starts spinning at circa min=50h..60h, and, once when started keeps spinning downto circa min=38h. The exact motor start boundary depends on the current position of the weight (if it's at the "falling" side, then gravity helps starting), and also depends on external movements (eg. it helps if the user or the other rumble motor is shaking the controller), and may also vary from controller to controller, and may also depend on the room temperature, dirty or worn-out mechanics, etc.

Normal Mode - Command 43h "C" - Enter/Exit Configuration Mode

When issuing command 43h from inside normal mode, the response is same as for command 42h (button data) (and analog inputs when in analog mode) (but without M1 and M2 parameters). While in config mode, the ID bytes are always "F3h 5Ah" (instead of the normal analog/digital ID bytes).

Entering configuration mode enables the DualShock rumble method and disables the legacy rumble method. Unknown if it's possible to re-enable the old rumble method programmatically.

Caution: Additionally to activating configuration commands, entering config mode does also activate a Watchdog Timer which does reset the controller if there's been no communication for about 1 second or so. The watchdog timer remains active even when returning to normal mode via Exit Config command. The reset does disable and lock rumble motors, and switches the controller to Digital Mode (with LED=off, and analog inputs disabled). To prevent this, be sure to keep issuing joypad reads even when not needing user input (eg. while loading data from CDROM).
Caution 2: A similar reset occurs when the user pushes the Analog button; this is

causing rumble motors to be stopped and locked, and of course, the analog/digital state gets changed.
Caution 3: If config commands were used, and the user does then push the analog button, then the 5Ah-byte gets replaced by 00h (ie. responses change from "HiZ id 5Ah ..." to "HiZ id 00h ...").

Config Mode - Command 42h "B" - Read Buttons AND analog inputs

Same as command 42h in normal mode, but with forced analog response (ie. analog inputs and L3/R3 buttons are returned even in Digital Mode with LED=Off).

Config Mode - Command 43h "C" - Enter/Exit Configuration Mode

Equivalent to command 43h in normal mode, but returning 00h bytes rather than button data, can be used to return to normal mode.

Back in normal mode, the rumble motors (if they were enabled) can be controlled with normal command 42h. Some controller revisions will only change config mode state upon receiving the entire command sequence while others will set it immediately on receiving the xx byte. So while aborting the command sequence early (i.e. setting JOY_CTRL to 0 after sending xx, which some homebrew software does) may work for setting config mode state on some controllers, in practice this technique is undefined behavior and use should be avoided.

Config Mode - Command 44h "D" - Set LED State (analog mode on/off)

If "sel=02h", then "val" is applied as new LED state (val=00h for LED off aka Digital mode, and val=01h for LED on/red aka analog mode).

Using this command has a side effect of resetting the DualShock rumble configuration to FFh FFh FFh FFh FFh FFh.

Config Mode - Command 45h "E" - Get LED State (and whatever values)

Returns val=00h for LED off, and val=01h for LED on/red. The other values might indicate the number of rumble motors, analog inputs, or version information, or so.

Config Mode - Command 46h "F" - Get Variable Response A

Purpose unknown, but seems to be used by some games to identify the controller as a DualShock Analog Controller since Jogcon returns different values. Response varies: If xx=00h then yy=01h,02h,00h,0ah, else if xx=01h then yy=01h,01h,01h,14h. Other values of xx will cause yy = 00h 00h 00h 00h to be returned. Unknown what happens if the other values after xx are modified, but it is likely that the controller will default to yy = 00h 00h 00h 00h.

Config Mode - Command 47h "G" - Get whatever values

Purpose unknown, but seems to be used by some games to identify the controller as a DualShock Analog Controller since Jogcon returns different values. Modifications to the sequence being sent to the controller seem to cause the controller to respond with 00h 00h 00h 00h 00h 00h.

Config Mode - Command 4Ch "L" - Get Variable Response B

Purpose unknown, but as with 46h and 47h, may be used by some games to identify the controller as a DualShock Analog Controller. Response varies: If xx=00h then yy=04h, else if xx=01h then yy=07h. All other xx values and sequences seem to have a 00h 00h 00h 00h 00h 00h response.

Config Mode - Command 4Dh "M" - Set Rumble Configuration (and select response length)

Nocash's original notes state that this command unlocks the rumble motors -- however, based on hardware testing, the DualShock rumble mode is unlocked as soon as a controller enters configuration mode. This command just sets the DualShock rumble configuration.

The rumble configuration works as follows: The last 00h byte in the rumble configuration sequence determines which byte in the 42h pad read command will be used for setting the small motor strength. The last 01h byte in the sequence similarly determines which byte will be used for the large motor strength. If either 00h or 01h or both are not present in the configuration sequence, the corresponding motor will not be able to receive any rumble, as expected.

Most games will use 00h 01h FFh FFh FFh FFh, however a number of games will use uncommon sequences. For example, Final Fantasy VIII uses FFh 00h 01h FFh FFh FFh while Armored Core Project Phantasma uses 00h 00h 01h 01h FFh FFh.

Untested Nocash notes: The extra halfword(s) increase the transfer length by 1 or 2 halfwords (and accordingly, the digital mode ID changes from 41h to 42h or 43h), the controller returns 00h bytes for the extra halfwords, in analog mode, the ID and transfer length remains unchanged (since it always has extra halfwords for analog responses). So, the extra halfwords seem to be output to the controller (rather than response data), possibly intended to control additional rumble motors or to output data to other hardware features.

Config Mode - Command 48h "H" - Unknown (response HiZ F3h 5Ah 4x00h 01h 00h)

This command does return a bunch of 00h bytes, and one 01h byte. Purpose unknown. The command does not seem to be used by any games.

Config Mode - Command 40h "@" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 41h "A" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 49h "I" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 4Ah "J" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 4Bh "K" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 4Eh "N" - Unknown (response HiZ F3h 5Ah 6x00h)

Config Mode - Command 4Fh "O" - Unknown (response HiZ F3h 5Ah 6x00h)

These commands do return a bunch of 00h bytes. Purpose unknown. These commands do not seem to be used by any games.

Note

Rumble is a potentially annoying feature, so games that do support rumble should also include an option to disable it.

12.14 Controllers - Dance Mats

PSX Dance Mats are essentially normal joypads with uncommonly arranged buttons, the huge mats are meant to be put on the floor, so the user could step on them.

Dance Mat vs Joypad Compatibility

There are some differences to normal joypads: First of, the L1/L2/R1/R2 shoulder buttons are missing in most variants. And, the mats are allowing to push Left+Right and Up+Down at once, combinations that aren't mechanically possible on normal joypads (some dancing games do actually require those combinations, whilst some joypad games may get confused on them).

Dance Mat Unknown Things

Unknown if the mat was sold in japan, and if so, with which SLPH/SCPH number.
Unknown if the mat's middle field is also having a button assigned.
Unknown if the mat is having a special controller ID, or if there are other ways to detect mats (the mats are said to be compatible with skateboard games, so the mats are probably identifying themselves as normal digital joypad; assuming that those skateboard games haven't been specifically designed for mats).

Dance Mat Games

The mats can be reportedly also used with whatever skateboard games.

Dance Mat Variants

There is the US version (DDR Dance Pad, SLUH-00071), and a slightly different European version (Official Dance Mat, SLEH-00023: shiny latex style with perverted colors, and Start/Select arranged differently). The japanese version (RU017) resembles the US version, but without Triangle/Square symbols drawn in lower left/right edges. And there is a handheld version (with additional L1/L2/R2/R1 buttons; maybe unlicensed; produced as MINI DDR, and also as Venom Mini Dance Pad).

Stay Cool?

Despite of the "Stay Cool!" slogan, the mat wasn't very cool - not at all! It offered only two steps back-and-forth, and also allowed to do extremly uncool side-steps. Not to mention that it would melt when dropping a burning cigarette on it. Stay Away!

12.15 Controllers - Pop'n Controllers

Controllers used for Konami's Pop'n Music series. At least a few different versions of the controller (Pop'n Controller, Pop'n Controller 2, larger arcade-size version, possibly others and in different color variations) have been released for the PS1 and PS2. Unknown if the controllers released in the PS2 era have any additional commands not present in the original Pop'n Controller, but they are supposedly fully compatible with PS1 Pop'n Music games.

Pop'n Controllers report as digital controllers (ID byte 41h), but the left, right, and down d-pad controls are not connected to any physical buttons and are always reported as pressed (in the first transferred button byte, bits 5-7 are always 0). Pop'n Music games check these bits to determine if a Pop'n Controller is connected and will change the in-game controls accordingly if so.

12.16 Controllers - Densha de Go! / Jet de Go! Controllers

Controllers used for Taito's Densha de Go! and Jet de Go! series. Unknown what method is being used by Densha de Go! and Jet de Go! games for detecting these controllers.

- The workings of Densha de Go! PSX controllers have been extensively researched in the ddgo-controller-docs repo.
- The Jet de Go! PSX controller comes in gray and black color. It seems to work the same as an analog controller and supports vibration. The steering wheel is mapped to the left stick (wheel rotation as horizontal, wheel raise/lower as vertical axis). The thrust throttle seems mapped to the right stick Y-axis full range (so half throttle matches vertically centered right stick).

12.17 Controllers - Fishing Controllers

The fishing rods are (next to lightguns) some of the more openly martial playstation controllers - using the credo that "as long as you aren't using dynamite: it's okay to kill them cause they don't have any feelings."

PSX Fishing Controller Games

For more see: http://www.gamefaqs.com/ps/list-109 (sports->nature->fishing)

Logos on CD Covers

US Fishing games should have a "SLUH-00063" logo. European Fishing games don't have any fishing logos; apparently fishing controllers haven't been officially released/ supported in Europe.
Japanese Fishing games can have a bunch of logos: Usually BANC-0001 or SLPH-00100 (or both).
Moreover, some japanese games have a yellow/green fishing logo with japanese text (found on Perfect Fishing: Bass Fishing, Perfect Fishing: Rock Fishing, Simple 1500 Series Vol. 29: The Tsuri, Super Bass Fishing) (unknown if that logo refer to other special hardware, or if it means the "normal" BANC-0001 or SLPH-00100 controllers.
And Moreover, some japanese games have some sort of "headset" logos with japanese text, these seem to have same meaning as SLPH-00100; as indicated by photos on CD cover of Tsuwadou Keiryuu Mizuumihen (Best Edition) (2000); that CD cover also has a "headset 2" logo, which seems to mean a newer PS2 variant of the SLPH-00100.

PSX Fishing Controllers

Of these, the ASCII/agetec controllers seem to be most popular (and most commonly supported). The Bandai contoller is also supported by a couple of games (though the Bandai controller itself seems to be quite rare). The Interact/Naki controllers are probably just clones of the ASCII/agetec ones. The Hori controller is quite rare (and with its string and plastic fish, it's apparently working completely different than the other fishing controllers).

Tech Info (all unknown)

Unknown how to detect fishing controllers.
Unknown how to read buttons, joystick, crank, motion sensors.
Unknown how to control rumble/vibration.
Unknown if/how Bandai differs from ASCII/agetec (aside from less buttons).
Unknown how the Hori thing works.

ASCII SLPH-00100 / agetec SLUH-00063 (silver)

Bandai BANC-0001 (dark gray/blue)

Hori HPS-97 / HPS-98 (black/gray)

12.18 Controllers - Stepper Controller

A controller in the form of a stepper machine. Comes in an orange variant (TW-20001) and a blue variant (TW-20002). Was available for purchase as a standalone controller or as a bundle with Happy Jogging in Hawaii. Requires an analog controller to be connected to the stepper in order to work. The stepper forces the analog controller to switch to analog mode. With some (cheap?) third-party Playstation controllers, this might actually result in permanently activating the controller's rumble. The stepper overwrites the left stick X-axis values with the position of the stepper pedals: - Left stepper pedal fully down equals to left stick fully to left - Right stepper pedal fully down equals to left stick fully to right - Left and right stepper pedal equal height matches left stick horizontally centered Further details unknown.

12.19 Controllers - I-Mode Adaptor (Mobile Internet)

The I-Mode Adaptor cable (SCPH-10180) allows to connect an I-mode compatible mobile phone to the playstation's controller port; granting a mobile internet connection to japanese games.

PSX Games for I-Mode Adaptor (Japan only)

The supported games should have a I-Mode adaptor logo on the CD cover (the logo depicts two plugs: the PSX controller plug, and the smaller I-Mode plug).

Note: "Dragon Quest Monsters 1 & 2" was announced/rumoured to support I-mode (however, its CD cover doesn't show any I-Mode adapter logo).

Tech Details (all unknown)

Unknown how to detect the thing, and how to do the actual data transfers.
The cable does contain a 64pin chip, an oscillator, and some smaller components (inside of the PSX controller port connector).

Hardware Variant

Keitai Eddy seems to have the phone connect to the SIO port (on rear side of the PSX, at least it's depicted like so on the CD cover). This is apparently something different than the SCPH-10180 controller-port cable. Unknown what it is exactly - probably some mobile internet connection too, maybe also using I-mode, or maybe some other protocol.

12.20 Controllers - Additional Inputs

Reset Button

PSX only (not PSone). Reboots the PSX via /RESET signal. Probably including for forcefully getting through the WHOLE BIOS Intro, making it rather useless/annoying? No idea if it clears ALL memory during reboot?

CDROM Shell Open

Status bit of the CDROM controller. Can be used to sense if the shell is opened (and also memorizes if the shell was opened since last check; allowing to sense possible disk changes).

PocketStation

Memory Card with built-in LCD screen and Buttons (which can be used as miniature handheld console). However, when it is connected to the PSX, the buttons are vanishing in the cartridge slot, so the buttons cannot be used as additional inputs for PSX games.

Serial Port PSX only (not PSone)

With an external adaptor (voltage conversion), the serial port can be used (among others) to connect a RS232 Serial Mouse. Although, most or all commercial games with mouse input are probably (?) supporting only Sony's Mouse (on the controller port) (rather than standard RS232 devices on the serial port).

TTY Debug Terminal

If present, the external DUART can be used for external keyboard input, at the BIOS side, this is supported as "std_in".

12.21 Controllers - Misc

Standard Controllers

Special Controllers

BANDAI "BANC-0002" - 4 Buttons (Triangle, Circle, Cross, Square) (nothing more)

SCPH-2000 Keyboard/Mouse adapter

A PS/2 to PSX controller port adaptor, for educational Lightspan titles.
There are two variants of the adaptor:

Joystick

12.22 Memory Card Read/Write Commands

Reading Data from Memory Card

Non-sony cards additionally send eight 5Ch bytes after the end flag.
When sending an invalid sector number, original Sony memory cards respond with FFFFh as Confirmed Address (and do then abort the transfer without sending any data, checksum, or end flag), third-party memory cards typically respond with the sector number ANDed with 3FFh (and transfer the data for that adjusted sector number).

Writing Data to Memory Card

Get Memory Card ID Command

This command is supported only by original Sony memory cards. Not sure if all sony cards are responding with the same values, and what meaning they have, might be number of sectors (0400h) and sector size (0080h) or whatever.

Invalid Commands

Transfer aborts immediately after the faulty command byte, or, occasionally after one more byte (with response FFh to that extra byte).

FLAG Byte

The initial value of the FLAG byte on power-up (and when re-inserting the memory card) is 08h.
Bit3=1 is indicating that the directory wasn't read yet (allowing to sense memory card changes). For some strange reason, bit3 is NOT reset when reading from the card, but rather when writing to it. To reset the flag, games are usually issuing a dummy write to sector number 003Fh, more or less unneccessarily stressing the lifetime of that sector.
Bit2=1 seems to be intended to indicate write errors, however, the write command seems to be always finishing without setting that bit, instead, the error flag may get set on the NEXT command.
Note: Some (not all) non-sony cards also have Bit5 of the FLAG byte set.

Timings

IRQ7 is usually triggered circa 1500 cycles after sending a byte (counted from the begin of the first bit), except, the last byte doesn't trigger IRQ7, and, after the 7th byte of the Read command, an additional delay of circa 31000 cycles occurs before IRQ7 gets triggered (that strange extra delay occurs only on original Sony cards, not on cards from other manufacturers).
There seems to be no extra delays in the Write command, as it seems, the data is written on the fly, and one doesn't need to do any write-busy handling... although, theoretically, the write shouldn't start until verifying the checksum... so it can't be done on the fly at all...?

Notes

Responses in brackets are don't care, (00h) means usually zero, (pre) means usually equal to the previous command byte (eg. the response to LSB is MSB).

Memory cards are reportedly "Flash RAM" which sounds like bullshit, might be battery backed SRAM, or FRAM, or slower EEPROM or FLASH ROM, or vary from card to card...?

12.23 Memory Card Data Format

Data Size

The memory is split into 16 blocks (of 8 Kbytes each), and each block is split into 64 sectors (of 128 bytes each). The first block is used as Directory, the remaining 15 blocks are containing Files, each file can occupy one or more blocks.

Header Frame (Block 0, Frame 0)

Directory Frames (Block 0, Frame 1..15)

Filesize [04h..07h] and Filename [0Ah..1Eh] are stored only in the first directory entry of a file (ie. with State=51h or A1h), other directory entries have that bytes zero-filled.

Filename Notes

The first some letters of the filename should indicate the game to which the file belongs, in case of commercial games this is conventionally done like so: Two character region code:

followed by 10 character game code,

where the "AAAA" part does imply the region too; (SLPS/SCPS=Japan, SLUS/SCUS=America, SLES/SCES=Europe) (SCxS=Made by Sony, SLxS=Licensed by Sony), followed by up to 8 characters,

(which may identify the file if the game uses multiple files; this part often contains a random string which seems to be allowed to contain any chars in range of 20h..7Fh, of course it shouldn't contain "?" and "*" wildcards).

Broken Sector List (Block 0, Frame 16..35)

If Block0/Frame(16+N) indicates that a given sector is broken, then the data for that sector is stored in Block0/Frame(36+N).

Broken Sector Replacement Data (Block 0, Frame 36..55)

Unused Frames (Block 0, Frame 56..62)

Write Test Frame (Block 0, Frame 63)

Reportedly "write test". Usually same as Block 0 ("MC", 253 zero-bytes, plus checksum 0Eh).

Title Frame (Block 1..15, Frame 0) (in first block of file only)

For more info on entries [50h..5Fh], see Pocketstation File Header/Icons

Icon Frame(s) (Block 1..15, Frame 1..3) (in first block of file only)

Note: The icons are shown in the BIOS bootmenu (which appears when starting the PlayStation without a CDROM inserted). The icons are drawn via GP0(2Ch) command, ie. as Textured four-point polygon, opaque, with texture-blending, whereas the 24bit blending color is 808080h (so it's quite the same as raw texture without blending). As semi-transparency is disabled, Palette/CLUT values can be 0000h=FullyTransparent, or 8000h=SolidBlack (the icons are usually shown on a black background, so it doesn't make much of a difference).

Data Frame(s) (Block 1..15, Frame N..63; N=excluding any Title/Icon Frames)

Note: Files that occupy more than one block are having only ONE Title area, and only one Icon area (in the first sector(s) of their first block), the additional blocks are using sectors 0..63 for plain data.

Shift-JIS Character Set (16bit) (used in Title Frames)

Can contain japanese or english text, english characters are encoded like so:

Titles shorter than 32 characters are padded with 00h-bytes.
Note: The titles are \<usually> in 16bit format (even if they consist of raw english text), however, the BIOS memory card manager does also accept 8bit characters 20h..7Fh (so, in the 8bit form, the title could be theoretically up to 64 characters long, but, nethertheless, the BIOS displays only max 32 chars).
For displaying Titles, the BIOS includes a complete Shift-JIS character set,
BIOS Character Sets
Shift-JIS is focused on asian languages, and does NOT include european letters (eg. such with accent marks). Although the non-japanese PSX BIOSes DO include a european character set, the BIOS memory card manager DOESN'T seem to translate any title character codes to that character set region.

12.24 Memory Card Images

There are a lot of different ways to get a save from a memory card onto your PC's hard disk, and these ways sometimes involve sticking some additional information into a header at the beginning of the file.

Raw Memory Card Images (without header) (ie. usually 128K in size)

don't stick any header on the data at all, so you can just read it in and treat it like a raw memory card.

All of these headers contain a signature at the top of the file. The three most common formats and their signatures are:

some programs will OMIT any blank or unallocated blocks from the end of the memory card -- if only three save blocks on the card are in use, for example, saving the other twelve is pointless.

Xploder and Action Replay Files (54 byte header)

This format contains only a single file (not a whole memory card). The filename should be the same as used in the Memory Card Directory. The title is more or less don't care; it may be the SHIFT-JIS title from the Title Sector converted to ASCII.

.MCS Files (Single Save Format)

MCS files consist of the 128 byte directory frame for the savefile's first block followed by all of that savefile's blocks in linked list order. When importing this format, the directory frame should be parsed for the save filename and the filesize while other fields should be ignored. The rest of the directory frame fields and any extra directory frames, in the case of multi-block saves, should be reconstructed based on the destination memory card.

.GME Files (usually 20F40h bytes)

InterAct GME format, produced by the DexDrive.

This is a very strange file format, no idea where it comes from. It contains a F40h bytes header (mainly zerofilled), followed by the whole 128K of FLASH memory (mainly zerofilled, too, since it usually contains only a small single executable file).

12.25 Memory Card Notes

Sony PSX Memory Cards

Sony has manufactured only 128KByte memory cards for PSX, no bigger/smaller ones.

Sony PS2 Memory Cards

A special case would be PS2 cards, these are bigger, but PS2 cards won't fit into PSX cards slots (unless when cutting an extra notch in the card edge connector), a PSX game played on a PS2 console could theoretically access PS2 cards (if it supports the different directory structure on that cards).

Third Party Cards with bigger capacity

Some third party cards contain larger memory chips, however, the PSX games/kernel are supporting only regular 128Kbyte cards, so the extra memory can be used only by dividing it into several 128Kbyte memory card images.
Selecting a different memory card image can be done by a switch or button on the card, or via joypad key combinations (joypad/card are sharing the same signals, so the card could watch the traffic on joypad bus, provided that the MIPS CPU is actually reading the joypad).

Third Party Cards with bigger capacity and Data Compression

Some cards are additionally using data compression to increase the card capacity, but that techinque is having rather bad reputation and could result in data loss. For example, if a game has allocated four blocks on the memory card, then it'll expect to be able to overwrite that four blocks at any time (without needing to handle "memory card full" errors), however, if the card is full, and if the newly written data has worse compression ratio, then the card will be unable to store the new game position (and may have already overwritten parts of the old game position). As a workaround, such cards may use a LED to warn users when running low on memory (ideally, there should be always at least 128Kbytes of free memory).

Joytech Smart Card Adaptor

The smart card adaptor plugs into memory card slot, and allows to use special credit card-shaped memory cards. There don't seem to be any special features, ie. the hardware setup does just behave like normal PSX memory cards.

Datel VMEM (virtual memory card storage on expansion port)

The Datel/Interact VMEM exists as standalone VMEM cartridge, and some Datel Cheat Devices do also include the VMEM feature. Either way, the VMEM connects to expansion port, and contain some large FLASH memory, for storing multiple memory cards on it. Unknown, how that memory is accessed (maybe it must be copied to a regular memory card, or maybe they've somehow hooked the Kernel (or even the hardware signals?) so that games could directly access the VMEM?

Passwords (instead of Memory Cards)

Some older games are using passwords instead of memory cards to allow the user to continue at certain game positions. That's nice for people without memory card, but unfortunately many of that games are restricted to it - it'd be more user friendly to support both passwords, and, optionally, memory cards.

Yaroze Access Cards (DTL-H3020)

The Yaroze Access Card connects to memory card slot, the card resembles regular memory cards, but it doesn't contain any storage memory. Instead, it does merely support a very basic Access Card detection command:

Ie. when receiving 21h as first byte, it replies by an ACK, and does then output 0xh as response to the next byte.
Without the Access Card, the Yaroze Bootdisc will refuse to work (the disc contains software for transferring data to/from PC, for developing homebrew games).

Pocketstation (Memory Card with built-in LCD screen and buttons)

Pocketstation

13. Pocketstation

Pocketstation

13.1 Pocketstation Overview

Sony's Pocketstation (SCPH-4000) (1998)

The Pocketstation is a memory card with built-in LCD screen and buttons; aside from using it as memory storage device, it can be also used as miniature handheld console.

The RTC Problem

The main problem of the Pocketstation seems to be that it tends to reset the RTC to 1st January 1999 with time 00:00:00 whenever possible.
The BIOS contains so many RTC-reset functions, RTC-reset buttons, RTC-reset flags, RTC-reset communication commands, RTC-reset parameters, RTC-reset exceptions, RTC-reset sounds, and RTC-reset animations that it seems as if Sony actually WANTED the Time/Date to be destroyed as often as possible.
The only possible reason for doing this is that the clock hardware is so inaccurate that Sony must have decided to "solve" the problem at software engineering side, by erasing the RTC values before the user could even notice time inaccuracies.

CPU Specs

For details on the ARM7TDMI CPUs opcodes and exceptions, check GBATEK at,
http://problemkaputt.de/gbatek.htm (or .txt)
The GBA uses an ARM7TDMI CPU, too.

Thanks to Exophase, Orion, Fezzik, Dr.Hell for Pocketstation info.

13.2 Pocketstation I/O Map

Memory and Memory-Control Registers

Interrupts and Timers

Communication Ports, Audio/Video

BIOS and FLASH can be read only in 16bit and 32bit units (not 8bit).
Upon reset, BIOS ROM is mirrored to address 00000000h (instead of RAM).
For most I/O ports, it is unknown if they are (R), (W), or (R/W)...?
I/O ports are usually accessed at 32bit width, occassionally some ports are (alternately) accessed at 16bit width. A special case are the F_SN registers which seem to be required to be accessed at 16bit (not 32bit).

Memory Access Time

Memory Access Time for Opcode Fetch:

Memory Access Time for Data Read/Write:

For data access, it doesn't matter if the access is 8bit/16bit/32bit (unlike as for opcode fetch, where 16bit/thumb can be faster than 32bit/arm). There seems to be no timing differences for sequential/non-sequential access.
Additional memory waitstates can be added via F_WAIT2 (and F_WAIT1 maybe).

Invalid/Unused Memory Locations

Unsupported 8bit Reads

Unsupported 16bit Reads

garbage_byte (for unsupported 8bit reads)

The "garbage_byte" depends on the LSBs of the read address, prefetched opcodes, and recent data fetches:

For ARM code, the "prefetch" is the 2nd next opcode after the LDRB:

For THUMB code, the "prefetch" is the 2nd next opcode after the LDRB (no matter if that opcode is word-aligned or not), combined with the most recent ARM opcode prefetch (eg. from the BX opcode switched from ARM to THUMB mode; that value may get changed on interrupts):

The "ramdata" is related to most recent RAM read (eg. from POP or LDR opcodes that have read data from RAM; however, writes to RAM, or literal pool reads from FLASH don't affect it):

There might be some more/unknown things that affect the garbage (eg. opcode fetches from RAM instead of FLASH, partial 8bit/16bit data reads from RAM, or reads from I/O areas, current CPU clock speed, or unpredictable things like temperature).
Note: The garbage_byte is "used" by the pocketstation "Rockman" series games.

13.3 Pocketstation Memory Map

Overall Memory Map

00000000h..000001FFh - Kernel RAM

The first 200h bytes of RAM are reserved for the kernel.

Although one can modify that memory, one usually shouldn't do that, or at least one must backup and restore the old values before returning control to the GUI or to other executables. Otherwise, the only way to restore the original values would be to press the Reset button (which would erase the RTC time/date).

00000200h..000007FFh - User RAM and User stack (stacktop at 800h)

This region can be freely used by the game. The memory is zerofilled when the game starts.

02000000h - FLASH1 - Flash ROM (virtual file-mapped addresses in this region)

This region usually contains the currently selected file (including its title and icon sectors), used to execute the file in this region, mapped to continous addresses at 2000000h and up.

08000000h - FLASH2 - Flash ROM (128K) (physical addresses in this region)

This region is used by the BIOS when reading the memory card directory (and when writing data to the FLASH memory). The banking granularity is 2000h bytes (one memory card block), that means that the hardware cannot map Replacement Sectors which may be specified in the for Broken Sector List.

04000000h - BIOS ROM (16K) - Kernel and GUI

The "110" version does contain some patches, but does preserve same function addresses as the "061" version, still it'd be no good to expect the BIOS to contain any code/data at fixed locations (except maybe the GUI version string). Kernel functions can be accessed via SWI Opcodes, and, from the PSX-side, via BU Commands.

Bus-Width Restrictions

FLASH and BIOS ROM seem to be allowed to be read only in 16bit and 32bit units, not in 8bit units? Similar restrictions might apply for some I/O ports...? RAM can be freely read/written in 8bit, 16bit, and 32bit units.

Waitstates

Unknown if and how many waitstates are applied to the different memory regions. The F_WAIT1 and F_WAIT2 registers seem to be somehow waitstate related. FLASH memory does probably have a 16bit bus, so 32bit data/opcode fetches might be slower then 16bit reads...? Similar delays might happen for other memory and I/O regions...?

13.4 Pocketstation IO Video and Audio

0D000000h - LCD_MODE - LCD control word (R/W)

Software should usually set LCD_MODE.7 equal to INT_INPUT.Bit11 (docking flag). In handheld mode, the button-side is facing towards the player, whilst in Docked mode (when the Pocketstation is inserted into the PSX controller port), the button-side is facing towards the PSX, so the screen coordinates become vice-versa, which can be "undone" by the Rotation flag.

0D000004h - LCD_CAL - LCD Calibration (maybe contrast or so?)

Upon the reset, the kernel sets LCD_CAL = F_CAL AND 0000003Fh. Aside from that, it doesn't use LCD_CAL.

0D000100h..D00017Fh - LCD_VRAM - 32x32 pixels, 1bit color depth (R/W)

This region consists of 32 words (32bit values), The separate scanlines consist of 32bit each,

That [D000100h].Bit0=Upper-left arrangement applies if the Rotate bit in LCD_MODE.7 is set up in the conventional way, if it is set the opposite way, then it becomes [D00017Ch].Bit31=Upper-left.
The LCD_VRAM area is reportedly mirrored to whatever locations?

0D800010h - DAC_CTRL - Audio Control (R/W)

Note: Aside from the bit in DAC_CTRL, audio must be also enabled/disabled via IOP_STOP/IOP_START bit5. Unknown if/which different purposes that bits have.

0D800014h - DAC_DATA - Audio D/A Converter

Unknown how many bits are passed to the D/A converter, probably bit8-15, ie. 8 bits...?

The Pocketstation doesn't have any square wave or noise generator (nor a sound DMA channel). So the output levels must be written to DAC_DATA by software, this is usually done via Timer1/IRQ-8 (to reduce CPU load caused by high audio frequencies, it may be much more recommended to use Timer2/FIQ-13, because the FIQ handler doesn't need to push r8-r12).
For example, to produce a 1kHz square wave, the register must be toggled high/low at 2kHz rate. If desired, multiple channels can be mixed by software. High frequencies and multiple voices may require high CPU speed settings, and thus increase battery consumption (aside from that, battery consumption is probably increased anyways when the speaker is enabled).

13.5 Pocketstation IO Interrupts and Buttons

0A000004h - INT_INPUT - Raw Interrupt Signal Levels (R)

The buttons are usually read directly from this register (rather than being configured to trigger IRQs) (except in Sleep mode, where the Fire Button IRQ is usually used to wakeup). Also, bit9-11 are often read from this register.
The direction keys seem to be separate buttons, ie. unlike as on a joystick or DPAD, Left/Right (and Up/Down) can be simultaneously pressed...?

0A000008h - INT_MASK_SET - Set Interrupt Mask (W)

0A00000Ch - INT_MASK_CLR - Clear Interrupt Mask (W)

0A000008h - INT_MASK_READ - Read Interrupt Mask (R)

The locations of the separate bits are same as in INT_INPUT (see there).

0A000000h - INT_LATCH - Interrupt Request Flags (R)

0A000010h - INT_ACK - Acknowledge Interrupts (W)

The locations of the separate bits are same as in INT_INPUT (see there).
The interrupts seem to be edge-triggered (?), ie. when the corresponding bits in INT_INPUT change from 0-to-1. Unknown if the request bits get set when the corresponding interrupt is disabled in INT_MASK...?

ATTENTION: The GUI doesn't acknowledge Fire Button interrupts on wakeup... so, it seems as if button interrupts are NOT latched... ie. the button "INT_LATCH" bits seem to be just an unlatched mirror of the "INT_INPUT" bits... that might also apply for some other interrupt...?
However, after wakeup, the gui does DISABLE the Fire Button interrupt, MAYBE that does automatically acknowledge it... in that case it might be latched...?

Reading outside the readable region (that is where exactly?) seems to mirror to 0A000008h. Enabling IRQs for the buttons seems to make it impossible to poll them... is that really true?

13.6 Pocketstation IO Timers and Real-Time Clock

Timer and RTC interrupts

0A800000h - T0_RELOAD - Timer 0 Reload Value

0A800010h - T1_RELOAD - Timer 1 Reload Value

0A800020h - T2_RELOAD - Timer 2 Reload Value

Writes to this register are ignored if the timer isn't stopped?

0A800004h - T0_COUNT - Timer 0 Current value

0A800014h - T1_COUNT - Timer 1 Current value

0A800024h - T2_COUNT - Timer 2 Current value

Timer interrupts: The timers will automatically raise interrupts if they're enabled, there's no need to set a bit anywhere for IRQs (but you need to enable the respect interrupts in INT_MASK).

0A800008h - T0_MODE - Timer 0 Control

0A800018h - T1_MODE - Timer 1 Control

0A800028h - T2_MODE - Timer 2 Control

Timers are clocked by the System Clock (usually 4MHz, when CLK_MODE=7), divided by the above divider setting. Note that the System Clock changes when changing the CPU speed via CLK_MODE, so Timer Divider and/or Timer Reload must be adjusted accordingly.

0B800000h - RTC_MODE - RTC control word

The selection bits can be:

When paused, the RTC IRQ bit in INT_INPUT.9 runs at 4096Hz (instead 1Hz).

0B800004h - RTC_ADJUST - Modify value (write only)

Writing a value here seems to increment the current selected parameter (by the RTC control). What is perhaps (?) clear is that you have to wait for the RTC interrupt signal to go low before writing to this.

0B800008h - RTC_TIME - Real-Time Clock Time (read only) (R)

Reading RTC_TIME seems to be somewhat unstable: the BIOS uses a read/retry loop, until it has read twice the same value (although it does read the whole 32bit at once by a LDR opcode, the data is maybe passed through a 8bit or 16bit bus; so the LSBs might be a few clock cycles older than the MSBs...?).

0B80000Ch - RTC_DATE - Real-Time Clock Date (read only) (R)

Reading RTC_DATE seems to require the same read/retry method as RTC_TIME (see there). Note: The century is stored in battery-backed RAM (in the reserved kernel RAM region) rather than in the RTC_DATE register. The whole date, including century, can be read via SWI 0Dh, GetBcdDate().

13.7 Pocketstation IO Infrared

The BIOS doesn't contain any IR functions (aside from doing some basic initialization and power-down stuff).
IR is used in Final Fantasy 8's Chocobo World (press Left/Right in the Map screen to go

to the IR menu), and in Metal Gear Solid Integral (Press Up in the main screen), and in PDA Remote 1 & 2 (one-directional TV remote control).

0C800000h - IRDA_MODE - Controlling the protocol - send/recv, etc. (R/W)

0C800004h - IRDA_DATA - Infrared TX Data

Bits are usually encoded as long or short ON pulses, separated by short OFF pulses. Where long is usually twice as long as short.

0C80000Ch - IRDA_MISC

Unknown? Reportedly reserved.

INT_INPUT.12 - IRQ - Infrared RX Interrupt

Seems to get triggered on raising or falling (?) edges of incoming data. The interrupt handler seems to read the current counter value from one of the timers (usually Timer 2, with reload=FFFFh) to determine the length of the incoming IR pulse.

IR Notes

Mind that IR hardware usually adopts itself to the normal light conditions, so if it receives an IR signal for a longer period, then it may treat that as the normal light conditions (ie. as "OFF" state). To avoid that, one would usually send a group of ON-OFF-ON-OFF pulses, instead of sending a single long ON pulse:

that might be maybe done automatically by the hardware...?

Reportedly, Bit4 of Port 0D80000Ch (IOP_DATA) is also somewhat IR related...?

13.8 Pocketstation IO Memory-Control

06000000h - F_CTRL

Written values are:

The GUI does additionally read from this register (and gets itself trapped in a bizarre endless loop if bit0 was zero). Unknown if it's possible to re-enable ROM at location 00000000h by writing any other values to this register?

06000004h F_STAT

The kernel issues a dummy read from this address (before setting F_CTRL to 00000001h).

06000008h F_BANK_FLG ;FLASH virtual bank mapping enable flags (16 bits)(R/W)

06000100h F_BANK_VAL ;FLASH virtual bank mapping addresses (16 words)(R/W)

This region contains 16 words, the first word at 06000100h for physical bank 0, the last word at 0600013Ch for physical bank 15. Each word is:

Unused physical banks are usually mapped to 0Fh (and are additionally disabled in the F_BANK_FLG register).

0600000Ch F_WAIT1 ;waitstates...?

Unknown, seems to control some kind of memory waitstates for FLASH (or maybe RAM or BIOS ROM). Normally it is set to the following values:

Note: The kernels Docking/Undocking IRQ-11 handler does additionally do this: "F_WAIT1=max(08h,(CLK_MODE AND 0Fh))" (that is a bug, what it actually wants to do is to READ the current F_WAIT.Bit4 setting).

06000010h F_WAIT2 ;waitstates, and FLASH-Write-Control-and-Status...?

Unknown, seems to control some kind of memory waitstates, maybe for another memory region than F_WAIT1, or maybe F_WAIT2 is for writing, and F_WAIT1 for reading or so. Normally it is set to the following values:

In SWI 0Fh and SWI 10h it is also set to:

Before completion, those SWIs do additionally,

08002A54h - F_KEY1 - Flash Unlock Address 1 (W)

080055AAh - F_KEY2 - Flash Unlock Address 2 (W)

Unlocks FLASH memory for writing. The complete flowchart for writing sector data (or header values) is:

During the write operation one can (probably?) not read data (nor opcodes) from FLASH memory, so the above code must be executed either in RAM, or in BIOS ROM (see SWI 03h, SWI 0Fh, SWI 10h).

06000300h - F_SN_LO - Serial Number LSBs

06000302h - F_SN_HI - Serial Number MSBs

06000308h - F_CAL - Calibration value for LCD

This seems to be an additional "header" region of the FLASH memory (additionally to the 128K of data). The F_SN registers contain a serial number or so (purpose unknown,

maybe intended as some kind of an "IP" address for more complex infrared network applications), the two LO/HI registers must be read by separate 16bit LDRH opcodes (not by a single 32bit LDR opcode). The F_CAL register contains a 6bit calibration value for LCD_CAL (contrast or so?).
Although only the above 3 halfwords are used by the BIOS, the "header" is unlike to be 6 bytes in size, probably there are whatever number of additional "header" locations at 06000300h and up...?
Note: Metal Gear Solid Integral uses F_SN as some kind of copy protection (the game refuses to run and displays "No copy" if F_SN is different as when the pocketstation file was initially created).

F_BANK_VAL and F_BANK_FLG Notes

Observe that the physical_bank number (p) is used as array index, and that the virtual bank number (v) is stored in that location, ie. table[p]=v, which is unlike as one may have expected it (eg. on a 80386 CPU it'd be vice-versa: table[v]=p).
Due to the table[p]=v assignment, a physical block cannot be mirrored to multiple virtual blocks, instead, multiple physical blocks can be mapped to the same virtual block (unknown what happens in that case, maybe the data becomes ANDed together).

13.9 Pocketstation IO Communication Ports

0C000000h - COM_MODE - Com Mode

0C000008h - COM_DATA - Com RX/TX Data

0C000004h - COM_STAT1 - Com Status Register 1 (Bit1=Error)

Seems to indicate whatever error (maybe /SEL disabled during transfer, or timeout, or parity error or something else?) in bit1. Meaning of the other bits is unknown. Aside from checking the error flag, the kernel does issue a dummy read at the end of each transfer, maybe to acknowledge something, maybe the hardware simply resets the error bit after reading (although the kernel doesn't handle the bit like so when receiving the 1st command byte).
Aside from the above error flag, one should check if INT_INPUT.11 becomes zero during transfer (which indicates undocking).

0C000014h - COM_STAT2 - Com Status Register 2 (Bit0=Ready)

0C000010h - COM_CTRL1 - Com Control Register 1

Used values are:

When doing the enable thing, Bit1 should be set to 0-then-1...? Bit0 might enable the data shift register... and bit1 might be a master enable and master acknowledge for the COM interrupt... or something else?

0C000018h - COM_CTRL2 - Com Control Register 2

Used values are:

Maybe that two bits acknowledge the ready/error bits?

INT_INPUT.6 FIQ (!) COM for the COM_registers? (via /SEL Pin?)

INT_INPUT.11 IRQ Docked ("IOP") (0=Undocked, 1=Docked to PSX)

Probably senses the voltage on the cartridge slots VCC Pin. Becomes zero when Undocked (and probably also when the PSX is switched off).
The Kernel uses IRQ-11 for BOTH sensing docking and undocking, ie. as if the IRQ would be triggered on both 0-to-1 and 1-to-0 transistions... though maybe that feature just relies on switch-bounce. For the same reason (switch bounce), the IRQ-11 handler performs a delay before it checks the new INT_INPUT.11 setting (ie. the delay skips the unstable switch bound period, and allows the signal to stabilize).

IOP_START/IOP_STOP.Bit1

The BIOS adjusts this bit somehow in relation to communication. Unknown when/why/how it must be used. For details on IOP_START/IOP_STOP see Power Control chapter.

Opcode E6000010h (The Undefined Instruction) - Write chr(r0) to TTY

This opcode is used by the SN Systems emulator to write chr(r0) to a TTY style text window. r0 can be ASCII characters 20h and up, or 0Ah for CRLF. Using that opcode is a not too good idea because the default BIOS undef instruction handler simply runs into an endless loop, so games that are using it (eg. Break-Thru by Jason) won't work on real hardware. That, unless the game would change the undef instruction vector at [04h] in Kernel RAM, either replacing it by a MOVS R15,R14 opcode (ignore exception and return to next opcode), or by adding exception handling that outputs the character via IR or via whatever cable connection. Observe that an uninitialized FUNC3 accidently destroys [04h], so first init FUNC3 handler via SWI 17h, before trying to change [04h], moreover, mind that SWI 05h may reset FUNC3, causing the problem to reappear.
Altogether, it'd be MUCH more stable to write TTY characters to an unused I/O port... only problem is that it's still unknown which I/O ports are unused... ie. which do neither trap data aborts, nor do mirror to existing ports...?

13.10 Pocketstation IO Power Control

0B000000h - CLK_MODE - Clock control (CPU and Timer Speed) (R/W)

Allows to change the CPU clock (and Timer clock, which is usually one half of the CPU clock, or less, depending on the Timer Divider). Possible values are:

Before changing CLK_MODE, F_WAIT1 and F_WAIT2 should be adjusted accordingly (see there for details). Note that many memory regions have waitstates, the full CPU speed can be reached mainly with code/data in WRAM.

0B000004h - CLK_STOP - Clock stop (Sleep Mode)

Stops the CPU until an interrupt occurs. The pocketstation doesn't have a power-switch nor standby button, the closest thing to switch "power off" is to enter sleep mode. Software should do that when the user hasn't pressed buttons for 1-2 seconds (that, only in handheld mode, not when docked to the PSX; where it's using the PSX power supply instead of the battery).

Wakeup is usually done by IRQ-0 (Fire Button) and IRQ-11 (Docking). If alarm is enabled, then the GUI also enables IRQ-9 (RTC), and compares RTC_TIME against the

alarm setting each time when it wakes up.
Before writing to CLK_STOP, one should do:

The GUI uses CLK_STOP only for Standby purposes (not for waiting for its 30Hz "frame rate" timer 0 interrupt; maybe that isn't possible, ie. probably CLK_STOP does completely disable the system clock, and thus does stop Timer0-2...?)

0D800000h - IOP_CTRL - Configures whatever...? (R/W)

Unknown. Set to 0000000Fh by BIOS upon reset. Aside from that, the BIOS does never use that register.

0D800004h - IOP_STAT (R) - Read Current bits? -- No, seems to be always 0

0D800004h - IOP_STOP (W) - Set IOP_DATA Bits

0D800008h - IOP_START (W) - Clear IOP_DATA Bits

These two ports are probably accessing a single register, writing "1" bits to IOP_STOP sets bits in that register, and writing "1" bits to IOP_START clears bits... or vice-versa...? Writing "0" bits to either port seems to leave that bits unchanged. The meaning of most bits is still unknown:

Aside from Bit1, it's probably not neccessary to change the unknown bits...?
Sound is usually disabled by setting IOP_STOP=00000020h. IOP_STAT is rarely used. Although, one piece of code in the BIOS disables sound by setting IOP_STOP=IOP_STAT OR 00000020h, that is probably nonsense, probably intended to keep bits stopped if they are already stopped (which would happen anyways), however, the strange code implies that reading from 0D800004h returns the current status of the register, and that the bits in that register seem to be 0=Started, and 1=Stopped...?

0D80000Ch - IOP_DATA (R)

Unknown. Not used by the BIOS. Reportedly this register is 0010h if IR Connection...?
This register is read by Rewrite ID, and by Harvest Moon. Maybe bit4 doesn't mean \<if> IR connection exist, but rather \<contains> the received IR data level...?

0D800020h - BATT_CTRL - Battery Monitor Control?

Unknown. Somehow battery saving related. Upon reset, and upon leaving sleep mode, the BIOS does set BATT_CTRL=00000000h. Before entering sleep mode, it does set BATT_CTRL=BATT_CTRL AND FFFFFFFCh, whereas, assuming that BATT_CTRL was 00000000h, ANDing it with FFFFFFFCh would simply leave it unchanged... unless the hardware (or maybe a game) sets some bits in BATT_CTRL to nonzero values...?

Battery Low Interrupt

Can be used to sense if the battery is low, if so, one may disable sound output and/or reduce the CPU speed to increase the remaining battery lifetime. Unknown how long the battery lasts, and how much the lifetime is affected by audio, video, infrared, cpu speed, and sleep mode...?
The pocketstation can be also powered through the VCC pin (ie. when docked to the PSX, then it's working even if the battery is empty; or even without battery).

13.11 Pocketstation SWI Function Summary

SWI Function Summary

BIOS functions can be called via SWI opcodes (from both ARM and THUMB mode) (in ARM mode, the SWI function number is in the lower 8bit of the 24bit field; unlike as for example on the GBA, where it'd be in the upper 8bit). Parameters (if any) are passed in r0,r1,r2. Return value is stored in r0 (all other registers are left unchanged).

The BIOS uses the same memory region for SWI and IRQ stacks, so both may not occur simultaneously, otherwise one stack would be destroyed by the other (normally that is no problem; IRQs are automatically disabled by the CPU during SWI execution, SWIs aren't used from inside of default IRQ handlers, and SWIs shouldn't be used from inside of hooked IRQ handlers).

13.12 Pocketstation SWI Misc Functions

SWI 01h - SetCallbacks(index,proc)

All callbacks are called via BX opcodes (ie. proc.bit0 can be set for THUMB code). SetCallbacks returns the old proc value (usually zero). The callbacks are automatically reset to zero when (re-)starting an executable, or when returning control to the GUI, so there's no need to restore the values by software.

IRQ and FIQ Callbacks

Registers r0,r1,r12 are pushed by the kernels FIQ/IRQ handlers (so the callbacks can use that registers without needing to push them). The FIQ handler can additionally use r8..r11 without pushing them (the CPU uses a separate set of r8..r12 registers in FIQ mode, nethertheless, the kernel DOES push r12 in FIQ mode, without reason). Available stack is 70h bytes for the FIQ callback, and 64h bytes for the IRQ callback.
The callbacks don't receive any incoming parameters, and don't need to respond with a return value. The callback should return to the FIQ/IRQ handler (via normal BX r14) (ie. it should not try to return to User mode).
The kernel IRQ handler does (after the IRQ callback) process IRQ-11 (IOP) (which does mainly handle docking/undocking), and IRQ-9 (RTC) (which increments the century if the year wrapped from 99h to 00h).
And the kernel FIQ handler does (before the FIQ callback) process IRQ-6 (COM) (which does, if ComFlags.Bit9 is set, handle bu_cmd's) (both IRQs and FIQs are disabled, and the main program is stopped until the bu_cmd finishes, or until a joypad command is identified irrelevant, among others that means that sound/timer IRQs aren't processed during that time, so audio output may become distorted when docked).
When docked, the FIQ callback should consist of only a handful of opcodes, eg. it may contain a simple noise, square wave generator, or software based sound "DMA" function, but it should not contain more time-consuming code like sound envelope processing; otherwise IRQ-6 (COM) cannot be executed fast enough to handle incoming commands.

SWI 02h - CustomSwi2(r0..r6,r8..r10) out: r0

Calls the SWI2 callback function (which can be set via SWI 01h). The default callback address is 00000000h (so, by default, it behaves identically as SWI 00h). Any parameters can be passed in r0..r6 and r8..r10 (the other registers aren't passed to the callback function). Return value can be stored in r0 (all other registers are pushed/popped by the swi handler, as usually). Available space on the swi stack is 38h bytes.
SWI2 can be useful to execute code in privileged mode (eg. to initialize FIQ registers r8..r12 for a FIQ based sound engine) (which usually isn't possible because the main program runs in non-privileged user mode).

SWI 04h - SetCpuSpeed(speed) out: old_speed

Changes the CPU speed. The BIOS uses it with values in range 01h..07h. Unknown if value 00h can be also used? The function also handles values bigger than 07h, of which, some pieces of BIOS code look as if 08h would be the maximum value...?
Before setting the new speed, the function sets F_WAIT1 and F_WAIT2 to 00000000h (or to 00000010h if speed.bit3=1). After changing the speed (by writing the parameter to CLK_MODE) it does wait until the new speed is applied (by waiting for CLK_MODE.bit4 to become zero). The function returns the old value of CLK_MODE, anded with 0Fh.

13.13 Pocketstation SWI Communication Functions

Communication (aka BU Commands, received from the PSX via the memory card slot) can be handled by the pocketstations kernel even while a game is running. However, communications are initially disabled when starting a game, so the game should enable them via SWI 11h, and/or via calling SWI 05h once per frame.

SWI 11h - SetComOnOff(flag)

Can be used to enable/disable communication. When starting an executable, communication is initially disabled, so it'd be a good idea to enable them (otherwise the PSX cannot communicate with the Pocketstation while the game is running).
When flag=0, disables communication: Intializes the COM_registers, disables IRQ-6 (COM), and clears ComFlags.9. When flag=1, enables communication: Intializes the COM_registers, enables IRQ-6 (COM), sets ComFlags.9 (when docked), or clears Sys.Flags.9 (when undocked), and sets FAST cpu_speed=7 (only when docked). The function returns garbage (r0=retadr to swi_handler).

SWI 06h - GetPtrToComFlags()

Returns a pointer to the ComFlags word in RAM, which contains several communication related flags (which are either modified upon docking/undocking, or upon receiving certain bu_cmd's). The ComFlags word consists of the following bits:

Bit16-18 can be changed via SWI 07h, ChangeAutoDocking(flags). Bit10 can be cleared by SWI 0Bh, ClearComFlagsBit10().

SWI 07h - ChangeAutoDocking(flags.16-18)

Copies bit16-18 of the incoming parameter to ComFlags.16-18, specifying how the kernel IRQ-11 (IOP) handler shall process docking/undocking from the PSX cartridge slot. The function returns the incoming flags value ANDed with 70000h.

SWI 0Bh - ClearComFlagsBit10()

Resets ComFlags.Bit10, ie. enables bu_cmd_57h (write_sector) to write to the Broken Sector region in FLASH memory (sector 16..55). SWI 0Bh returns the current ComFlags value (the new value, with bit10=0).
Aside from calling SWI 0Bh, ComFlags.10 is also automatically cleared upon IRQ-10 (IOP) (docking/undocking). ComFlags.10 can get set/cleared by the Download Notification callback.

SWI 05h - SenseAutoCom()

Checks if docking/undocking has occurred (by examining ComFlags.8, which gets set by the kernel IRQ-11 (IOP) handler). If that flag was set, then the function does reset it, and does then reset FUNC3=0000h and [0CAh]=00h (both only if docked, not when undocked), and, no matter if docked or undocked, it enables communication; equivalent to SetComOnOff(1); which sets/clears ComFlags.9. The function returns garbage (r0=whatever).
The GUI is calling SWI 05h once per frame. The overall purpose is unknown. It's a good idea to reset FUNC3 and to Enable Communication (although that'd be required only when docked, not when undocked), but SWI 05h is doing that only on (un-)docking transitions (not when it was already docked). In general, it'd make more sense to do proper initializations via SWI 11h and SWI 17h as than trusting SWI 05h to do the job. The only possibly useful effect is that SWI 05h does set/clear ComFlags.9 when docked/ undocked.

SWI 17h - GetPtrToFunc3addr()

Returns a pointer to a halfword in RAM which contains the FUNC3 address (for bu_cmd_5bh and bu_cmd_5ch). The address is only 16bit, originated at 02000000h in FLASH (ie. it can be only in the first 64K of the file), bit0 can be set for THUMB code.
The default address is zero, which behaves bugged: It accidently sets [00000004h]=00000000h, ie. replaces the Undefined Instruction exception vector by a "andeq r0,r0,r0" opcode, due to that NOP-like opcode, any Undefined Instruction exceptions will run into the SWI vector at [00000008h], and randomly execute an SWI function; with some bad luck that may execute one of the FlashWrite functions and destroy the saved files.
Although setting 0000h acts bugged, one should restore that setting before returning control to GUI or other executables; otherwise the address would still point to the FUNC3 address of the old unloaded executable, which is worse than the bugged effect.
The FUNC3 address is automatically reset to 0000h when (if) SWI 05h (SenseAutoCom) senses new docking.

Download Notification callback

Can be used to mute sound during communication, see SWI 01h, SetCallbacks(index,proc), and BU Command 5Dh for details.

13.14 Pocketstation SWI Execute Functions

SWI 08h - PrepareExecute(flag,dir_index,param)

dir_index should be 0=GUI, or 1..15=First block of game. When calling DoExecute, param is passed to the entrypoint of the game or GUI in r0 register (see notes on GUI \<param> values belows). For games, param may be interpreted in whatever way. When flag=0, the function simply returns the old dir_index value. When flag=1, the new dir_index and param values are stored in Kernel RAM (for being used by DoExecute); the values are stored only if dir_index=0 (GUI), or if dir_index belongs to a file with "SC" and "MCX0" or "MCX1" IDs in it's title sector. If dir_index was accepted, then the new dir_index value is returned, otherwise the old dir_index is returned.

GUI \<param> values - for PrepareExecute(1,0,param)

PrepareExecute(1,0,param) prepares to execute the GUI (rather than a file). When executing the GUI, \<param> consists of the following destructive bits:

The command numbers can be:

For Command 2xh and 3xh, the lower 4bit of the command (x) must be a valid dir_index of the 1st block of a pocketstation executable, otherwise the BIOS erases the RTC time/date. Bit8 is just a "funny" nag feature, allowing the user to change the alarm setting, but with the changes being ignored (bit8 can be actually useful in BU Command 59h, after FUNC2 was used for changing alarm).

SWI 09h - DoExecute(), or DoExecute(snapshot_saving_flag) for MCX1

Allows to return control to the GUI (when dir_index=0), or to start an executable (when dir_index=1..15). Prior to calling DoExecute, parameters should be set via

PrepareExecute(1,dir_index,param), when not doing that, DoExecute would simply restart the current executable (which may be a desired effect in some cases).
The "snapshot_saving_flag" can be ommited for normal (MCX0) files, that parameter is used only for special (MCX1) files (see Snapshot Notes for details).
Caution: DoExecute fails (and returns r0=unchanged) when ComFlags.9=1 (which indicates that communications are enabled, and that the Pocketstation is believed to be docked to the PSX). ComFlags.9 can be forcefully cleared by calling SetComOnOff(0), or it can be updated according to the current docking-state by calling SetComOnOff(1) or SenseAutoCom().

SWI 16h - GetDirIndex()

Returns the dir_index for the currently executed file. If that value is zero, ie. if there is no file executed, ie. if the function is called by the GUI, then it does instead return the "alternate" dir_index (as set via SWI 15h).

SWI 15h - MakeAlternateDirIndex(flag,dir_index) out: alt_dir_index (new/old)

Applies the specified dir_index as "alternate" dir_index (for being retrieved via SWI 16h for whatever purpose). The dir_index is applied only when flag=1, and only if dir_index is 0=none, or if it is equal to the dir_index of the currently executed file (ie. attempts to make other files being the "alternate" one are rejected). If successful, the new dir_index is returned, otherwise the old dir_index is returned (eg. if flag=0, or if the index was rejected).

SWI 12h - TestSnapshot(dir_index)

Tests if the specified file contains a load-able snapshot, ie. if it does have the "SC" and "MCX1" IDs in the title sector, and the 01h,00h,"SE" ID in the snapshot header. If so, it returns r0=1, and otherwise returns r0=0.

Snapshot Notes (MCX1 Files)

Snapshots are somewhat automatically loaded/saved when calling DoExecute:
If the old file (the currently executed file) contains "SC" AND "MCX1" IDs in the title sector, then the User Mode CPU registers and User RAM at 200h..7FFh are automatically saved in the files snapshot region in FLASH memory, with the snapshot_saving_flag being applied as bit0 of the 0xh,00h,"SE" ID of the snapshot header).
If the new file (specified in dir_index) contains load-able snapshot data (ie. if it has "SC" and "MCX1" IDs in title sector, and 01h,00h,"SE" ID in the snapshot region), then the

BIOS starts the saved snapshot data (instead of restarting the executable at its entrypoint). Not too sure if that feature is really working... the snapshot loader seems to load User RAM from the wrong sectors... and it seems to jump directly to User Mode return address... without removing registers that are still stored on SWI stack... causing the SWI stack to underflow after loading one or two snapshots...?

13.15 Pocketstation SWI Date/Time/Alarm Functions

SWI 0Ch - SetBcdDateTime(date,time)

Sets the time and date, the parameters are having the same format as SWI 0Dh and SWI 0Eh return values (see there). The SWI 0Ch return value contains only garbage (r0=RTC_DATE/10000h).

SWI 0Dh - GetBcdDate()

Returns the current date, the lower 24bit are read from RTC_DATE, the century in upper 8bit is read from Kernel RAM.

SWI 0Eh - GetBcdTime()

Returns the current time and day of week, read from RTC_TIME.

SWI 13h - GetPtrToAlarmSetting()

Returns a pointer to a 64bit value in Kernel RAM, the upper word (Bit32-63) isn't actually used by the BIOS, except that, the bu_cmd FUNC3 does transfer the whole 64bits. The meaning of the separate bits is:

The RTC hardware doesn't have a hardware-based alarm feature, instead, the alarm values must be compared with the current time by software. Alarm is handled only by the GUI portion of the BIOS. The Kernel doesn't do any alarm handling, so alarm won't occur while a game is executed (unless the game contains code that handles alarm). Games are usually using only the lower 16bit of the charset address, ORed with 04000000h (although the full 32bit is stored in RAM).

13.16 Pocketstation SWI Flash Functions

SWI 10h - FlashWritePhysical(sector,src)

Writes 80h-bytes at src to the physical sector number (0..3FFh, originated at 08000000h), and does then compare the written data with the source data. Returns 0=okay, or 1=failed.

SWI 03h - FlashWriteVirtual(sector,src)

The sector number (0..3FFh) is a virtual sector number (originated at 02000000h), the function uses the F_BANK_VAL settings to translate it to a physical sector number, and does then write the 80h-bytes at src to that location (via the FlashWritePhysical function). Returns 0=okay, or 1=failed (if the write failed, or if the sector number exceeded the filesize aka the virtually mapped memory region).

SWI 0Ah - FlashReadSerial()

Returns the 32bit value from the two 16bit F_SN registers (see F_SN for details).

SWI 0Fh - FlashWriteSerial(serial_number) ;old BIOS only!

Changes the 32bit F_SN value in the "header" region of the FLASH memory. The function also rewrites the F_CAL value (but it simply rewrites the old value, so it's left unchanged). The function isn't used by the BIOS, no idea if it is used by any games. No return value (always returns r0=0).
This function is supported by the old "061" version BIOS only (the function is padded with jump opcodes which hang the CPU in endless loops on newer "110" version).

SWI 18h - FlashReadWhateverByte(sector)

Returns [8000000h+sector*80h+7Eh] AND 00FFh. Purpose is totally unknown... the actual FLASH memory doesn't contain any relevant information at that locations (eg. the in the directory sectors, that byte is unused, usually zero)... and, reading some kind of status or manufacturer information would first require to command the hardware to output that info...?

13.17 Pocketstation SWI Useless Functions

SWI 00h - Reset() ;don't use, destroys RTC settings

Reboots the pocketstation, similar as when pressing the Reset button. Don't use! The BIOS bootcode does (without any good reason) reset the RTC registers and alarm/century settings in RAM to Time 00:00:00, Date 01 Jan 1999, and Alarm 00:00 disabled, so, after reset, the user would need to re-enter these values.
Aside from the annoying destroyed RTC settings, the function is rather unstable: it does jump to address 00000000h in RAM, which should usually redirect to 04000000h in ROM, however, most pocketstation games are programmed in C language, where "pointer" is usually pronounced "pointer?" without much understanding of whether/why/how to initialize that "strange things", so there's a good probability that one of the recently executed games has accidently destroyed the reset vector at [00000000h] in battery-backed RAM.

SWI 14h - GetPtrToPtrToSwiTable()

Returns a pointer to a word in RAM, which contains another pointer which usually points to SWI table in ROM. Changing that word could be (not very) useful for setting up a custom SWI table in FLASH or in RAM. When doing that, one must restore the original setting before returning control to the GUI or to another executable (the setting isn't automatically restored).

13.18 Pocketstation BU Command Summary

The Pocketstation supports the standard Memory Card commands (Read Sector, Write Sector, Get Info), plus a couple of special commands.

BU Command Summary

Commands 5Bh and 5Ch can use the following functions:

13.19 Pocketstation BU Standard Memory Card Commands

For general info on the three standard memory card commands (52h, 53h, 57h), and for info on the FLAG response value, see:
Memory Card Read/Write Commands

BU Command 52h (Read Sector)

Works much as on normal memory cards, except that, on the Pocketstation, the Read Sector command return 00h as dummy values; instead of the "(pre)" dummies that occur on normal memory cards.
The Read Sector command does reproduce the strange delay (that occurs between 5Ch and 5Dh bytes), similar as on normal original Sony memory cards, maybe original cards did (maybe) actually DO something during that delay period, the pocketstation BIOS simply blows up time in a wait loop (maybe for compatibility with original cards).

BU Command 53h (Get ID)

The Get ID command (53h) returns exactly the same values as normal original Sony memory cards.

BU Command 57h (Write Sector)

The Write Sector command has two new error codes (additonally to the normal 47h="G"=Good, 4Eh="N"=BadChecksum, FFh=BadSector responses). The new error codes are (see below for details):

And, like Read Sector, it returns 00h instead of "(pre)" as dummy values.

Write Error Code FDh (Directory Entries of currently executed file)

The FDh error code is intended to prevent the PSX bootmenu (or other PSX games) to delete the currently executed file (which would crash the pocketstation - once when the deleted region gets overwritten by a new file), because the PSX bootmenu and normal PSX games do not recognize the new FDh error code the pocketstation does additionally set FLAG.3 (new card), which should be understood by all PSX programs.
The FDh error code occurs only on directory sectors of the file (not on its data blocks).

However, other PSX games should never modify files that belong to other games (so there should be no compatibility problem with other PSX programs that aren't aware of the file being containing currently executed code).
However, the game that has created the executable pocketstation file must be aware of that situation. If the file is broken into a Pocketstation Executable region and a PSX Gameposition region, then it may modify the Gameposition stuff even while the Executable is running. If the PSX want to overwrite the executable then it must first ensure that it isn't executed (eg. by retrieving the dir_index of the currently executed file via BU Command 5Ah, and comparing it against the first block number in the files FCB at the PSX side; for file handle "fd", the first block is found at "[104h]+fd*2Ch+24h" in PSX memory).

Write Error Code FEh (write-protected Broken Sector region, sector 16..55)

The write-protection is enabled by ComFlags.bit10 (which can be set/cleared via BU Command 5Dh). That bit should be set before writing Pocketstation excecutables (the Virtual Memory banking granularity is 2000h bytes, which allows to map whole blocks only, but cannot map single sectors, which would be required for files with broken sector replacements).
Unlike Error FDh, this error code doesn't set FLAG.3 for notifying normal PSX programs about the error (which is no problem since normally Error FEh should never occur since ComFlags.10 is usually zero). For more info on ComFlags.10, see SWI 0Bh aka ClearComFlagsBit10(), and BU Command 5Dh.

13.20 Pocketstation BU Basic Pocketstation Commands

BU Command 50h (Change a FUNC 03h related value or so)

Might be somehow related to FUNC 03h...?

BU Command 58h (Get an ID or Version value or so)

BU Command 59h (Prepare File Execution with Dir_index, and Parameter)

The new dir_index can be the following:

Upon dir_index=0000h (Start GUI) or 0001..000Fh (start file), a request flag in ComFlags.11 is set, the GUI does handle that request, but the Kernel doesn't handle it (so it must be handled in the game; ie. check ComFlags.11 in your mainloop, and call DoExecute when that bit is set, there's no need to call PrepareExecute, since that was already done by the BU Command).
Caution: When dir_index=0000h, then \<param> should be a value that does NOT erase the RTC time/date (eg. 10h or 20h) (most other values do erase the RTC, see SWI 08h for details).
Upon dir_index=FFFEh, a similar request flag is set in ComFlags.30, and, the Kernel (not the GUI) does handle that request in its FIQ handler (however, the request is: To reset the RTC time/date and to start the GUI with uninitialized irq/svc stack pointers, so this unpleasant and bugged feature shouldn't ever be used). Finally, dir_index=FFFFh allows to read the current dir_index value (which could be also read via BU Command 5Ah).

BU Command 5Ah (Get Dir_index, ComFlags, F_SN, Date, and Time)

At midnight, the function may accidently return the date for the old day, and the time for the new day.

BU Command 5Eh (Get-and-Send ComFlags.bit1,3,2)

BU Command 5Fh (Get-and-Send ComFlags.bit0)

13.21 Pocketstation BU Custom Pocketstation Commands

BU Command 5Bh (Execute Function and transfer data from Pocketstation to PSX)

See below for more info on the FUNC value and the corresponding functions.

BU Command 5Ch (Execute Function and transfer data from PSX to Pocketstation)

See below for more info on the FUNC value and the corresponding functions.

BU Command 5Dh (Execute Custom Download Notification Function)

Can be used to notify the GUI (or games that do support this function) about following "download" operations (or uploads or other BU commands).
BU commands are handled inside of the kernels FIQ handler, that means both IRQs and FIQs are disabled during a BU command transmission, so any IRQ or FIQ based audio frequency generators will freeze during BU commands. To avoid distorted noise, it's best to disable sound for the duration specified in bit0-7. If the PSX finishes before the originally specified duration has expired, then it can resend this command with bit8=1 to notify the pocketstation that the "download" has completed.

The Download Notification callback address can be set via SWI 01h, SetCallbacks(3,proc), see there for details. At kernel side, the function execution is like so:

In the GUI, the bu_cmd_5dh_hook/callback handles parameter bits as so (and games should probably handle that bits in the same fashion, too):

If PSX games send any of the standard commands (52h,53h,57h) to access the memory card without using command 5Dh, then GUI automatically sets the duration to 01h (and pauses sound only for that short duration).

FUNC 00h - Get or Set Date/Time (FUNC0)

LEN1 is 00h (no parameters), and LEN2 is 08h (eight data bytes):

At midnight, the function may accidently return the date for the old day, and the time for the new day.

FUNC 01h - Get or Set Memory Block (FUNC1)

LEN1 is 05h (five parameters bytes):

LEN2 is variable (using the 5th byte of the above parameters):

Can be used to write to RAM (and eventually also to I/O ports; when you know what you are doing). In the read direction it can read almost anything: RAM, BIOS ROM, I/O Ports, Physical and Virtual FLASH memory. Of which, trying to read unmapped Virtual FLASH does probably (?) cause a Data Abort exception (and crash the Pocketstation), so that region may be read only if a file is loaded (check that dir_index isn't zero, via BU Command 5Ah, and, take care not to exceed the filesize of that file).
BUG: When sending more than 2 data bytes in the PSX-to-Pocketstation direction, then ADDR must be word-aligned (the BIOS tries to handle odd destination addresses, but when doing that, it messes up the alignment of another internal pointer).

FUNC 02h - Get or Set Alarm/Flags (FUNC2)

LEN1 is 00h (no parameters), and LEN2 is 08h (eight data bytes):

Changing the alarm value while the GUI is running works only with some trickery: For a sinister reason, the GUI copies the alarm setting to User RAM when it gets started, that copy isn't affected by FUNC2, so the GUI believes that the old alarm setting does still apply (and writes that old values back to Kernel RAM when leaving the GUI). The only workaround is:
Test if the GUI is running, if so, restart it via Command 59h (with dir_index=0, and param=0120h or similar, ie. with param.bit8 set), then execute FUNC2, then restart the GUI again (this time with param.bit8 zero).

FUNC 03h - Custom Function 3 (aka FUNC3)

LEN1 is 04h (fixed) (four parameters bytes):

LEN2 is variable (depends on the return value of the 1st function call):

The function address can be set via SWI 17h, GetPtrToFunc3addr(), see there for details. Before using FUNC 03h one must somehow ensure that the desired file is loaded (and that it does have initialized the function address via SWI 17h, otherwise the pocketstation would crash).
The FUNC3 address is automatically reset to 0000h when (if) SWI 05h (SenseAutoCom) senses new docking.
Note: The POC-XBOO circuit uses FUNC3 to transfer TTY debug messages.

FUNC 80h..FFh - Custom Function 80h..FFh

LEN1 is variable (depends on the LEN1 value in Function Table in File Header):

LEN2 is variable (depends on the return value of the 1st function call):

The function addresses (and LEN1 values) are stored in the Function Table FLASH memory (see Pocketstation File Header for details).

Before using FUNC 80h..FFh one must somehow ensure that the desired file is loaded (ie. that the function table with the desired functions is mapped to flash memory; otherwise the pocketstation would crash).

First Function Call (Pre-Data)

Incoming parameters on 1st Function Call:

Return Value on 1st Function Call:

That 64bits are:

dst is written in 8bit units
src is read in 16bit units (and then split to 8bit units)

Second Function Call (Post-Data)

Incoming parameters on 2nd Function Call:

Return Value on 2nd Function Call:

Function flags (r0)

For each function, there is only one single function vector which is called for both To- and From-PSX, and both Pre- and Post-Data, and also on errors. The function must decipher the flags in r0 to figure out which of that operations it should handle:

There are only six possible flags combinations:

The kernel doesn't call FUNC 03h if the Error bit is set (ie. Post-Bad-Data needs to be handled only by FUNC 80h..FFh, not by FUNC 03h.)

13.22 Pocketstation File Header/Icons

Pocketstation File Content

Pocketstation files consists of the following elements (in that order):

The Title sector contains some information about the size of the above regions, but not about their addresses (ie. to find a given region, one must compute the size of the preceeding regions).

Special "P" Filename in Directory Sector

For pocketstation executables, the 7th byte of the filename must be a "P" (for other files that location does usually contain a "-", assuming the file uses a standard filename, eg. "BESLES-12345abcdefgh" for a Sony licensed european title).

Special Pocketstation Entries in the Title Sector at [50h..5Fh]

In normal PSX files, the region at 50h..5Fh is usually zerofilled. For more info on the standard entries in the Title Sector (and for info on Directory Entries), see:
Memory Card Data Format

Snapshot Region (in "MCX1" Files only)

For a load-able snapshot the Snapshot ID must be 01h,00h,"SE", the Kernel uses snapshots only once (after loading a snapshot, it forcefully changes the ID to 00h, 00h,"SE" in FLASH memory).

For MCX1 files, snapshots can be automatically loaded and saved via the SWI 09h, DoExecute function (the snapshot handling seems to be bugged though; see SWI 09h for details).

Function Table (FUNC 80h..FFh)

The table can contain 00h..7Fh entries, for FUNC 80h..FFh. Each entry occupies 8 bytes:

If the number of table entries isn't a multiple of 10h, then the table should be zero-padded to a multiple of 80h bytes (the following File Viewer Mono Icon data is located on the next higher 80h-byte boundary after the Function Table).
For details see BU Commands 5Bh and 5Ch.

File Viewer Mono Icon

Animation Length (0001h..any number) (icon frames) is stored in hdr[50h], for the File Viewer Icon, the Animation Delay is fixed (six 30Hz units per frame).
The File Viewer Icon is shown in the Directory Viewer (which is activated when holding

the Down-button pressed for some seconds in the GUI screen with the speaker and memory card symbols, and which shows icons for all files, including regular PSX game positions, whose colored icons are converted without any contrast optimizations to unidentify-able dithered monochrome icons). If the animation length of the File Viewer Icon is 0000h, then the Directory Viewer does instead display the first Executable Mono Icon.
Each icon frame is 32x32 pixels with 1bit color depth (32 words, =128 bytes),

A normal icon occupies 80h bytes, animated icons have more than one frame and do occupy N*80h bytes.

Executable Mono Icon List

The number of entries in the Executable Mono Icon List is specified in hdr[56h] (usually 01h). Each entry in the Icon List occupies 8 bytes:

The icon frame(s) can be anywhere on a word-aligned location in the file Body (as specified in the above Address entry), the format of the frame(s) is the same as for File Viewer Mono Icons (see there).
The Executable Icons are shown in the Executable File Selection Menu (which occurs when pressing Left/Right buttons in the GUI). Pressing Fire button in that menu starts the selected executable. If the Icon List has more than 1 entry, then pressing Up/Down buttons moves to the previous/next entry (this just allows to view the corresponding icons, but doesn't have any other purpose, namely, the current list index is NOT passed to the game when starting it).
The Executable Mono Icon List is usually zero-padded to 80h-bytes size (although that isn't actually required, the following file Body could start at any location).

Entrypoint

The whole file (including the header and icons) gets mapped to 02000000h and up. The entrypoint can be anywhere in the file Body, and it gets called with a parameter value in r0 (when started by the GUI, that parameter is always zero, but it may be nonzero when

the executable was started by a game, ie. the \<param> from SWI 08h, PrepareExecute, or the \<param> from BU Command 59h).
Caution: Games (and GUI) are started with the ARM CPU running in Non-privileged User Mode (however, there are several ways to hook IRQ/FIQ handlers, and from there one can switch to Privileged System Mode).

Returning to the GUI

Games should always include a way to return to the GUI (eg. an option in the game over screen, a key combination, a watchdog timer, and/or the docking signal) (conventionally, games should prompt Exit/Continue when holding Fire pressed for 5 seconds), otherwise it wouldn't be possible to start other games - except by pushing the Reset button (which is no good idea since the bizarre BIOS reset handler does reset the RTC time for whatever reason).
The kernel doesn't pass any return address to the entrypoint (neither in R14, nor on stack). To return control to the GUI, use SWI functions PrepareExecute(1,0,GetDirIndex()+30h), and then DoExecute(0).

13.23 Pocketstation File Images

Pocketstation files are normally stored in standard Memory Card images,
Memory Card Images

Pocketstation specific files

Aside from that standard formats, there are two Pocketstation specific formats, the "SC" and "SN" variants. Both contain only the raw file, without any Directory sectors, and thus not including a "BESLESP12345"-style filename string. The absence of the filename means that a PSX game couldn't (re-)open these files via filenames, so they are suitable only for "standalone" pocketstation games.

Pocketstation .BIN Files ("SC" variant)

Contains the raw Pocketstation Executable (ie. starting with the "SC" bytes in the title sector, followed by icons, etc.), the filesize should be padded to a 2000h-byte block boundary.

Pocketstation .BIN Files ("SN" variant)

This is a strange incomplete .BIN file variant which starts with a 4-byte ID ("SN",00h, 00h), which is directly followed by executable code, without any title sector, and without any icons.

The filesize is don't care (no padding to block, sector, word, or halfword boundaries required).

13.24 Pocketstation XBOO Cable

This circuit allows to connect a pocketstation to PC parallel port, allowing to upload executables to real hardware, and also allowing to download TTY debug messages (particulary useful as the 32x32 pixel LCD screen is way too small to display any detailed status info).

POC-XBOO Circuit

Use a standard parallel port cable (with 36pin centronics connector or 25pin DB connector) and then build a small adaptor like this:

The circuit is same as for "Direct Pad Pro" (but using a memory card connector instead of joypad connector, and needing +5V from PC power supply instead of using parallel port D3..D7 as supply). Note: IRQ7 is optional (for faster/early timeout).

POC-XBOO Upload

The upload function is found in no$gba "Utility" menu. It does upload the executable and autostart it via standard memory card/pocketstation commands (ie. it doesn't require any special transmission software installed on the pocketstation side).
Notes: Upload is overwriting ALL files on the memory card, and does then autostart the first file. Upload is done as "read and write only if different", this provides faster transfers and higher lifetime.

POC-XBOO TTY Debug Messages

TTY output is conventionally done by executing the ARM CPU's Undefined Opcode with an ASCII character in R0 register (for that purpose, the undef opcode handler should simply point to a MOVS PC,LR opcode).
That kind of TTY output works in no$gba's pocketstation emulation. It can be also used via no$gba's POC-XBOO cable, but requires some small customization in the executable:
First of, the executable needs "TTY+" ID in some reserved bytes of the title sector (telling the xboo uploader to stay in transmission mode and to keep checking for TTY messages after the actual upload):

With that ID, and with the XBOO-hardware being used, the game will be started with with "TTY+" in R0 (notifying it that the XBOO hardware is present, and that it needs to install special transmission handlers):

Usage: Call "init_tty" at the executable's entrypoint (with incoming R0 passed on). Call "tty_wrchr" to output ASCII characters.
Note: The TTY messages are supported only in no$gba debug version (not no$gba gaming version).

14. Serial Port (SIO)

1F801050h SIO_TX_DATA (W)

Writing to this register starts transmit (if, or as soon as, TXEN=1 and CTS=on and SIO_STAT.2=Ready). Writing to this register while SIO_STAT.0=Busy causes the old value to be overwritten.
The "TXEN=1" condition is a bit more complex: Writing to SIO_TX_DATA latches the current TXEN value, and the transfer DOES start if the current TXEN value OR the latched TXEN value is set (ie. if TXEN gets cleared after writing to SIO_TX_DATA, then the transfer may STILL start if the old latched TXEN value was set; this appears for SIO transfers in Wipeout 2097).

1F801050h SIO_RX_DATA (R)

A data byte can be read when SIO_STAT.1=1. Data should be read only via 8bit memory access (the 16bit/32bit "preview" feature is rather unusable).

1F801054h SIO_STAT (R)

Note: Bit0 gets cleared after sending the Startbit, Bit2 gets cleared after sending all bits up to including the Stopbit.

1F801058h SIO_MODE (R/W) (eg. 004Eh --> 8N1 with Factor=MUL16)

1F80105Ah SIO_CTRL (R/W)

1F80105Ch SIO_MISC (R/W)

This is an internal register, which usually shouldn't be accessed by software. Messing with it has rather strange effects: After writing a value "X" to this register, reading returns "X ROR 8" eventually "ANDed with 1F1Fh and ORed with C0C0h or 8080h" (depending on the character length in SIO_MODE).

1F80105Eh SIO_BAUD (R/W) (eg. 00DCh --> 9600 bauds; when Factor=MUL16)

The Baudrate is calculated (based on SIO_BAUD, and on Factor in SIO_MODE):

SIO_TX_DATA Notes

The hardware can hold (almost) 2 bytes in the TX direction (one being currently transferred, and, once when the start bit was sent, another byte can be stored in SIO_TX_DATA). When writing to SIO_TX_DATA, both SIO_STAT.0 and SIO_STAT.2 become zero. As soon as the transfer starts, SIO_STAT.0 becomes set (indicating that one can write a new byte to SIO_TX_DATA; although the transmission is still busy). As soon as the transfer of the most recently written byte ends, SIO_STAT.2 becomes set.

SIO_RX_DATA Notes

The hardware can hold 8 bytes in the RX direction (when receiving further byte(s) while the RX FIFO is full, then the last FIFO entry will by overwritten by the new byte, and SIO_STAT.4 gets set; the hardware does NOT automatically disable RTS when the FIFO becomes full).
Data can be read from SIO_RX_DATA when SIO_STAT.1 is set, that flag gets automatically cleared after reading from SIO_RX_DATA (unless there are still further bytes in the RX FIFO). Note: The hardware does always store incoming data in RX FIFO (even when Parity or Stop bits are invalid).
Note: A 16bit read allows to read two FIFO entries at once; nethertheless, it removes only ONE entry from the FIFO. On the contrary, a 32bit read DOES remove FOUR entries (although, there's nothing that'd indicate if the FIFO did actually contain four entries).
Reading from Empty RX FIFO returns either the most recently received byte or zero (the hardware stores incoming data in ALL unused FIFO entries; eg. if five entries are used, then the data gets stored thrice, after reading 6 bytes, the FIFO empty flag gets set, but nethertheless, the last byte can be read two more times, but doing further reads returns 00h).

Interrupt Acknowledge Notes

First reset I_STAT.8, then set SIO.CTRL.4 (when doing it vice-versa, the hardware may miss a new IRQ which may occur immediately after setting SIO.CTRL.4) (and I_STAT.8 is edge triggered, so that bit can be reset even while SIO_STAT.9 is still set).
When acknowledging via SIO_CTRL.4 with the enabled condition(s) in SIO_CTRL.10-12 still being true (eg. the RX FIFO is still not empty): the IRQ does trigger again (almost) immediately (it goes off only for a very short moment; barely enough to allow I_STAT.8 to sense a edge).

SIO_BAUD Notes

Timer reload occurs when writing to SIO_BAUD, and, automatically when the Baudrate Timer reaches zero. There should be two 16bit SIO timers (for TX and RX), the upper 15bit of one of that timers can be read from SIO_STAT (not sure which one, and no idea if there's a way to read the other timer, too).
Or... maybe there is only ONE timer, and RX/TX are separated only by separate "timer ellapsed" counters, in that case the MUL1 factor won't work properly, but, with the MUL16 or MUL64 factors, RX could start anytime (eg. when TX has already ellapsed a bunch of times)...?
The maximum baud rate may vary depending on the length and quality of the cable, whether and how many inverters and anti-inverters are used (on the mainboard and in external adaptor, and on whether signals are externally converted to +/-12V levels)... anyways, rates up to 9600 baud should be working in all cases.
However, running in no$psx, Wipeout 2097 seems to use about 2 million bauds... although, in older no$psx versions, I believe I did see it using some kind of baudrate detection, where it did try different rates in steps of 200 bauds or so...?

SIO Ports vs JOY Ports

SIO uses I/O Addresses 1F801050h..1F80105Fh, which seem to be organized similar to the Controller/Memory Card registers at 1F801040h..1F80104Fh, though not identical, and with an additional register at 1F80105Ch, which has no corresponding port at 1F80104Ch.
SIO_BAUD is \<effectively> same as for JOY_BAUD, but, \<internally> they are a bit different:

Unlike for the Controller/Memory Card ports, the data is transferred without CLK signal, instead, it's using RS232 format, ie. the transfer starts with a start bit, and is then transferred at a specific baudrate (which must be configured identically at the receiver side). For RS232, data is usually 8bit, and may optionally end with a parity bit, and one or two stop bits.

Note

For SIO Pinouts, PSone SIO upgrading, and for building RS232 adaptors, see:
Pinouts - SIO Pinouts

Aside from the internal SIO port, the PSX BIOS supports two additional external serial ports, connected to the expansion port,
EXP2 Dual Serial Port (for TTY Debug Terminal)

SIO Games

The SIO ports on two consoles can be connected with an SCPH-1040 Link Cable (known as Taisen Cable, or "Fight Cable" in Japan) for multiplayer functionality on games that support this method. This was used by a small number of games in the console's lifecycle, but inconveniently required a second console and copy of the game.

Two-Console Link Cable Games (Incomplete List):

The serial port is used (for 2-player link) by Wipeout 2097 (that game accidently assumes BAUDs based on 64*1024*1025 Hz rather than on 600h*44100 Hz).
Ridge Racer Revolution is also said to support 2P link.
Keitai Eddy seems to allow to connect a mobile phone to the SIO port (the games CD cover suggests so; this seems to be something different than the "normal" I-Mode adaptor, which would connect to controller port, not to SIO port).

8251A Note

The Playstation Serial Port is apparently based/inspired on the Intel 8251A USART chip; which has very similar 8bit Mode/Command/Status registers.

15. Expansion Port (PIO)

Expansion Port can contain ROM, RAM, I/O Ports, etc. For ROM, the first 256 bytes would contain the expansion ROM header.

For region 1, the CPU outputs a chip select signal (CPU Pin 98, /EXP).
For region 2, the CPU doesn't produce a chip select signal (the region is intended to contain multiple I/O ports, which require an address decoder anyways, that address decoder could treat any /RD or /WR with A13=Hi and A23=Hi and A22=Lo as access to expansion region 2 (for /WR, A22 may be ignored; assuming that the BIOS is read-only).

Size/Bus-Width

The BIOS initalizes Expansion Region 1 to 512Kbyte with 8bit bus, and Region 2 to 128 bytes with 8bit bus. However, the size and data bus-width of these regions can be changed, see:
Memory Control
For Region 1, 32bit reads are supported even in 8bit mode (eg. 32bit opcode fetches are automatically processed as four 8bit reads).
For Region 2, only 8bit access seems to be supported (except that probably 16bit mode allows 16bit access), anyways, larger accesses seem to cause exceptions... not sure if that can be disabled...?

Expansion 1 - EXP1 - Intended to contain ROM

EXP1 Expansion ROM Header

Expansion 2 - EXP2 - Intended to contain I/O Ports

EXP2 Dual Serial Port (for TTY Debug Terminal)
EXP2 DTL-H2000 I/O Ports
EXP2 Post Registers
EXP2 Nocash Emulation Expansion

Expansion 3 - EXP3 - Intended to contain RAM

Not used by BIOS nor by any games. Seems to contain 1Mbyte RAM with 16bit databus (ie. 512Kx16) in DTL-H2000.

Other Expansions

Aside from the above, the Expansion regions can be used for whatever purpose, however, mind that the BIOS is reading from the ROM header region, and is writing to the POST register (so 1F000000h-1F0000FFh and 1F802041h should be used only if the hardware isn't disturbed by those accesses).

Missing Expansion Port

The expansion port is installed only on older PSX boards, newer PSX boards and all PSone boards don't have that port. However, the CPU should still output all expansion signals, and there should be big soldering points on the board, so it'd be easy to upgrade the console.

Latched Address Bus

Note that A0..A23 are latched outputs, so they can be used as general purpuse 24bit outputs, provided that the system bus isn't used for other purposes (such like /BIOS, /SPU, /CD accesses) (A0..A23 are not affected by Main RAM and GPU addressing, nor by internal I/O ports like Timer and IRQ registers).

15.1 EXP1 Expansion ROM Header

Expansion 1 - ROM Header (accessed with 8bit databus setting)

The entrypoints are called if their corresonding ID strings are present, return address to BIOS is passed in R31, so the expansion ROM may return control to BIOS, if that should

be desired.
Aside from verifying the IDs, the BIOS will also display the Post-Boot ID string (and the following message string) via TTY (done right before calling the Post-Boot Entrypoint).

Pre-Boot Function

The Pre-Boot function is called almost immediately after Reset, with only some Memory Control registers initialized, the BIOS function vectors at A0h, B0h, and C0h are NOT yet initialized, so the Pre-Boot function can't use them.

Post-Boot Function

The Post-Boot function gets called while showing the "PS" logo, but before loading the .EXE file. The BIOS function vectors at A0h, B0h, and C0h are already installed and can be used by the Post-Boot Function.
Note that the Post-Boot Function is called ONLY when the "PS" logo is shown (ie. not if the CDROM drive is empty, or if it contains an Audio CD).

Mid-Boot Hook

One common trick to hook the Kernel after BIOS initialization, but before CDROM loading is to use the Pre-Boot handler to set a COP0 opcode fetch hardware breakpoint at 80030000h (after returning from the Pre-Boot handler, the Kernel will initialize important things like A0h/B0h/C0h tables, and will then break again when starting the GUI code at 80030000h) (this trick is used by Action Replay v2.0 and up).

Note

Expansion ROMs are most commonly used in cheat devices,
Cheat Devices

15.2 EXP2 Dual Serial Port (for TTY Debug Terminal)

SCN2681 Dual Asynchronous Receiver/Transmitter (DUART)

The PSX/PSone retail BIOS contains some TTY Debug Terminal code; using an external SCN2681 chip which can be connected to the expansion port.
Whilst supported by all PSX/PSone retail BIOSes on software side, there aren't any

known PSX consoles/devboards/expansions actually containing DUARTs on hardware side.

1F802023h/Read - RHRA - DUART Rx Holding Register A (FIFO) (R)

1F80202Bh/Read - RHRB - DUART Rx Holding Register B (FIFO) (R)

1F802023h/Write - THRA - DUART Tx Holding Register A (W)

1F80202Bh/Write - THRB - DUART Tx Holding Register B (W)

The hardware can hold max 2 Tx characters per channel (1 in the THR register, and one currently processed in the Tx Shift Register), and max 4 Rx characters (3 in the RHR FIFO, plus one in the Rx Shift Register) (when receiving a 5th character, the "old newest" value in the Rx Shift Register is lost, and the overrun flag is set).

1F802020h/FirstAccess - MR1A - DUART Mode Register 1.A (R/W)

1F802028h/FirstAccess - MR1B - DUART Mode Register 1.B (R/W)

Note: In block error mode, block error conditions must be cleared by using the error reset command (command 4) or a receiver reset (command 2).

1F802020h/SecondAccess - MR2A - DUART Mode Register 2.A (R/W)

1F802028h/SecondAccess - MR2B - DUART Mode Register 2.B (R/W)

Stop Bit Lengths:

Add 0.5 to values shown for 0..7 if channel is programmed for 5 bits/char.

1F802021h/Write - CSRA - DUART Clock Select Register A (W)

1F802029h/Write - CSRB - DUART Clock Select Register B (W)

The 2681 has some sets of predefined baud rates (set1/set2 selected via ACR.7), additionally, in BRG Test Mode, set3/set4 are used instead of set1/set2), the baud rates for selections 00h..0Dh are:

Selection 0Eh/0Fh are using an external clock source (derived from IP3,IP4,IP5,IP6 pins; for TxA,RxA,TxB,RxB respectively).

1F802022h/Write - CRA - DUART Command Register A (W)

1F80202Ah/Write - CRB - DUART Command Register B (W)

The command values for CRA (or CRB) are:

Access to the upper four bits of the command register should be separated by 3 edges of the X1 clock. A disabled transmitter cannot be loaded.

1F802025h/Read - ISR - DUART Interrupt Status Register (R)

1F802025h/Write - IMR - DUART Interrupt Mask Register (W)

1F802021h/Read - SRA - DUART Status Register A (R)

1F802029h/Read - SRB - DUART Status Register B (R)

Bit7-5 are appended to the corresponding data character in the receive FIFO. A read of the status provides these bits (7:5) from the top of the FIFO together with bits (4:0). These bits are cleared by a "reset error status" command. In character mode they are discarded when the corresponding data character is read from the FIFO. In block error mode, block error conditions must be cleared by using the error reset command (command 4x) or a receiver reset.

1F802024h/Write - ACR - DUART Aux. Control Register (W)

Counter/Timer Mode and Clock Source Settings:

In Counter Mode, the Counter Ready flag is set on any underflow, and the counter wraps to FFFFh and keeps running (but may get stopped by software).
In Timer Mode, automatic reload occurs on any underflow, the counter flag (which can be output to OP3) is toggled on any underflow, but the Counter Ready flag is set only on each 2nd underflow (unlike as in Counter mode).

1F802024h/Read - IPCR - DUART Input Port Change Register (R)

Reading from this register automatically resets IPCR.7-4 and ISR.7.

1F80202Dh/Read - IP - DUART Input Port (R)

IP0-6 can be used as general purpose inputs, or for following special purposes:

Note: The 24pin chip doesn't have any inputs, the 28pin chip has only one input (IP2), the 40pin/44pin chips have seven inputs (IP0-IP6).

1F80202Eh/Write - DUART Set Output Port Bits Command (Set means Out=LOW)

1F80202Fh/Write - DUART Reset Output Port Bits Command (Reset means Out=HIGH)

Note: The 24pin chip doesn't have any outputs, the 28pin chip has only two outputs (OP0,OP1), the 40pin/44pin chips have eight outputs (OP0-OP7).

1F80202Dh/Write - OPCR - DUART Output Port Configuration Register (W)

Additionally, the OP0 and OP1 outputs are controlled via MR2A.5 and MR2B.5.

1F802022h/Read - - DUART Toggle Baud Rate Generator Test Mode (Read=Strobe)

1F80202Ah/Read - - DUART Toggle 1X/16X Test Mode (Read=Strobe)

BGR Test switches between Baud Rate Set1/Set2 and Set3/Set4.
1X/16X Test switches between whatever...?

1F80202Eh/Read - CT_START - DUART Start Counter Command (Read=Strobe)

1F80202Fh/Read - CT_STOP - DUART Stop Counter Command (Read=Strobe)

Start: Forces reload (copies CTLR/CTUR to CTL/CTU), and starts the timer.
Stop-in-Counter-Mode: Resets ISR.3, and stops the timer.
Stop-in-Timer-Mode: Resets ISR.3, but doesn't stop the timer.

1F802026h/Read - CTU - DUART Counter/Timer Current Value, Upper/Bit15-8 (R)

1F802027h/Read - CTL - DUART Counter/Timer Current Value, Lower/Bit7-0 (R)

1F802026h/Write - CTUR - DUART Counter/Timer Reload Value, Upper/Bit15-8 (W)

1F802027h/Write - CTLR - DUART Counter/Timer Reload Value, Lower/Bit7-0 (W)

The CTLR/CTUR reload value is copied to CTL/CTU upon Start Counter Command. In Timer mode (not in Counter mode), it is additionally copied automatically when the timer undeflows.

1F80202Ch - N/A - DUART Reserved Register (neither R nor W)

Reserved.

Chip versions

The SCN2681 is manufactured with 24..44 pins, the differences are:

Unknown which of them is supposed to be used with the PSX?
Note: The Motorola 68681 should be the same as the Philips/Signetics 2681.

Notes

Unknown if the Interrupt signal is connected to the PSX... there seems to be no spare IRQ for it, though it \<might> share an IRQ with whatever other hardware...?
The BIOS seems to use only one of the two channels; for the std_io functions:
BIOS TTY Console (std_io)
Aside from the external DUART, the PSX additionally contains an internal UART,
Serial Port (SIO)

The DTL-H2000 devboard uses a non-serial "ATCONS" channel for TTY stuff, EXP2 DTL-H2000 I/O Ports

15.3 EXP2 DTL-H2000 I/O Ports

The DTL-H2000 contains extended 8Mbyte Main RAM (instead of normal 2Mbyte), plus additional 1MByte RAM in Expansion Area at 1FA00000h, plus some I/O ports at 1F8020xxh:

1F802000h - DTL-H2000: EXP2: - ATCONS STAT (R)

1F802002h - DTL-H2000: EXP2: - ATCONS DATA (R and W)

TTY channel for message output (TX) and debug console keyboard input (RX). The DTL-H2000 is using this "ATCONS" stuff instead of the DUART stuff used in retail console BIOSes ("CONS" seems to refer to "Console", and "AT" might refer to PC/AT or whatever).

1F802004h - DTL-H2000: EXP2: - 16bit - ?

1F802030h - DTL-H2000: Secondary IRQ10 Controller (IRQ Flags)

This register does expand IRQ10 (Lightgun) to more than one IRQ source. The register contains only Secondary IRQ Flags (there seem to be no Secondary IRQ Enable bits; at least not for Lightguns).

Retail games that support IRQ10-based "Konami" Lightguns are containing code for detecting and accessing port 1F802030h. The detection works by examining a value in the BIOS ROM like so:

Normal consoles don't include Port 1F802030h, and IRQ10 is wired directly to the controller port, and the value at [BFC00104h] is always 00000003h. Accordingly, one cannot upgrade the console just by plugging a Secondary IRQ10 controller to the expansion port (instead, one would need to insert the controller between the CPU and controller plug, and to install a BIOS with [BFC00104h]=00002000h).
The DTL-H2000 BIOS accesses 1F802030h with 8bit load/store opcodes, however, the Lightgun games use 32bit load/store - which is theoretically overlapping port 1F802032h, though maybe the memory system does ignore the upper bits.

1F802032h - DTL-H2000: EXP2: - maybe IRQ enable?

1F802040h - DTL-H2000: EXP2: 1-byte - DIP Switch?

This register selects the DTL-H2000 boot mode, for whatever reason it's called "DIP Switch" register, although the DTL-H2000 boards don't seem to contain any such DIP Switches (instead, it's probably configured via some I/O ports on PC side). Possible values are:

1F802042h - DTL-H2000: EXP2: POST/LED (R/W)

EXP2 Post Registers

15.4 EXP2 Post Registers

1F802041h - POST - External 7-segment Display (W)

During boot, the BIOS writes incrementing values to this register, allowing to display the current boot status on an external 7 segment display (much the same as Port 80h used in PC BIOSes).

1F802042h - DTL-H2000: EXP2: POST/LED (R/W)

8bit wide, otherwise same as POST 1F802041h on retail consoles.

1F802070h - POST2 - Unknown? (W) - PS2

Might be a configuration port, or it's another POST register (which is used prior to writing the normal POST bytes to 1FA00000h).
The first write to 1F802070h is 32bit, all further writes seem to be 8bit.

1FA00000h - POST3 - External 7-segment Display (W) - PS2

Similar to POST, but PS2 BIOS uses this address.

15.5 EXP2 Nocash Emulation Expansion

1F802060h Emu-Expansion ID1 "E" (R)

1F802061h Emu-Expansion ID2 "X" (R)

1F802062h Emu-Expansion ID3 "P" (R)

1F802063h Emu-Expansion Version (01h) (R)

Contains ID and Version.

1F802064h Emu-Expansion Enable1 "O" (R/W)

1F802065h Emu-Expansion Enable2 "N" (R/W)

Activates the Halt and Turbo Registers (when set to "ON").

1F802066h Emu-Expansion Halt (R)

When enabled (see above), doing an 8bit read from this address stops the CPU emulation unless/until an Interrupt occurs (when "CAUSE AND SR AND FF00h" becomes nonzero). Can be used to reduce power consumption, and to make the emulation faster.

1F802067h Emu-Expansion Turbo Mode Flags (R/W)

When enabled (see above), writing to this register activates/deactivates "turbo" mode, which is causing new data to arrive immediately after acknowledging the previous interrupt.

16. Memory Control

The Memory Control registers are initialized by the BIOS, and, normally software doesn't need to change that settings. Some registers are useful for expansion hardware (allowing to increase the memory size and bus width).

1F801000h - Expansion 1 Base Address (usually 1F000000h)

1F801004h - Expansion 2 Base Address (usually 1F802000h)

For Expansion 1, the address is forcefully aligned to the selected expansion size (see below), ie. if the size is bigger than 1 byte, then the lower bit(s) of the base address are ignored.
For Expansion 2, trying to use ANY other value than 1F802000h seems to disable the Expansion 2 region, rather than mapping it to the specified address (ie. Port 1F801004h doesn't seem to work).
For Expansion 3, the address seems to be fixed (1FA00000h).

1F801008h - Expansion 1 Delay/Size (usually 0013243Fh) (512Kbytes, 8bit bus)

1F80100Ch - Expansion 3 Delay/Size (usually 00003022h) (1 byte)

1F801010h - BIOS ROM Delay/Size (usually 0013243Fh) (512Kbytes, 8bit bus)

1F801014h - SPU Delay/Size (200931E1h) (use 220931E1h for SPU-RAM reads)

1F801018h - CDROM Delay/Size (00020843h or 00020943h)

1F80101Ch - Expansion 2 Delay/Size (usually 00070777h) (128 bytes, 8bit bus)

When booting, all these registers are using the maximum cycle delays for both reads and writes. Then, the BIOS will immediately select a faster read access delay, resulting in a visible speed up after the first few instructions. The effects aren't immediate however. The BIOS boots using the following instructions:

When using a logic analyzer to monitor the boot sequence, the instruction at bfc00014 is still read using the old timings since reset, and then the instruction at bfc00018 is finally read using the sped up timings.

Reads and writes access times aren't symmetrical, and are each controlled with their own values. By default, EXP1 will be set to 16 cycles when writing, which is the slowest possible. If the programmer wants to write to a flash chip on EXP1, or communicate with a computer, speeding up write access is recommended.

The fastest a port could go would be by setting the lowest 16 bits to zero, which will result in 3 CPU cycles for a single byte access.

!CS always goes active at least one cycle before !WR or !RD go active. The various timing changes are between all the events inside the data read/write waveform. The whole formula for computing the total access time is fairly complex overall, and difficult to properly describe.

- The pre-strobe period will add delays between the moment the data bus is set, and the moment !CS goes active.
- The hold period will keep the data in the data bus for some more cycles after !WR goes inactive, and before !CS goes inactive. The accessed device is supposed to sample the data bus during this interval.
- The floating period will keep the data bus floating for some more cycles after !RD goes inactive, and before !CS goes inactive. The accessed device is supposed to stop driving the data bus during this interval. The CPU will sample the data bus somewhere before or exactly when !CS goes inactive.
- The recovery period will add delays between two operations.

The data bus width will influence if the CPU does full 16 bits reads, or only 8 bits. When doing 32 bits operations, the CPU will issue 2 16-bits operations, or 4 8-bits operations, keeping !CS active the whole time, and strobing !WR or !RD accordingly. When doing these sequences, the address bus will also increment automatically between each operation, if the auto-increment bit is active.

This means it is possible to slightly shorten the read time of 4 bytes off the same address by disabling auto-increment, and reading a full word. The CPU will then read 4 bytes off the same address, and place them all into each byte of the loaded register.

The DMA timing override portion will replace the access timing when doing DMA, only if the DMA override flag is set.

The Wide DMA flag will enable full 32 bits DMA operations on the bus, by reusing the low 16-bits address signals as the high 16-bits data. This means that if the CPU is doing Wide DMA reads, the low 16-bits of the address bus will become inputs.

Trying to access addresses that exceed the selected size causes a bus exception. Maximum size would be Expansion 1 = 17h (8MB), BIOS = 16h (4MB), Expansion 2 = 0Dh (8KB), Expansion 3 = 15h (2MB). Trying to select larger sizes would overlap the internal I/O ports, and crash the PSX. The Size bits seem to be ignored for SPU/CDROM. The SPU timings seem to be applied for both the 200h-byte SPU region at 1F801C00h and for the 200h-byte unknown region at 1F801E00h.

1F801020h - COM_DELAY / COMMON_DELAY (00031125h or 0000132Ch or 00001325h)

This register contains clock cycle offsets that can be added to the Access Time values in Port 1F801008h..1Ch. Works (somehow) like so:

The total access time is the sum of First Access, plus any Sequential Access(es), eg. for a 32bit access with 8bit bus: Total=1ST+SEQ+SEQ+SEQ.
If the access is done from code in (uncached) RAM, then 0..4 cycles are added to the Total value (the exact number seems to vary depending on the used COMx values or so).

1F801060h - RAM_SIZE (R/W) (usually 00000B88h) (or 00000888h)

Possible values for Bit9-11 are:

The BIOS initializes this to setting 5 (8MB) (ie. the 2MB RAM repeated 4 times), although the "correct" would be setting 4 (2MB, plus other 6MB Locked). The remaining memory, after the first 8MB, and up to the Expansion/IO/BIOS region seems to be always Locked.
The HighZ regions are FFh-filled, that even when grounding data lines on the system bus (ie. it is NOT a mirror of the PIO expansion region).
Locked means that the CPU generates an exception when accessing that area.
Note: Wipeout uses a BIOS function that changes RAM_SIZE to 00000888h (ie. with corrected size of 2MB, and with the unknown Bit8 cleared). Gundam Battle Assault 2 does actually use the "8MB" space (with stacktop in mirrored RAM at 807FFFxxh).
Clearing bit7 causes many games to hang during CDROM loading on both EARLY-PU-8 and LATE-PU-8 (but works on PU-18 through PM-41).

FFFE0130h Cache Control (R/W)

Used by BIOS to initialize cache (in combination with COP0), like so:

Note: FFFE0130h is described in LSI's "L64360" datasheet, chapter 14 (and probably also in their LR33300/LR33310 datasheet, if it were available in internet).

17. Unpredictable Things

Normally, I/O ports should be accessed only at their corresponding size (ie. 16bit read/write for 16bit ports), and of course, only existing memory and I/O addresses should be used. When not recursing that rules, some more or less (un-)predictable things may happen...

I/O Write Datasize

Whereas,

It's somewhat "legit" to use 16bit writes on 16bit registers like RAM_SIZE, I_STAT, I_MASK, and Timer 0-2.
Non-4-byte aligned 8bit/16bit writes to RAM_SIZE do crash (probably because the "(w32)" effect is left-shifting the value, so lower 8bit become zero).
Results on unaligned I/O port writes (via SWL/SWR opcodes) are unknown.

I/O Read Datasize

In most cases, I/O ports can be read in 8bit, 16bit, or 32bit units, regardless of their size, among others allowing to read two 16bit ports at once with a single 32bit read. If there's only one 16bit port within a 32bit region, then 32bit reads often return garbage in the unused 16bits. Also, 8bit or 16bit VRAM data reads via GPUREAD probably won't work? Expansion 2 Region can be accessed only via 8bit reads, and 16bit/32bit reads seem to cause exceptions (or rather: no such exception!) (except, probably 16bit reads are allowed when the region is configured to 16bit databus width).
There are at least some special cases:

```
```

I/O Write Datasize

Performing a 8-bit or 16-bit write (/) will place the entirety of the GPR on the bus, regardless of the write size. Therefore, the data is **not** masked. This has an effect when performing a narrower write to a wider address, for example the DMA controller, but not others such as the CD-ROM controller.

Emulators should therefore treat all access widths as having 32 bits of data, but depending on the device perform masking/splitting (see Memory Control).

The CD audio visualizer (aka Soundscope) in the SCPH-7xxx series of consoles is an example of where this behavior is required, as it issues halfword writes to the DMA controller addresses.

Cache Problems

The functionality of the Cache is still widely unknown. Not sure if DMA transfers are updating or invalidating cache. Cached Data within KSEG0 should be automatically also cached at the corresponding mirrored address in KUSEG and vice versa. Mirrors within KSEG1 (or within KUSEG) may be a different thing, eg. when using addresses spead across the first 8MB region to access the 2MB RAM. Same problems may occor for Expansion and BIOS mirrors, although, not sure if that regions are cached.

Writebuffer Problems

The writebuffer seems to be disabled for the normal I/O area at 1F801000h, however, it appears to be enabled for the Expansion I/O region at 1F802000h (after writing to 1F802041h, the BIOS issues 4 dummy writes to RAM, apparently (?) in order to flush the writebuffer). The same might apply for Expansion Memory region at 1F000000h, although usually that region would contain ROM, so it'd be don't care whether it is write-buffered or not.

CPU Load/Store Problems

XXcpuREG ---> applies ONLY to LOAD (not to store)
Memory read/write opcodes take a 1-cycle delay until the data arrives at the destination, ie. the next opcode should not use the destination register (or more unlikely, the destination memory location) as source operand. Usually, when trying to do so, the second opcode would receive the OLD value - however, if an exception occurs between the two opcodes, then the read/write operation may finish, and the second opcode would probably receive the NEW value.

CPU Register Problems - R1 (AT), R26 (K0), R29 (SP)

Exception handlers cannot preserve all registers, before returning, they must load the return address into a general purpose register (conventionally R26 aka K0), so be careful not to use that register, unless you are 100% sure that no interrupts and no other exceptions can occur. Some exception handlers might also destroy R27 aka K1 (though execption handler in the PSX Kernel leaves that register unchanged).
Some assemblers (not a22i in nocash syntax mode) are internally using R1 aka AT as scratch register for some pseudo opcodes, including for a "sw rx,imm32" pseudo opcode (which is nearly impossible to separate from the normal "sw rx,imm16" opcode), be careful not to use R1, unless you can trust your assembler not to destroy that register behind your back.
The PSX Kernel uses "Full-Decrementing-Wasted-Stack", where "Wasted" means that when calling a sub-function with N parameters, then the caller must pre-allocate N works on stack, and the sub-function may freely use and destroy these words; at [SP+0..N*4-1].

Locked Locations in Memory and I/O Area

Trying to access these locations generates an exception. For KSEG0 and KSEG1, locked regions are same as for first 512MB of KUSEG.

Mirrors in I/O Area

Read/writeable mirrors of DMA Control registers at 1F801088h+N*10h.

Garbage Locations in I/O Area

Unlike all other unused I/O addresses, these addresses are unlocked (ie. they do not trigger exceptions on access), however they do not seem to contain anything useful. The BIOS never seems to use them. Writing any values to them seems to have no effect. And reading acts somewhat unstable:
Usually returns zeros in most cases. Except that, the first byte on a 10h-byte boundary often returns the lower 8bit of the memory address (eg. [FFFE0010h]=10h). And, when [FFFE0130h].Bit11=0, then reading from these registers does return the 32bit opcode that is to be executed next (or at some locations, the opcode thereafter).

PSX as Abbreviation for Playstation 1

In gaming and programming scene, "PSX" is most commonly used as abbreviation for the original Playstation series (occasionally including PSone). Sony has never officially used that abbreviation, however, the Playstation BIOS contains the ASCII strings "PSX" and "PS-X" here and there. The letters "PS" are widely believed to stand for PlayStation, and the meaning of the "X" is totally unknown (although, actually it may stand for POSIX.1, see below).

PSX as Abbreviation for POSIX.1

According to JMI Software Systems, "PSX" is a trademark of themselves, and stands for "single-user, single-group, subset of POSIX.1" (POSIX stands for something commonly used by HLL programmers under UNIX or so). That "PSX" kernel from JMI is available for various processors, including MIPS processors, and like the playstation, it does include functions like "atoi", and does support TTY access via Signetics 2681 DUART chips. The DTL-H2000 does also have POSIX-style "PSX>" prompt. So, altogether, it's quite possible that Sony has licensed the kernel from JMI.

PSX as Abbreviation for an Extended Playstation 2

As everybody agrees, PSX should be used only as abbreviation for Playstation 1, and nobody should never ever use it for the Playstation 2. Well, nobody, except Sony... despite of the common use as abbreviation for Playstation 1 (and despite of the JMI trademark)... in 2003, Sony has have released a "Playstation 2 with built-in HDD/DVD Videorecorder" and called that thing "PSX" for the best of confusion.

18. CPU Specifications

CPU

System Control Coprocessor (COP0)

18.1 CPU Registers

All registers are 32bit wide.

R0 is always zero.
R31 can be used as general purpose register, however, some opcodes are using it to

store the return address: JAL, BLTZAL, BGEZAL. (Note: JALR can optionally store the return address in R31, or in R1..R30. Exceptions store the return address in cop0r14 - EPC).

R29 (SP) - Full Decrementing Wasted Stack Pointer

The CPU doesn't explicitly have stack-related registers or opcodes, however, conventionally, R29 is used as stack pointer (SP). The stack can be accessed with normal load/store opcodes, which do not automatically increase/decrease SP, so the SP register must be manually modified to (de-)allocate data.
The PSX BIOS is using "Full Decrementing Wasted Stack".
Decrementing means that SP gets decremented when allocating data (that's common for most CPUs) - Full means that SP points to the first ALLOCATED word on the stack, so the allocated memory is at SP+0 and above, free memory at SP-1 and below, Wasted means that when calling a sub-function with N parameters, then the caller must pre-allocate N works on stack, and the sub-function may freely use and destroy these words; at [SP+0..N*4-1].

For example, "push ra,r16,r17" would be implemented as:

where the allocated 20h bytes have the following purpose:

18.2 CPU Opcode Encoding

Primary opcode field (Bit 26..31)

Secondary opcode field (Bit 0..5) (when Primary opcode = 00h)

Opcode/Parameter Encoding

Coprocessor Opcode/Parameter Encoding

Illegal Opcodes

All opcodes that are marked as "N/A" in the Primary and Secondary opcode tables are causing a Reserved Instruction Exception (excode=0Ah).
The unused operand bits (eg. Bit21-25 for LUI opcode) should be usually zero, but do not necessarily trigger exceptions if set to nonzero values.

18.3 CPU Load/Store Opcodes

Load instructions

Load instructions can read from the data cache (if the data is not in the cache, or if the memory region is uncached, then the CPU gets halted until it has read the data) (however, the PSX doesn't have a data cache).

Caution - Load Delay

The loaded data is NOT available to the next opcode, ie. the target register isn't updated until the next opcode has completed. So, if the next opcode tries to read from the load destination register, then it would (usually) receive the OLD value of that register (unless an IRQ occurs between the load and next opcode, in that case the load would complete during IRQ handling, and so, the next opcode would receive the NEW value).

Store instructions

Store operations are passed to the write-buffer, so they can execute within a single clock cycle (unless the write-buffer was full, in that case the CPU gets halted until there's room in the buffer). But, the PSX doesn't have a writebuffer...?

Load/Store Alignment

Halfword addresses must be aligned by 2, word addresses must be aligned by 4, trying to access mis-aligned addresses will cause an exception. There's no alignment restriction for bytes.

Unaligned Load/Store

There's no delay required between lwl and lwr, so you can use them directly following eachother, eg. to load a word anywhere in memory without regard to alignment:

Unaligned Load/Store (Details)

LWR/SWR transfers the right (=lower) bits of Rt, up-to 32bit memory boundary:

LWL/SWL transfers the left (=upper) bits of Rt, down-to 32bit memory boundary:

The CPU has four separate byte-access signals, so, within a 32bit location, it can transfer all fragments of Rt at once (including for odd 24bit amounts). The transferred data is not zero- or sign-expanded, eg. when transferring 8bit data, the other 24bit of Rt and [mem] will remain intact.

Note: The aligned variant can also misused for blocking memory access on aligned addresses (in that case, if the address is known to be aligned, only one of the opcodes are needed, either LWL or LWR).... Uhhhhhhhm, OR is that NOT allowed... more PROBABLY that doesn't work?

18.4 CPU ALU Opcodes

arithmetic instructions

The opcodes "with overflow trap" do trigger an exception (and leave rd unchanged) in case of overflows.

comparison instructions

logical instructions

shifting instructions

Unlike many other opcodes, shifts use 'rt' as second (not third) operand.
The hardware does NOT generate exceptions on SHL overflows.

Multiply/divide

The mul/div opcodes are starting the multiply/divide operation, starting takes only a single clock cycle, however, trying to read the result from the hi/lo registers while the mul/div operation is busy will halt the CPU until the mul/div has completed. For multiply, the execution time depends on rs (ie. "small*large" can be much faster than "large*small").

For example, when executing "umul 123h,12345678h" and "mov r1,lo", one can insert up to six (cached) ALU opcodes, or read one value from PSX Main RAM (which has 6 cycle access time) between the "umul" and "mov" opcodes without additional slowdown. The hardware does NOT generate exceptions on divide overflows, instead, divide errors are returning the following values:

For udiv, the result is more or less correct (as close to infinite as possible). For sdiv, the results are total garbage (about farthest away from the desired result as possible). Note: After accessing the lo/hi registers, there seems to be a strange rule that one should not touch the lo/hi registers in the next 2 cycles or so... not yet understood if/ when/how that rule applies...?

18.5 CPU Jump Opcodes

jumps and branches

Note that the instruction following the branch will always be executed.

JALR cautions

Caution: The JALR source code syntax varies (IDT79R3041 specs say "jalr rs,rd", but MIPS32 specs say "jalr rd,rs"). Moreover, JALR may not use the same register for both operands (eg. "jalr r31,r31") (doing so would destroy the target address; which is normally no problem, but it can be a problem if an IRQ occurs between the JALR opcode and the following branch delay opcode; in that case BD gets set, and EPC points "back" to the JALR opcode, so JALR is executed twice, with destroyed target address in second execution).

exception opcodes

Unlike for jump/branch opcodes, exception opcodes are immediately executed (ie. without executing the following opcode).

The 20bit immediate doesn't affect the CPU (however, the exception handler may interprete it by software; by examing the opcode bits at [epc-4]).

18.6 CPU Coprocessor Opcodes

Coprocessor Instructions (COP0..COP3)

Unknown if any tlb-opcodes (tlbr,tlbwi,tlbwr,tlbp) are implemented in the psx?

Caution - Load Delay

When reading from a coprocessor register, the next opcode cannot use the destination register as operand (much the same as the Load Delays that occur when reading from memory; see there for details).
Reportedly, the Load Delay applies for the next TWO opcodes after coprocessor reads, but, that seems to be nonsense (the PSX does finish both COP0 and COP2 reads after ONE opcode).

Caution - Store Delay

In some cases, a similar delay occurs when writing to a coprocessor register. COP0 is more or less free of store delays (eg. one can read from a cop0 register immediately after writing to it), the only known exception is the cop2 enable bit in cop0r12.bit30 (setting that cop0 bit acts delayed, and cop2 isn't actually enabled until after 2 clock cycles or so).
Writing to cop2 registers has a delay of 2..3 clock cycles. In most cases, that is probably (?) only 2 cycles, but special cases like writing to IRGB (which does additionally affect IR1,IR2,IR3) take 3 cycles until the result arrives in all registers).
Note that Store Delays are counted in numbers of clock cycles (not in numbers of opcodes). For 3 cycle delay, one must usually insert 3 cached opcodes (or one uncached opcode).

18.7 CPU Pseudo Opcodes

Pseudo instructions (native/spasm)

Pseudo instructions (nocash/a22i)

Below are pseudo instructions combined of two 32bit opcodes...

Below are pseudo instructions combined of two or more 32bit opcodes...

Possible more Pseudos...

Directives (nocash)

Directives (native)

Syntax for unknown assembler (for pad.s)

It uses "0x" for HEX values (but doesn't use "$" for registers).
It uses "#" instead of ";" for comments.
It uses ":" for labels (fortunately).
The assembler has at least one directive: ".byte" (equivalent to "db" on other assemblers).
I've no clue which assembler is used for that syntax... could that be the Psy-Q assembler?

18.8 COP0 - Register Summary

COP0 Register Summary

18.9 COP0 - Exception Handling

cop0r13 - CAUSE - (Read-only, except, Bit8-9 are R/W)

Describes the most recently recognised exception

cop0r12 - SR - System status register (R/W)

cop0r14 - EPC - Return Address from Trap (R)

This address is the instruction at which the exception took place, unless BD is set in CAUSE, then the instruction was at EPC+4.
Interrupts should always return to EPC+0, no matter of the BD flag. That way, if BD=1, the branch gets executed again, that's required because EPC stores only the current program counter, but not additionally the branch destination address.

Other exceptions may require to handle BD. In simple cases, when BD=0, the exception handler may return to EPC+0 (retry execution of the opcode), or to EPC+4 (skip the opcode that caused the exception). Note that jumps to faulty memory locations are executed without exception, but will trigger address errors and bus errors at the target location, ie. EPC (and BadVAddr, in case of address errors) point to the faulty address, not to the opcode that has jumped to that address).

Interrupts vs GTE Commands

If an interrupt occurs "on" a GTE command (cop2cmd), then the GTE command is executed, but nethertheless, the return address in EPC points to the GTE command. So, if the exeception handler would return to EPC as usually, then the GTE command would be executed twice. In best case, this would be a waste of clock cycles, in worst case it may lead to faulty result (if the results from the 1st execution are re-used as incoming parameters in the 2nd execution). To fix the problem, the exception handler must do:

Note: The above exception handling is working only in newer PSX BIOSes, but in very old PSX BIOSes, it is only incompletely implemented (see "BIOS Patches" chapter for common workarounds; or write your own exception handler without using the BIOS).
Of course, the above exeption handling won't work in branch delays (where BD gets set to indicate that EPC was modified) (best workaround is not to use GTE commands in branch delays).

cop0cmd=10h - RFE opcode - Prepare Return from Exception

The RFE opcode moves some bits in cop0r12 (SR): bit2-3 are copied to bit0-1, and bit4-5 are copied to bit2-3, all other bits (including bit4-5) are left unchanged.
The RFE opcode does NOT automatically jump to EPC. Instead, the exception handler must copy EPC into a register (usually R26 aka K0), and then jump to that address. Because of branch delays, that would look like so:

If you expect exceptions to be nested deeply, also push/pop SR. Note that there's no way to leave all registers intact (ie. above code destroys K0).

cop0r8 - BadVaddr - Bad Virtual Address (R)

Contains the address whose reference caused an exception. Set on any MMU type of exceptions, on references outside of kuseg (in User mode) and on any misaligned reference. BadVaddr is updated ONLY by Address errors (Excode 04h and 05h), all other exceptions (including bus errors) leave BadVaddr unchanged.

Exception Vectors (depending on BEV bit in SR register)

Note: Changing vectors at 800000xxh (kseg0) seems to be automatically reflected to the instruction cache without needing to flush cache (at least it worked SOMETIMES in my test proggy... but NOT always? ...anyways, it'd be highly recommended to flush cache when changing any opcodes), whilst changing mirrors at 000000xxh (kuseg) seems to require to flush cache.
The PSX uses only the BEV=0 vectors (aside from the reset vector, the PSX BIOS ROM doesn't contain any of the BEV=1 vectors).

Exception Priority

18.10 COP0 - Misc

cop0r15 - PRID - Processor ID (R)

For a Playstation with CXD8606CQ CPU, the PRID value is 00000002h.
Unknown if/which other Playstation CPU versions have other values...?

cop0r6 - JUMPDEST - Randomly memorized jump address (R)

The is a rather strange totally useless register. After certain exceptions, the CPU does memorize a jump destination address in the register. Once when it has memorized an address, the register becomes locked, and the memorized value won't change until it becomes unlocked by a new exception. Exceptions that do unlock the register are Reset and Interrupts (cause.bit10). Exceptions that do NOT unlock the register are syscall/ break opcodes, and software generated interrupts (eg. cause.bit8).
In the unlocked state, the CPU does more or less randomly memorize one of the next some jump destinations - eg. the destination from the second jump after reset, or from a jump that occured immediately before executing the IRQ handler, or from a jump inside of the IRQ handler, or even from a later jump that occurs shortly after returning from the IRQ handler.
The register seems to be read-only (although the Kernel initialization code writes 0 to it for whatever reason).

cop0r0..r2, cop0r4, cop0r10, cop0r32..r63 - N/A

Registers 0,1,2,4,10 control virtual memory on some MIPS processors (but there's none such in the PSX), and Registers 32..63 (aka "control registers") aren't used in any MIPS processors. Trying to read any of these registers causes a Reserved Instruction Exception (excode=0Ah).

cop0cmd=01h,02h,06h,08h - TLBR,TLBWI,TLBWR,TLBP

The PSX supports only one cop0cmd (cop0cmd=10h aka RFE). Trying to execute the TLBxx opcodes causes a Reserved Instruction Exception (excode=0Ah).

jf/jt cop0flg,dest - conditional cop0 jumps

mov [mem],cop0reg / mov cop0reg,[mem] - coprocessor cop0 load/store

Not supported by the CPU. Trying to execute these opcodes causes a Coprocessor Unusable Exception (excode=0Bh, ie. unlike above, not 0Ah).

cop0r16-r31 - Garbage

Trying to read these registers returns garbage (but does not trigger an exception). When reading one of the garbage registers shortly after reading a valid cop0 register, the garbage value is usually the same as that of the valid register. When doing the read later on, the return value is usually 00000020h, or when reading much later it returns 00000040h, or even 00000100h. No idea what is causing that effect...?
Note: The garbage registers can be accessed (without causing an exception) even in "User mode with cop0 disabled" (SR.Bit1=1 and SR.Bit28=0); accessing any other existing cop0 registers (or executing the rfe opcode) in that state is causing Coprocessor Unusable Exceptions (excode=0Bh).

18.11 COP0 - Debug Registers

The COP0 debug registers seem to be PSX specific, normal R30xx CPUs like IDT's R3041 and R3051 don't have anything similar.

cop0r7 - DCIC - Breakpoint control (R/W)

When a breakpoint address match occurs the PSX jumps to 80000040h (ie. unlike normal exceptions, not to 80000080h). The Excode value in the CAUSE register is set to 09h (same as BREAK opcode), and EPC contains the return address, as usually. One of the first things to be done in the exception handler is to disable breakpoints (eg. if the any-jump break is enabled, then it must be disabled BEFORE jumping from 80000040h to the actual exception handler).

cop0r7.bit12-13 - Jump Redirection Note

If one or both of these bits are nonzero, then the PSX seems to check for the following opcode sequence,

if it does sense that sequence, then it sets PC=[00000000h], but does not store any useful information in any cop0 registers, namely it does not store the return address in EPC, so it's impossible to determine which opcode has caused the exception. For the jump target address, there are 31 registers, so one could only guess which of them contains the target value; for "POP PC" code it'd be usually R31, but for "JMP [vector]" code it may be any register. So far the feature seems to be more or less unusable...?

cop0r5 - BDA - Breakpoint on Data Access Address (R/W)

cop0r9 - BDAM - Breakpoint on Data Access Mask (R/W)

Break condition is "((addr XOR BDA) AND BDAM)=0".

cop0r3 - BPC - Breakpoint on Execute Address (R/W)

cop0r11 - BPCM - Breakpoint on Execute Mask (R/W)

Break condition is "((PC XOR BPC) AND BPCM)=0".

Note (BREAK Opcode)

Additionally, the BREAK opcode can be used to create further breakpoints by patching the executable code. The BREAK opcode uses the same Excode value (09h) in CAUSE register. However, the BREAK opcode jumps to the normal exception handler at 80000080h (not 80000040h).

Note (LibCrypt)

The debug registers are mis-used by "Legacy of Kain: Soul Reaver" (and maybe also other games) for storing libcrypt copy-protection related values (ie. just as a "hidden" location for storing data, not for actual debugging purposes).
CDROM Protection - LibCrypt

Note (Cheat Devices/Expansion ROMs)

The Expansion ROM header supports only Pre-Boot and Post-Boot vectors, but no Mid-Boot vector. Cheat Devices are often using COP0 breaks for Mid-Boot Hooks, either with BPC=BFC06xxxh (break address in ROM, used in older cheat firmwares), or with BPC=80030000h (break address in RAM aka relocated GUI entrypoint, used in later cheat firmwares). Moreover, aside from the Mid-Boot Hook, the Xplorer cheat device is also supporting a special cheat code that uses the COP0 break feature.

Note (Datasheet)

Note: COP0 debug registers are described in LSI's "L64360" datasheet, chapter 14. And in their LR33300/LR33310 datasheet, chapter 4.

19. Kernel (BIOS)

19.1 BIOS Overview

BIOS CDROM Boot

The main purpose of the BIOS is to boot games from CDROM, unfortunately, before doing that, it displays the Sony intro. It's also doing some copy protection and region checks, and refuses to boot unlicensed games, or illegal copies, or games for other regions.

BIOS Bootmenu

The bootmenu shows up when starting the Playstation without CDROM inserted. The menu allows to play Audio CDs, and to erase or copy game positions on Memory Cards.

BIOS Functions

The BIOS contains a number of more or less useful, and probably more or less inefficient functions that can be used by software.
No idea if it's easy to take full control of the CPU, ie. to do all hardware access and interrupt handling by software, without using the BIOS at all?
Eventually the BIOS functions for accessing the CDROM drive are important, not sure how complicated/compatible it'd be to access the CDROM drive directly via I/O ports... among others, there might be different drives used in different versions of the Playstation, which aren't fully compatible with each other?

BIOS Memory

The BIOS occupies 512Kbyte ROM with 8bit address bus (so the BIOS ROM is rather slow, for faster execution, portions of it are relocated to the first 64K of RAM). For some very strange reason, the original PSX BIOS executes all ROM functions in uncached ROM, which is incredible slow (nocash BIOS uses cached ROM, which does work without problems).
The first 64Kbyte of the 2Mbyte Main RAM are reserved for the BIOS (containing exception handlers, jump tables, other data, and relocated code). That reserved region does unfortunately include the "valuable" first 32Kbytes (valuable because that memory could be accessed directly via [R0+immediate], without needing to use R1..R31 as base register).

19.2 BIOS Memory Map

BIOS ROM Map (512Kbytes)

BIOS ROM Header/Footer

BIOS RAM Map (1st 64Kbytes of RAM) (fixed addresses mainly in 1st 500h bytes)

User Memory (not used by Kernel)

Table of Tables (see BIOS Control Blocks for details)

Each table entry consists of two 32bit values; containing the base address, and total size (in bytes) of the corresponding control blocks.

File handles (fd=00h..0Fh) can be simply converted as fcb=[140h]+fd*2Ch.
Event handles (event=F10000xxh) as evcb=[120h]+(event AND FFFFh)*1Ch.

Garbage Area at Address 00000000h

The first some bytes of memory address 00000000h aren't actually used by the Kernel, except for storing some garbage at that locations. However, this garbage is actually important for bugged games like R-Types and Fade to Black (ie. games that do read from address 00000000h due to using uninitialized pointers).
Initially, the garbage area is containing a copy of the 16-byte exception handler at address 80h, but the first 4-bytes are typically smashed (set to 00000003h from some useless dummy writes in some useless CDROM delays). Ie. the 16-bytes should have these values:

For R-Types, the halfword at [0] must non-zero (else the game will do a DMA to address 0, and thereby destroy kernel memory). Fade to Black does several garbage reads from [0..9], a wrong byte value at [5] can cause the game to crash with an invalid memory access exception upon memory card access.

19.3 BIOS Function Summary

Parameters, Registers, Stack

Argument(s) are passed in R4,R5,R6,R7,[SP+10h],[SP+14h],etc.
Caution: When calling a sub-function with N parameters, the caller MUST always allocate N words on the stack, and, although the first four parameters are passed in registers rather than on stack, the sub-function is allowed to use/destroy these words at [SP+0..N*4-1].
BIOS Functions (and custom callback functions) are allowed to destroy registers R1-R15, R24-R25, R31 (RA), and HI/LO. Registers R16-R23, R29 (SP), and R30 (FP) must be left unchanged (if the function uses that registers, then it must push/pop them). R26 (K0) is reserved for exception handler and should be usually not used by other functions. R27 (K1) and R28 (GP) are left more or less unused by the BIOS, so one can more or less freely use them for whatever purpose.
The return value (if any) is stored in R2 register.

A-Functions (Call 00A0h with function number in R9 Register)

Below functions A(A0h..B4h) not supported on pre-retail DTL-H2000 devboard:

B-Functions (Call 00B0h with function number in R9 Register)

Below functions B(4Ah..5Dh) not supported on pre-retail DTL-H2000 devboard:

C-Functions (Call 00C0h with function number in R9 Register)

SYS-Functions (Syscall opcode with function number in R4 aka A0 Register)

The 20bit immediate in the "syscall imm" opcode is unused (should be zero).

BREAK-Functions (Break opcode with function number in opcode's immediate)

BRK opcodes may be used within devkits, however, the standard BIOS simply calls DeliverEvent(F0000010h,1000h) and SystemError_A_40h upon any BRK opcodes (as well as on any other unresolved exceptions).

Below breaks are used in DTL-H2000 BIOS:

The break functions have argument(s) in A1,A2,A3 (ie. unlike normal BIOS functions not in A0,A1,A2), and TWO return values (in R2, and R3). These functions require a commercial/homebrew devkit... consisting of a Data Cable (for accessing the PC's harddisk)... and an Expansion ROM (for handling the BREAK opcodes)... or so?

19.4 BIOS File Functions

A(00h) or B(32h) - open(filename, accessmode) - Opens a file for IO

Opens a file on the target device for io. Accessmode is set like this:

The PSX can have a maximum of 16 files open at any time, of which, 2 handles are always reserved for std_io, so only 14 handles are available for actual files. Some functions (cd, testdevice, erase, undelete, format, firstfile2, rename) are temporarily allocating 1 filehandle (rename tries to use 2 filehandles, but, it does accidently use only 1 handle, too). So, for example, erase would fail if more than 13 file handles are opened by the game.

A(01h) or B(33h) - lseek(fd, offset, seektype) - Move the file pointer

Moves the file pointer the number of bytes in A1, relative to the location specified by A2. Movement from the eof is incorrect. Also, movement beyond the end of the file is not checked.

A(02h) or B(34h) - read(fd, dst, length) - Read data from an open file

Reads the number of bytes from the specified open file. If length is not specified an error is returned. Read per $0080 bytes from memory card (bu:) and per $0800 from cdrom (cdrom:).

A(03h) or B(35h) - write(fd, src, length) - Write data to an open file

Writes the number of bytes to the specified open file. Write to the memory card per $0080 bytes. Writing to the cdrom returns 0.

A(04h) or B(36h) - close(fd) - Close an open file

Returns r2=fd (or r2=-1 if failed).

A(08h) or B(3Ah) - getc(fd) - read one byte from file

Internally redirects to "read(fd,tempbuf,1)". For some strange reason, the returned character is sign-expanded; so, a return value of FFFFFFFFh could mean either character FFh, or error.

A(09h) or B(3Bh) - putc(char,fd) - write one byte to file

Observe that "fd" is the 2nd paramter (not the 1st paramter as usually).

Internally redirects to "write(fd,tempbuf,1)".

B(40h) - cd(name) - Change the current directory on target device

Changes the current directory on the specified device, which should be "cdrom:" (memory cards don't support directories). The PSX supports only a current directory, but NOT a current device (ie. after cd, the directory name may be ommited from filenames, but the device name must be still included in all filenames).

Returns 1=okay, or 0=failed.
The function doesn't verify if the directory exists. Caution: For cdrom, the function does always load the path table from the disk (even if it was already stored in RAM, so cd is causing useless SLOW read/seek delays).

B(42h) - firstfile2(filename,direntry) - Find first file to match the name

Returns r2=direntry (or r2=0 if no matching files).
Searches for the first file to match the specified filename; the filename may contain "?" and "*" wildcards. "*" means to ignore ALL following characters; accordingly one cannot specify any further characters after the "*" (eg. "DATA*" would work, but "*.DAT" won't work). "?" is meant to ignore a single character cell. Note: The "?" wildcards (but not "*") can be used also in all other file functions; causing the function to use the first matching name (eg. erase "????" would erase the first matching file, not all matching files).
Start the name with the device you want to address. (ie. pcdrv:) Different drives can be accessed as normally by their drive names (a:, c:, huh?) if path is omitted after the

device, the current directory will be used.
A direntry structure looks like this:

BUG: If "?" matches the ending 00h byte of a name, then any further characters in the search expression are ignored (eg. "FILE?.DAT" would match to "FILE2.DAT", but accidently also to "FILE").
BUG: For CDROM, the BIOS includes some code that is intended to realize disk changes during firstfile2/nextfile operations, however, that code is so bugged that it does rather ensure that the BIOS does NOT realize new disks being inserted during firstfile2/nextfile.
BUG: firstfile2/nextfile is internally using a FCB. On the first call to firstfile2, the BIOS is searching a free FCB, and does apply that as "search fcb", but it doesn't mark that FCB as allocated, so other file functions may accidently use the same FCB. Moreover, the BIOS does memorize that "search fcb", and, even when starting a new search via another call to firstfile2, it keeps using that FCB for search (without checking if the FCB is still free). A possible workaround is not to have any files opened during firstfile2/nextfile operations.

B(43h) - nextfile(direntry) - Searches for the next file to match the name

Returns r2=direntry (or r2=0 if no more matching files).
Uses the settings of a previous firstfile2/nextfile command.

B(44h) - rename(old_filename, new_filename)

Returns 1=okay, or 0=failed.

B(45h) - erase(filename) - Delete a file on target device

Returns 1=okay, or 0=failed.

B(46h) - undelete(filename)

Returns 1=okay, or 0=failed.

B(41h) - format(devicename)

Erases all files on the device (ie. for formatting memory cards).
Returns 1=okay, or 0=failed.

B(54h) - _get_errno()

Indicates the reason of the most recent file function error (open, lseek, read, write, close, _get_error, ioctl, cd, testdevice, erase, undelete, format, rename). Use _get_errno() ONLY if an error has occured (the error code isn't reset to zero by functions that are passing okay). firstfile2/nextfile do NOT affect _get_errno(). See below list of File Error Numbers for more info.

B(55h) - _get_error(fd)

Basically same as B(54h), but allowing to specify a file handle for which error information is to be received; accordingly it doesn't work for functions that do use 'hidden' internal file handles (eg. erase, or unsuccessful open). Returns FCB[18h], or FFFFFFFFh if the handle is invalid/unused.

A(05h) or B(37h) - ioctl(fd,cmd,arg)

Used only for TTY.

A(07h) or B(39h) - isatty(fd)

Returns bit1 of the file's DCB flags. That bit is set only for Duart/TTY, and is cleared for Dummy/TTY, Memory Card, and CDROM.

B(59h) - testdevice(devicename)

Whatever. Checks the devicename, and if it's accepted, calls a device specific function. For the existing devices (cdrom,bu,tty) that specific function simply returns without doing anything. Maybe other devices (like printers or modems) would do something more interesting.

File Error Numbers for B(54h) and B(55h)

19.5 BIOS File Execute and Flush Cache

A(41h) - LoadTest(filename, headerbuf)

Loads the 800h-byte exe file header to an internal sector buffer, and does then copy bytes [10h..4Bh] of that header to headerbuf[00h..3Bh].

A(42h) - Load(filename, headerbuf)

Same as LoadTest (see there for details), but additionally loads the body of the executable (using the size and destination address in the file header), and does call FlushCache. The exe can be then started via Exec (this isn't done automatically by LoadTest). Unlike "LoadExec", the "LoadTest/Exec" combination allows to return the new exe file to return to the old exe file (instead of restarting the boot executable).
BUG: Uses the unstable FlushCache function (see there for details).

A(43h) - Exec(headerbuf, param1, param2)

Can be used to start a previously loaded executable. The function saves R16,R28,R30,SP,RA in the reserved region of headerbuf (rather than on stack), more or less slowly zerofills the memfill region specified in headerbuf, reads the stack base and offset values and sets SP and FP to base+offset (or leaves them unchanged if base=0), reads the GP value from headerbuf and sets GP to that value. Then calls the excecutables entrypoint, with param1 and param2 passed in r4,r5.
If the executable (should) return, then R16,R28,R30,SP,RA are restored from headerbuf, and the function returns with r2=1.

A(51h) - LoadExec(filename, stackbase, stackoffset)

This is a rather bizarre function. In short, it does load and execute the specified file, and thereafter, it (tries to) reload and restart to boot executable.
Part1: Takes a copy of the filename, with some adjustments: Everything up to the first

":" or 00h byte is copied as is (ie. the device name, if it does exist, or otherwise the whole path\filename.ext;ver), the remaining characters are copied and converted to uppercase (ie. the path\filename.ext;ver part, or none if the device name didn't exist), finally, checks if a ";" exists (ie. the version suffix), if there's none, then it appends ";1" as default version. CAUTION: The BIOS allocates ONLY 28 bytes on stack for the copy of the filename, that region is followed by 4 unused bytes, so the maximum length would be 32 bytes (31 characters plus EOL) (eg. "device:\pathname\filename.ext;1",00h).
Part2: Enables IRQs via ExitCriticalSection, memorizes the stack base/offset values from the previously loaded executable (which should have been the boot executable, unless LoadExec should have been used in nested fashion), does then use LoadTest to load the desired file, replaces the stack base/offset values in its headerbuf by the LoadExec parameter values, and does then execute it via Exec(headerbuf,1,0).
Part3: If the exefile returns, or if it couldn't be loaded, then the boot file is (unsuccessfully) attempted to be reloaded: Enables IRQs via ExitCriticalSection, loads the boot file via LoadTest, replaces the stack base/offset values in its headerbuf by the values memorized in Part2 (which \<should> be the boot executable's values from SYSTEM.CNF, unless the nesting stuff occurred), and does then execute the boot file via Exec(headerbuf,1,0).
Part4: If the boot file returns, or if it couldn't be loaded, then the function looks up in a "JMP $" endless loop (normally, returning from the boot exe causes SystemError("B", 38Ch), however, after using LoadExec, this functionality is replaced by the "JMP $" lockup.
BUG: Uses the unstable FlushCache function (see there for details).
BUG: Part3 accidently treats the first 4 characters of the exename as memory address (causing an invalid memory address exception on address 6F726463h, for "cdrom:filename.exe").

A(9Ch) - SetConf(num_EvCB, num_TCB, stacktop)

Changes the number of EvCBs and TCBs, and the stacktop. These values are usually initialized from the settings in the SYSTEM.CNF file, so using this function usually shouldn't ever be required.
The function deallocates all old ExCBs, EvCBs, TCBs (so all Exception handlers, Events, and Threads (except the current one) are lost, and all other memory that may have been allocated via alloc_kernel_memory(size) is deallocated, too. It does then allocate the new control blocks, and enqueue the default handlers. Despite of the changed stacktop, the current stack pointer is kept intact, and the function returns to the caller.

A(9Dh) - GetConf(num_EvCB_dst, num_TCB_dst, stacktop_dst)

Returns the number of EvCBs, TCBs, and the initial stacktop. There's no return value in the R2 register, instead, the three 32bit return values are stored at the specified "dst" addresses.

A(44h) - FlushCache()

Flushes the Code Cache, so opcodes are ensured to be loaded from RAM. This is required when loading program code via DMA (ie. from CDROM) (the cache controller apparently doesn't realize changes to RAM that are caused by DMA). The LoadTest and LoadExec functions are automatically calling FlushCache (so FlushCache is required only when loading program code via "read" or via "CdReadSector").
FlushCache may be also required when relocating or modifying program code by software (the cache controller doesn't seem to realize modifications to memory mirrors, eg. patching the exception handler at 80000080h seems to be work without FlushCache, but patching the same bytes at 00000080h doesn't).
Note: The PSX doesn't have a Data Cache (or actually, it has, but it's misused as Fast RAM, mapped to a fixed memory region, and which isn't accessable by DMA), so FlushCache isn't required for regions that contain data.
BUG: The FlushCache function contains a handful of opcodes that do use the k0 register without having IRQs disabled at that time, if an IRQ occurs during those opcodes, then the k0 value gets destroyed by the exception handler, causing FlushCache to get trapped in an endless loop.
One workaround would be to disable all IRQs before calling FlushCache, however, the BIOS does internally call the function without IRQs disabled, that applies to:

for load_file/load_exec, IRQ2 (cdrom) and IRQ3 (dma) need to be enabled, so the "disable all IRQs" workaround cannot be used for that functions, however, one can/ should disable as many IRQs as possible, ie. everything except IRQ2/IRQ3, and all DMA interrupts except DMA3 (cdrom).

Executable Memory Allocation

LoadTest and LoadExec are simply loading the file to the address specified in the exe file header. There's absolutely no verification whether that memory is (or isn't) allocated via malloc, or if it is used by the boot executable, or by the kernel, or if it does contain RAM at all.
When using the "malloc" function combined with loading exe files, it may be recommended not to pass all memory to InitHeap (ie. to keep a memory region being reserved for loading executables).

Note

For more info about EXE files and their headers, see
CDROM File Formats

19.6 BIOS CDROM Functions

General File Functions

CDROMs are basically accessed via normal file functions, with device name "cdrom:" (which is an abbreviation for "cdrom0:", anyways, the port number is ignored).
BIOS File Functions
BIOS File Execute and Flush Cache
Before starting the boot executable, the BIOS automatically calls _96_init(), so the game doesn't need to do any initializations before using CDROM file functions.

Absent CD-Audio Support

The Kernel doesn't include any functions for playing Audio tracks. Also, there's no BIOS function for setting the XA-ADPCM file/channel filter values. So CD Audio can be used only by directly programming the CDROM I/O ports.

Asynchronous CDROM Access

The normal File functions are always using synchroneous access for CDROM (ie. the functions do wait until all data is transferred) (unlike as for memory cards, accessmode.bit15 cannot be used to activate asynchronous cdrom access).
However, one can read files in asynchrouneous fashion via CdGetLbn, CdAsyncSeekL,

and CdAsyncReadSector. CDROM files are non-fragmented, so they can be read simply from incrementing sector numbers.

A(A4h) - CdGetLbn(filename)

Returns the first sector number used by the file, or -1 in case of error.
BUG: The function accidently returns -1 for the first file in the directory (the first file should be a dummy entry for the current or parent directory or so, so that bug isn't much of a problem), if the file is not found, then the function accidently returns garbage (rather than -1).

A(A5h) - CdReadSector(count,sector,buffer)

Reads \<count> sectors, starting at \<sector>, and writes data to \<buffer>. The read is done in mode=80h (double speed, 800h-bytes per sector). The function waits until all sectors are transferred, and does then return the number of sectors (ie. count), or -1 in case of error.

A(A6h) - CdGetStatus()

Retrieves the cdrom status via CdAsyncGetStatus(dst) (see there for details; especially for cautions on door-open flag). The function waits until the event indicates completion, and does then return the status byte (or -1 in case of error).

A(78h) - CdAsyncSeekL(src)

Issues Setloc and SeekL commands. The parameter (src) is a pointer to a 3-byte sector number (MM,SS,FF) (in BCD format).
The function returns 0=failed, or 1=okay. Completion is indicated by events (class=F0000003h, and spec=20h, or 8000h).

A(7Ch) - CdAsyncGetStatus(dst)

Issues a GetStat command. The parameter (dst) is a pointer to a 1-byte location that receives the status response byte.
The function returns 0=failed, or 1=okay. Completion is indicated by events (class=F0000003h, and spec=20h, or 8000h).
Caution: The command acknowledges the door-open flag, but doesn't automatically reload the path table (which is required if a new disk is inserted); if the door-open flag was set, one should call a function that does forcefully load the path table (like cd).

A(7Eh) - CdAsyncReadSector(count,dst,mode)

Issues SetMode and ReadN (when mode.bit8=0), or ReadS (when mode.bit8=1) commands. count is the number of sectors to be read, dst is the destination address in RAM, mode.bit0-7 are passed as parameter to the SetMode command, mode.bit8 is the ReadN/ReadS flag (as described above). The sector size (for DMA) depends on the mode value: 918h-bytes (bit4=1, bit5=X), 924h-bytes (bit4=0, bit5=1), or 800h-bytes (bit4,5=0).
Before CdAsyncReadSector, the sector number should be set via CdAsyncSeekL(src).
The function returns 0=failed, or 1=okay. Completion is indicated by events (class=F0000003h, and spec=20h, 80h, or 8000h).

A(81h) - CdAsyncSetMode(mode)

Similar to CdAsyncReadSector (see there for details), but issues only the SetMode command, without any following ReadN/ReadS command.

A(94h) - CdromGetInt5errCode(dst1,dst2)

Returns the first two response bytes from the most recent INT5 error: [dst1]=status, [dst2]=errorcode. The BIOS doesn't reset these values in case of successful completion, so the values are quite useless.

A(54h) or A(71h) - _96_init()

A(56h) or A(72h) - _96_remove() ;does NOT work due to SysDeqIntRP bug

A(90h) - CdromIoIrqFunc1()

A(91h) - CdromDmaIrqFunc1()

A(92h) - CdromIoIrqFunc2()

A(93h) - CdromDmaIrqFunc2()

A(95h) - CdInitSubFunc() ;subfunction for _96_init()

A(9Eh) - SetCdromIrqAutoAbort(type,flag)

A(A2h) - EnqueueCdIntr() ;with prio=0 (fixed)

A(A3h) - DequeueCdIntr() ;does NOT work due to SysDeqIntRP bug

Internally used CDROM functions for initialization and IRQ handling.

19.7 BIOS Memory Card Functions

General File Functions

Memory Cards aka Backup Units (bu) are basically accessed via normal file functions, with device names "bu00:" (Slot 1) and "bu10:" (Slot 2),
BIOS File Functions
Before using the file functions for memory cards, first call InitCARD2(pad_enable), then StartCARD2(), and then _bu_init().

File Header, Filesize, and Sector Alignment

The first 100h..200h bytes (2..4 sectors) of the file must contain the title and icon bitmap(s). For details, see:
Memory Card Data Format
The filesize must be a multiple of 2000h bytes (one block), the maximum size would be 1E000h bytes (when using all 15 blocks on the memory card). The filesize must be

specified when creating the file (ie. accessmode bit9=1, and bit16-31=number of blocks). Once when the file is created, the BIOS does NOT allow to change the filesize (unless by deleting and re-creating the file).
When reading/writing files, the amount of data must be a multiple of 80h bytes (one sector), and the file position must be aligned to a 80h-byte boundary, too. There's no restriction on fragmented files (ie. one may cross 2000h-byte block boundaries within the file).

Poor Memcard Performance

PSX memory card accesses are typically super-slow. That, not so much because the hardware would be slow, but rather because of improper inefficent code at the BIOS side. The original BIOS tries to synchronize memory card accesses with joypad accesses simply by accessing only one sector per frame (although it could access circa two sectors). To the worst, the BIOS accesses Slot 1 only on each second frame, and Slot 2 only each other frame (although in 99% of all cases only one slot is accessed at once, so the access time drops to 0.5 sectors per frame).
Moreover, the memory card id, directory, and broken sector list do occupy 26 sectors (although the whole information would fit into 4 or 5 sectors) (a workaround would be to read only the first some bytes, and to skip the additional unused bytes - though that'd also mean to skip the checksums which are unfortunately stored at the end of the sector).
And, anytime when opening a file (in synchronous mode), the BIOS does additionally read sector 0 (which is totally useless, and gets especially slow when opening a bunch of files; eg. when extracting the title/icon from all available files on the card).

Asynchronous Access

The BIOS supports synchronous and asynchronous memory card access. Synchronous means that the BIOS function doesn't return until the access has completed (which means, due to the poor performance, that the function spends about 75% of the time on inactivity) (except in nocash PSX bios, which has better performance), whilst asynchronous access means that the BIOS function returns immediately after invoking the access (which does then continue on interrupt level, and does return an event when finished).
The file "read" and "write" functions act asynchronous when accessmode bit15 is set when opening the file. Additionally, the A(ACh) _card_load(port) function can be used to tell the BIOS to load the directory entries and broken sector list to its internal RAM buffers (eg. during the games title screen, so the BIOS doesn't need to load that data

once when the game enters its memory card menu). All other functions like erase or format always act synchronous. The open/findfirst/findnext functions do normally complete immediately without accessing the card at all (unless the directory wasn't yet read; in that case the directory is loading in synchronous fashion).
Unfortunately, the asynchronous response doesn't rely on a single callback event, but rather on a bunch of different events which must be all allocated and tested by the game (and of which, one event is delivered on completion) (which one depends on whether function completed okay, or if an error occurred).

Multitap Support (and Multitap Problems)

The BIOS does have some partial support for accessing more than two memory cards (via Multitap adaptors). Device/port names "bu01:", "bu02:", "bu03:" allow to access extra memory carts in slot1 (and "bu11:", "bu12:", "bu13:" in slot2). Namely, those names will send values 82h, 83h, 84h to the memory card slot (instead of the normal 81h value).
However, the BIOS directory_buffer and broken_sector_list do support only two memory cards (one in slot1 and one in slot2). So, trying to access more memory cards may cause great data corruption (though there might be a way to get the BIOS to reload those buffers before accessing a different memory card).
Aside from that problem, the BIOS functions are very-very-very slow even when accessing only two memory cards. Trying to use the BIOS to access up to eight memory cards would be very-extremly-very slow, which would be more annoying than useful.

B(4Ah) - InitCARD2(pad_enable) ;uses/destroys k0/k1 !!!

B(4Bh) - StartCARD2()

B(4Ch) - StopCARD2()

A(55h) or A(70h) - _bu_init()

A(ABh) - _card_info(port)

B(4Dh) - _card_info_subfunc(port) ;subfunction for "_card_info"

Can be used to check if the most recent call to _card_write has completed okay. Issues an incomplete dummy read command (similar to B(4Fh) - _card_read). The read command is aborted once when receiving the status byte from the memory card (the actual data transfer is skipped).

A(AFh) - card_write_test(port) ;not supported by old CEX-1000 version

Resets the card changed flag. For some strange reason, this flag isn't automatically reset after reading the flag, instead, the flag is reset upon sector writes. To do that, this function issues a dummy write to sector 3Fh.

B(50h) - _new_card()

Normally any memory card read/write functions fail if the BIOS senses the card change flag to be set. Calling this function tells the BIOS to ignore the card change flag on the next read/write operation (the function is internally used when loading the "MC" ID from sector 0, and when calling the card_write_test function to acknowledge the card change flag).

B(4Eh) - _card_write(port,sector,src)

B(4Fh) - _card_read(port,sector,dst)

Invokes asynchronous reading/writing of a single sector. The function returns 1=okay, or 0=failed (on invalid sector numbers). The actual I/O is done on IRQ level, completion of the I/O command transmission can be checked, among others, via get/wait_card_status(slot) functions (with slot=port/10h).
In case of the write function, completion of the \<transmission> does NOT mean that the actual \<writing> has completed, instead, write errors are indicated upon completion of the \<next sector> read/write transmission (or, if there are no further sectors to be accessed; one can use _card_info to verify completion of the last written sector).
The sector number should be in range of 0..3FFh, for some strange reason, probably a BUG, the function also accepts sector 400h. The specified sector number is directly accessed (it is NOT parsed through the broken sector replacement list).

B(5Ch) - _card_status(slot)

B(5Dh) - _card_wait(slot)

Returns the status of the most recent I/O command, possible values are:

_card_status returns immediately, _card_wait waits until a non-busy state occurs.

A(A7h) - bufs_cb_0()

A(A8h) - bufs_cb_1()

A(A9h) - bufs_cb_2()

A(AAh) - bufs_cb_3()

A(AEh) - bufs_cb_4()

These five callback functions are internally used by the BIOS, notifying other BIOS functions about (un-)successful completion of memory card I/O commands.

B(58h) - _card_chan()

This is a subfunction for the five bufs_cb__xxx functions (indicating whether the callback occured for a slot1 or slot2 access).

A(ACh) - _card_load(port)

Invokes asynchronous reading of the memory card directory. The function isn't too useful because the BIOS tends to read the directory automatically in various places in synchronous mode, so there isn't too much chance to replace the automatic synchronous reading by asynchronous reading.

A(ADh) - _card_auto(flag)

Can be used to enable/disable auto format (0=off, 1=on). The _bu_init function initializes auto format as disabled. If auto format is enabled, then the BIOS does automatically format memory cards if it has failed to read the "MC" ID bytes on sector 0. Although potentially useful, activating this feature is rather destructive (for example, read errors on sector 0 might occur accidently due to improperly inserted cards with dirty contacts, so it'd be better to prompt the user whether or not to format the card, rather than doing that automatically).

C(1Ah) - set_card_find_mode(mode)

C(1Dh) - get_card_find_mode()

Allows to get/set the card find mode (0=normal, 1=find deleted files), the mode setting affects only the firstfile2/nextfile functions. All other file functions are used fixed mode settings (always mode=0 for open, rename, erase, and mode=1 for undelete).

19.8 BIOS Interrupt/Exception Handling

The Playstation's Kernel uses an uncredible inefficient and unstable exception handler; which may have been believed to be very powerful and flexible.

Inefficiency

For a maximum of slowness, it does always save and restore all CPU registers (including such that aren't used in the exception handler). It does then go through a list of installed interrupt handlers - and executes ALL of them. For example, a Timer0 IRQ is first passed to the Cdrom and Vblank handlers (which must reject it, no thanks), before it does eventually reach the Timer0 handler.

Unstable IRQ Handling

A fundamental mistake in the exception handler is that it doesn't memorize the incoming IRQ flags. So the various interrupt handlers must check Port 1F801070h one after each other. That means, if a high priority handler has rejected IRQ processing (because the desired IRQ flag wasn't set at that time), then a lower priority handler may process it (assuming that the IRQ flag got set in the meantime), and, in worst case it may even acknowledge it (so the high priority handler does never receive it).

To avoid such problems, there should be only ONE handler installed for each IRQ source. However, that isn't always possible because the Kernel automatically installs some predefined handlers. Most noteworthy, the totally bugged DefaultInterruptHandlers is always installed (and cannot be removed), so it does randomly trigger Events. Fortunately, it does not acknowledge the IRQs (unless SetIrqAutoAck was used to enable that fatal behaviour).

B(18h) - ResetEntryInt()

Applies the default "Exit" structure (which consists of a pointer to ReturnFromException, and the Kernel's exception stacktop (minus 4, for whatever reason), and zeroes for the R16..R23,R28,R30 registers. Returns the address of that structure.
See HookEntryInt for details.

B(19h) - HookEntryInt(addr)

addr points to a structure (with same format as for the setjmp function):

The hook function is executed only if the ExceptionHandler has been fully executed (after processing an IRQ, many interrupt handlers are calling ReturnFromException to abort further exception handling, and thus do skip the hook function). Once when the hook function has finished, it should execute ReturnFromException. The hook function is called with r2=1 (that is important if the hook address was recorded with setjmp, where it "returns" to the setjmp caller, with r2 as "return value").

Priority Chains

The Kernel's exception handler has four priority chains, each may contain one or more Interrupt or Exception handlers. The default handlers are:

The exception handler calls all handlers, starting with the first element in the priority 0 chain (ie. usually CdromDmaIrq). The separate handlers must check if they want to process the IRQ (eg. CdromDmaIrq would process only CDROM DMA IRQs, but not joypad IRQs or so). If it has processed and acknowledged the IRQ, then the handler may execute ReturnFromException, which causes the handlers of lower priority to be skipped (if there are still other unacknowledge IRQs pending, then the hardware will re-enter the exception handler as soon as the RFE opcode in ReturnFromException does re-enable interrupts).

C(02h) - SysEnqIntRP(priority,struc) ;bugged, use with care

Inserts a new element in the specified priority chain. The new element is inserted at the begin of the chain, so (within that priority chain) the new element has highest priority.

BUG: SysDeqIntRP can remove only the first element in the chain (see there for details), so adding new chain elements may cause OTHER functions to be unable to remove their chain elements. The BIOS seems to be occassionally adding/removing the "CardSpecificIrq" and "PadCardIrq" (with priority 1 and 2), so using that priorities may cause the BIOS to be unable to remove that IRQ handlers. Using priority 0 and 3 should work (as long as the software takes care to remove only the newest elements) (but there should be no conflicts with the BIOS which does never remove priority 0 and 3 elements) (leaving apart that DequeueCdIntr and _96_remove try to remove priority 0 elements, but that functions won't work anyways; due to the same bug).

C(03h) - SysDeqIntRP(priority,struc) ;bugged, use with care

Removes the specified element from the specified priority chain.
BUG: The function tries to search the whole chain (and to remove the element if it finds it). However, it does only check the first element properly, and, thereafter it reads a garbage value from an uninitialized stack location, and acts more or less unpredictable. So, it can remove only the first element in the chain, ie. it should be called only if you are SURE that there's no newer element in the chain, and only if you are SURE that the element IS in the chain.

SYS(01h) - EnterCriticalSection() ;syscall with r4=01h

Disables interrupts by clearing SR (cop0r12) Bit 2 and 10 (of which, Bit2 gets copied to Bit0 once when returning from the syscall exception). Returns 1 if both bits were set, returns 0 if one or both of the bits were already zero.

SYS(02h) - ExitCriticalSection() ;syscall with r4=02h

Enables interrupts by set SR (cop0r12) Bit 2 and 10 (of which, Bit2 gets copied to Bit0 once when returning from the syscall exception). There's no return value (all registers except SR and K0 are unchanged).

C(0Dh) - SetIrqAutoAck(irq,flag)

Specifies if the DefaultInterruptHandler shall automatically acknowledge IRQs. The "irq" paramter is the number of the interrupt, ie. 00h..0Ah = IRQ0..IRQ10. The "flag" value should be 0 to disable AutoAck, or non-zero to enable AutoAck. By default, AutoAck is disabled for all IRQs.
Mind that the DefaultInterruptHandler is totally bugged. Especially the AutoAck feature doesn't work very well: It may cause higher priority handlers to miss their IRQ, and it may even cause the DefaultInterruptHandler to miss its own IRQs.

C(06h) - ExceptionHandler()

The C(06h) vector points to the exception handler, ie. to the function that is invoked from the 4 opcodes at address 80000080h, that opcodes jump directly to the exception handler, so patching the C(06h) vector has no effect.
Reading the C(06h) entry can be used to let a custom 80000080h handler pass control back to the default handler (that, by a "direct" jump, not by the usual "MOV R9,06h / CALL 0C0h" method, which would destroy main programs R9 register).
Also, reading C(06h) may be useful for patching the exception handler (which contains a bunch of NOP opcodes, which seem to be intended to insert additional opcodes, such like debugger exception handling) (Note: some of that NOPs are reserved for Memory Card IRQ handling).
BUG: Early BIOS versions did try to examine a copy of cop0r13 in r2 register, but did forgot cop0r13 to r2 (so they examined garbage), this was fixed in newer BIOS versions, additionally, most commercial games still include patches for compatibility with the old BIOS.

B(17h) - ReturnFromException()

Restores the CPU registers (R1-R31,HI,LO,SR,PC) (except R26/K0) from the current TCB. This function is usually executed automatically at the end of the ExceptionHandler, however, functions in the exception chain may call ReturnFromException to return immediately, without processing chain elements of lower priority.

C(00h) - EnqueueTimerAndVblankIrqs(priority) ;used with prio=1

C(01h) - EnqueueSyscallHandler(priority) ;used with prio=0

C(0Ch) - InitDefInt(priority) ;used with prio=3

Internally used to add some default IRQ and Exception handlers.

No Nested Exceptions

The Kernel doesn't support nested exceptions, that would require a decreasing exception stack, however, the kernel saves the incoming CPU registers in the current TCB, and an exception stack with fixed start address for internal push/pop during exception handling. So, nesting would overwrite these values. Do not enable IRQs, and don't trap other exceptions (like break or syscall opcodes, or memory or overlow errors) during exception handling.
Note: The execption stack has a fixed size of 1000h bytes (and is located somewhere in the first 64Kbytes of memory).

19.9 BIOS Event Functions

B(08h) - OpenEvent(class, spec, mode, func)

Adds an event structure to the event table.

Opens an event, should be called within a critical section. The return value is used to identify the event to the other event functions. A list of event classes, specs and modes is at the end of this section. Initially, the event is disabled.

Note: The desired max number of events can be specified in the SYSTEM.CNF boot file (the default is "EVENT = 10" (which is a HEX number, ie. 16 decimal; of which 5 events are internally used by the BIOS for CDROM functions, so, of the 16 events, only 11 events are available to the game). A bigger amount of events will slowdown the DeliverEvent function (which always scans all EvCBs, even if all events are disabled).

B(09h) - CloseEvent(event) - releases event from the event table

Always returns 1 (even if the event handle is unused or invalid).

B(0Ch) - EnableEvent(event) - Turns on event handling for specified event

Always returns 1 (even if the event handle is unused or invalid).

B(0Dh) - DisableEvent(event) - Turns off event handling for specified event

Always returns 1 (even if the event handle is unused or invalid).

B(0Ah) - WaitEvent(event)

Returns 0 if the event is disabled. Otherwise hangs in a loop until the event becomes ready, and returns 1 once when it is ready (and automatically switches the event back to busy status). Callback events (mode=1000h) do never set the ready flag (and thus WaitEvent would hang forever).
The main program simply hangs during the wait, so when using multiple threads, it may be more recommended to create an own waitloop that checks TestEvent, and to call ChangeTh when the event is busy.
BUG: The return value is unstable (sometimes accidently returns 0=disabled if the event status changes from not-ready to ready shortly after the function call).

B(0Bh) - TestEvent(event)

Returns 0 if the event is busy or disabled. Otherwise, when it is ready, returns 1 (and automatically switches the event back to busy status). Callback events (mode=1000h) do never set the ready flag.

B(07h) - DeliverEvent(class, spec)

This function is usually called by the kernel, it triggers all events that are enabled/busy, and that have the specified class and spec values. Depending on the mode, either the callback function is called (mode=1000h), or the event is marked as enabled/ready (mode=2000h).

B(20h) - UnDeliverEvent(class, spec)

This function is usually called by the kernel, undelivers all events that are enabled/ready, and that have mode=2000h, and that have the specified class and spec values. Undeliver means that the events are marked as enabled/busy.

C(04h) - get_free_EvCB_slot()

A subfunction for OpenEvent.

Event Classes

File Events:

Hardware Events:

Event Events:

Root Counter Events (Timers and Vblank):

User Events:

BIOS Events (including such that have nothing to do with BIOS):

Thread Events:

Event Specs

Event modes

19.10 BIOS Event Summary

Below is a list of all events (class,spec values) that are delivered and/or undelivered by the BIOS in one way or another. The BIOS does internally open five events for cdrom (class=F0000003h with spec=10h,20h,40h,80h,8000h). The various other class/spec's are only delivered by the BIOS (but not received by the BIOS) (ie. a game may open/enable memory card events to receive notifications from the BIOS).

CDROM Events

Memory Card - Higher Level File/Device Events

Memory Card - Lower Level Hardware I/O Events

Timer/Vblank Events

Default IRQ Handler Events (very unstable, don't use)

Unresolved Exception Events

19.11 BIOS Thread Functions

B(0Eh) - OpenTh(reg_PC,reg_SP_FP,reg_GP)

Searches a free TCB, marks it as used, and stores the inital program counter (PC), global pointer (GP aka R28), stack pointer (SP aka R29), and frame pointer (FP aka R30) (using the same value for SP and FP). All other registers are left uninitialized (eg. may contain values from an older closed thread, that includes the SR register, see note). The return value is the new thread handle (in range FF000000h..FF000003h, assuming that 4 TCBs are allocated) or FFFFFFFFh if there's no free TCB. The function returns to the old current thread, use "ChangeTh" to switch to the new thread.
Note: The desired max number of TCBs can be specified in the SYSTEM.CNF boot file (the default is "TCB = 4", one initially used for the boot executable, plus 3 free threads).

BUG - Unitialized SR Register

OpenTh does NOT initialize the SR register (cop0r12) of the new thread. Upon powerup, the bootcode zerofills the TCB memory (so, the SR of new threads will be initially zero; ie. Kernel Mode, IRQ's disabled, and COP2 disabled). However, when closing/reopening threads, the SR register will have the value of the old closed thread (so it may get started with IRQs enabled, and, in worst case, if the old thread should have switched to

User Mode, even without access to KSEG0, KSEG1 memory).
Or, ACTUALLY, the memory is NOT zerofilled on powerup... so SR is total random?

B(0Fh) - CloseTh(handle)

Marks the TCB for the specified thread as unused. The function can be used for any threads, including for the current thread.
Closing the current thread doesn't terminate the current thread, so it may cause problems once when opening a new thread, however, it should be stable to execute the sequence "DisableInterrupts, CloseCurrentThread, ChangeOtherThread".
The return value is always 1 (even if the handle was already closed).

B(10h) - ChangeTh(handle)

Pauses the current thread, and activates the selected new thread (or crashes if the specified handle was unused or invalid).
The return value is always 1 (stored in the R2 entry of the TCB of the old thread, so the return value will be received once when changing back to the old thread).
Note: The BIOS doesn't automatically switch from one thread to another. So, all other threads remain paused until the current thread uses ChangeTh to pass control to another thread.
Each thread is having it's own CPU registers (R1..R31,HI,LO,SR,PC), the registers are stored in the TCB of the old thread, and restored when switching back to that thread.
Mind that other registers (I/O Ports or GTE registers aren't stored automatically, so, when needed, they need to be pushed/popped by software before/after ChangeTh).

C(05h) - get_free_TCB_slot()

Subfunction for OpenTh, returns the number of the first free TCB (usually in range 0..3) or FFFFFFFFh if there's no free TCB.

SYS(03h) ChangeThreadSubFunction(addr) ;syscall with r4=03h, r5=addr

Subfunction for ChangeTh, R5 contains the address of the new TCB, just like all exceptions, the syscall exception is saving the CPU registers in the current TCB, but does then apply the new TCB as current TCB, and so, it does then enter the new thread when returning from the exception.

19.12 BIOS Timer Functions

Timers (aka Root Counters)

The three hardware timers aren't internally used by any BIOS functions, so they can be freely used by the game, either via below functions, or via direct I/O access.

Vblank

Some of the below functions are allowing to use Vblank IRQs as a fourth "timer". However, Vblank IRQs are internally used by the BIOS for handling joypad and memory card accesses. One could theoretically use two separate Vblank IRQ handlers, one for joypad, and one as "timer", but the BIOS is much too unstable for such "shared" IRQ handling (it may occassionally miss one of the two handlers).
So, although Vblank IRQs are most important for games, the PSX BIOS doesn't actually allow to use them for purposes other than joypad access. A possible workaround is to examine the status byte in one of the joypad buffers (ie. the InitPAD2(buf1,22h,buf2,22h) buffers). Eg. a wait_for_vblank function could look like so: set buf1[0]=55h, then wait until buf1[0]=00h or buf1[0]=FFh.

B(02h) - init_timer(t,reload,flags)

When t=0..2, resets the old timer mode by setting [1F801104h+t*16]=0000h, applies the reload value by [1F801108h+t*16]=reload, computes the new mode:

and applies it by setting [1F801104h+t*16]=mode, and returns 1. Does nothing and returns zero for t>2.

B(03h) - get_timer(t)

Reads the current timer value: Returns halfword[1F801100h+t*16] for t=0..2. Does nothing and returns zero for t>2.

B(04h) - enable_timer_irq(t)

B(05h) - disable_timer_irq(t)

Enables/disables timer or vblank interrupt enable bits in [1F801074h], bit4,5,6 for t=0,1,2, or bit0 for t=3, or random/garbage bits for t>3. The enable function returns 1 for t=0..2, and 0 for t=3. The disable function returns always 1.

B(06h) - restart_timer(t)

Sets the current timer value to zero: Sets [1F801100h+t*16]=0000h and returns 1 for t=0..2. Does nothing and returns zero for t>2.

C(0Ah) - ChangeClearRCnt(t,flag) ;root counter (aka timer)

Selects what the kernel's timer/vblank IRQ handlers shall do after they have processed an IRQ (t=0..2: timer 0..2, or t=3: vblank) (flag=0: do nothing; or flag=1: automatically acknowledge the IRQ and immediately return from exception). The function returns the old (previous) flag value.

19.13 BIOS Joypad Functions

Pad Input

Joypads should be initialized via InitPAD2(buf1,22h,buf2,22h), and StartPAD2(). The main program can read the pad data from the buf1/buf2 addresses (including Status, ID1, button states, and any kind of analogue inputs). For more info on ID1, Buttons and analogue inputs, see
Controllers and Memory Cards
Note: The BIOS doesn't include any functions for sending custom data to the pads (such like for controlling rumble motors).

B(12h) - InitPAD2(buf1, siz1, buf2, siz2)

Memorizes the desired buf1/buf2 addresses, zerofills the buffers by using the siz1/siz2 buffer size values (which should be 22h bytes each). And does some initialization on the PadCardIrq element (but doesn't enqueue it, that must be done by a following call to StartPAD2), and does set the "pad_enable_flag", that flag can be also set/cleared via

InitCARD2(pad_enable), where it selects if the Pads are kept handled together with Memory Cards. buf1/buf2 are having the following format:

Note: InitPAD2 does initially zerofill the buffers, so, until the first IRQ is processed, the initial status is 00h=okay, with buttons=0000h (all buttons pressed), to fix that situation, change the two status bytes to FFh after calling InitPAD2 (or alternately, reject ID1=00h).
Once when the PadCardIrq is enqueued via StartPAD2, and while "pad_enable_flag" is set, the data for (both) Pad1 and Pad2 is read on Vblank interrupts, and stored in the buffers, the IRQ handler stores up to 22h bytes in the buffer (regardless of the siz1/siz2 values) (eg. a Multitap adaptor uses all 22h bytes).

B(13h) - StartPAD2()

Should be used after InitPAD2. Enqueues the PadCardIrq handler, and does additionally initialize some flags.

B(14h) - StopPAD2()

Dequeues the PadCardIrq handler. Note that this handler is also used for memory cards, so it'll "stop" cards, too.

B(15h) - PAD_init2(type, button_dest, unused, unused)

This is an extremely bizarre and restrictive function - don't use! The function fails unless type is 20000000h or 20000001h (the type value has no other function). The function uses "buf1/buf2" addresses that are located somewhere "hidden" within the BIOS variables region, the only way to read from that internal buffers is to use the ugly "PAD_dr()" function. For some strange reason it FFh-fills buf1/buf2, and does then call InitPAD2(buf1,22h,buf2,22) (which does immediately 00h-fill the previously FFh-filled buffers), and does then call StartPAD2().
Finally, it does memorize the "button_dest" address (see PAD_dr() for details on that value). The two unused parameters have no function, however, they are internally written back to the stack locations reserved for parameter 2 and 3, ie. at [SP+08h] and [SP+0Ch] on the caller's stack, so the function MUST be called with all four parameters allocated on stack. Return value is 2 (or 0 if type was disliked).

B(16h) - PAD_dr()

This is a very ugly function, using the internal "buf1/buf2" values from "PAD_init2" and the "button_dest" value that was passed to that function.
If "button_dest" is non-zero, then this function is automatically called by the PadCardIrq handler, and stores it's return value at [button_dest] (where it may be read by the main program). If "button_dest" is zero, then it isn't called automatically, and it \<can> be called manually (with return value in R2), however, it does additionally write the return value to [button_dest], ie. to [00000000h] in this case, destroying that memory location.
The return value itself is useless garbage: The lower 16bit contain the pad1 buttons, the upper 16bit the pad2 buttons, of which, both values have reversed byte-order (ie. the first button byte in upper 8bit). The function works only with controller IDs 41h (digital joypad) and 23h (nonstandard device). For ID=23h, the halfword is ORed with 07C7h, and bit6,7 are then cleared if the analogue inputs are greater than 10h. For ID=41h the data is left intact. Any other ID values, or disconnected joypads, cause the halfword to be set to FFFFh (same as when no buttons are pressed), that includes newer analogue pads (unless they are switched to "digital" mode).

19.14 BIOS GPU Functions

A(48h) - SendGP1Command(gp1cmd)

Writes [1F801814h]=gp1cmd. There's no return value (r2 is left unchanged).

A(49h) - GPU_cw(gp0cmd) ;send GP0 command word

Calls gpu_sync(), and does then write [1F801810h]=gp0cmd. Returns the return value from the gpu_sync() call.

A(4Ah) - GPU_cwp(src,num) ;send GP0 command word and parameter words

Calls gpu_sync(), and does then copy "num" words from [src and up] to [1F801810h], src should usually point to a command word, followed by num-1 parameter words. Transfer is done by software (without DMA). Always returns 0.

A(4Bh) - send_gpu_linked_list(src)

Transfer an OT via DMA. Calls gpu_sync(), and does then write [1F801814h]=4000002h, [1F8010F4h]=0, [1F8010F0h]=[1F8010F0h] OR 800h, [1F8010A0h]=src, [1F8010A4h]=0, [1F8010A8h]=1000401h. The function does additionally output a bunch of TTY status messages via printf. The function doesn't wait until the DMA is completed. There's no return value.

A(4Ch) - gpu_abort_dma()

Writes [1F8010A8h]=401h, [1F801814h]=4000000h, [1F801814h]=2000000h, [1F801814h]=1000000h. Ie. stops GPU DMA, and issues GP1(4), GP1(2), GP1(1). Returns 1F801814h, ie. the I/O address.

A(4Dh) - GetGPUStatus()

Reads [1F801814h] and returns that value.

A(46h) - GPU_dw(Xdst,Ydst,Xsiz,Ysiz,src)

Waits until GPUSTAT.Bit26 is set (unlike gpu_sync, which waits for Bit28), and does then [1F801810h]=A0000000h, [1F801810h]=YdstXdst, [1F801810h]=YsizXsiz, and finally transfers "N" words from [src and up] to [1F801810h], where "N" is "Xsiz*Ysiz/2". The data is transferred by software (without DMA) (by code executed in the uncached BIOS region with high waitstates, so the data transfer is very SLOW).
Caution: If "Xsiz*Ysiz" is odd, then the last halfword is NOT transferred, so the GPU stays waiting for the last data value.
Returns [SP+04h]=Ydst, [SP+08h]=Xsiz, [SP+0Ch]=Ysiz, [SP+10h]=src+N*4, and R2=src=N*4.

A(47h) - gpu_send_dma(Xdst,Ydst,Xsiz,Ysiz,src)

Calls gpu_sync(), writes [1F801810h]=A0000000h, [1F801814h]=4000002h, [1F8010F0h]=[1F8010F0h] OR 800h, [1F8010A0h]=src, [1F8010A4h]=N*10000h+10h (where N="Xsiz*Ysiz/32"), [1F8010A8h]=1000201h.
Caution: If "Xsiz*Ysiz" is not a multiple of 32, then the last halfword(s) are NOT transferred, so the GPU stays waiting for that values.
Returns R2=1F801810h, and [SP+04h]=Ydst, [SP+08h]=Xsiz, [SP+0Ch]=Ysiz.

A(4Eh) - gpu_sync()

If DMA is off (when GPUSTAT.Bit29-30 are zero): Waits until GPUSTAT.Bit28=1 (or until timeout).
If DMA is on: Waits until D2_CHCR.Bit24=0 (or until timeout), and does then wait until GPUSTAT.Bit28=1 (without timeout, ie. may hang forever), and does then turn off DMA via GP1(04h).
Returns 0 (or -1 in case of timeout, however, the timeout values are very big, so it may take a LOT of seconds before it returns).

19.15 BIOS Memory Allocation

A(33h) - malloc(size)

Allocates size bytes on the heap, and returns the memory handle (aka the address of the allocated memory block). The address of the block is guaranteed to by aligned to 4-byte memory boundaries. Size is rounded up to a multiple of 4 bytes. The address may be in KUSEG, KSEG0, or KSEG1, depending on the address passed to InitHeap.
Caution: The BIOS (tries to) initialize the heap size to 0 bytes (actually it accidently overwrites that initial setting by garbage during relocation), so any call to malloc will fail, unless InitHeap has been used to initialize the address/size of the heap.

A(34h) - free(buf)

Deallocates the memory block. There's no return value, and no error checking. The function simply sets [buf-4]=[buf-4] OR 00000001h, so if buf is an invalid handle it may destroy memory at [buf-4], or trigger a memory exception (for example, when buf=0).

A(37h) - calloc(sizx, sizy) ;SLOW!

Allocates xsiz*ysiz bytes by calling malloc(xsiz*ysiz), and, unlike malloc, it does additionally zerofill the memory via SLOW "bzero" function. Returns the address of the memory block (or zero if failed).

A(38h) - realloc(old_buf, new_size) ;SLOW!

If "old_buf" is zero, executes malloc(new_size), and returns r2=new_buf (or 0=failed). Else, if "new_size" is zero, executes free(old_buf), and returns r2=garbage. Else, executes malloc(new_size), bcopy(old_buf,new_buf,new_size), and free(old_buf), and

returns r2=new_buf (or 0=failed).
Caution: The bcopy function is SLOW, and realloc does accidently copy "new_size" bytes from old_buf, so, if the old_size was smaller than new_size then it'll copy whatever garbage data - in worst case, if it exceeds the top of the 2MB RAM region, it may crash with a locked memory exception, although that'd happen only if SetMem(2) was used to restrict RAM to 2MBs.

A(39h) - InitHeap(addr, size)

Initializes the address and size of the heap - the BIOS does not automatically do this, so, before using the heap, InitHeap must be called by software. Usually, the heap would be memory region between the end of the boot executable, and the bottom of the executable's stack. InitHeap can be also used to deallocate all memory handles (eg. when a new exe file has been loaded, it may use it to deallocate all old memory).
The heap is used only by malloc/realloc/calloc/free, and by the "qsort" function.

B(00h) - alloc_kernel_memory(size)

B(01h) - free_kernel_memory(buf)

Same as malloc/free, but, instead of the heap, manages the 8kbyte control block memory at A000E000h..A000FFFFh. This region is used by the kernel to allocate ExCBs (4x08h bytes), EvCBs (N*1Ch bytes), TCBs (N*0C0h bytes), and the process control block (1x04h bytes). Unlike the heap, the BIOS does automatically initialize this memory region via SysInitMemory(addr,size), and does autimatically allocate the above data (where the number of EvCBs and TCBs is as specified in the SYSTEM.CNF file). Note: FCBs and DCBs are located elsewhere, at fixed locations in the kernel variables area.

Scratchpad Note

The kernel doesn't include any allocation functions for the scratchpad (nor do any kernel functions use that memory area), so the executable can freely use the "fast" memory at 1F800000h..1F8003FFh.

A(9Fh) - SetMem(megabytes)

Changes the effective RAM size (2 or 8 megabytes) by manipulating port 1F801060h, and additionally stores the size in megabytes in RAM at [00000060h].
Note: The BIOS bootcode accidently sets the RAM value to 2MB (which is the correct physical memory size), but initializes the I/O port to 8MB (which mirrors the physical

2MB within that 8MB region), so the initial values don't match up with each other.
Caution: Applying the correct size of 2MB may cause the "realloc" function to crash (that function may accidently access memory above 2MB).

19.16 BIOS Memory Fill/Copy/Compare (SLOW)

Like most A(NNh) functions, below functions are executed in uncached BIOS ROM, the ROM has very high waitstates, and the 32bit opcodes are squeezed through an 8bit bus. Moreover, below functions are restricted to process the data byte-by-byte. So, they are very-very-very slow, don't even think about using them.
Of course, that applies also for most other BIOS functions. But it's becoming most obvious for these small functions; memcpy takes circa 160 cycles per byte (reasonable would be less than 4 cycles), and bzero takes circa 105 cycles per byte (reasonable would be less than 1 cycles).

A(2Ah) - memcpy(dst, src, len)

Copies len bytes from [src..src+len-1] to [dst..dst+len-1]. Refuses to copy any data when dst=00000000h or when len>7FFFFFFFh. The return value is always the incoming "dst" value.

A(2Bh) - memset(dst, fillbyte, len)

Fills len bytes at [dst..dst+len-1] with the fillbyte value. Refuses to fill memory when dst=00000000h or when len>7FFFFFFFh. The return value is the incoming "dst" value (or zero, when len=0 or len>7FFFFFFFh).

A(2Ch) - memmove(dst, src, len) - bugged

Same as memcpy, but (attempts) to support overlapping src/dst regions, by using a backwards transfer when src\<dst (and, for some reason, only when dst>=src+len).
BUG: The backwards variant accidently transfers len+1 bytes from [src+len..src] down to [dst+len..dst].

A(2Dh) - memcmp(src1, src2, len) - bugged

Compares len bytes at [src1..src1+len-1] with [src2..src2+len-1], and (attempts) to return the difference between the first mismatching bytes, ie. [src1+N]-[src2+N], or 0 if there are no mismatches. Refuses to compare data when src1 or src2 is 00000000h, and

returns 0 in that case.
BUG: Accidently returns the difference between the bytes AFTER the first mismatching bytes, ie. [src1+N+1]-[src2+N+1].
That means that a return value of 0 can mean absolutely anything: That the memory blocks are identical, or that a mismatch has been found (but that the NEXT byte after the mismatch does match), or that the function has failed (due to src1 or src2 being 00000000h).

A(2Eh) - memchr(src, scanbyte, len)

Scans [src..src+len-1] for the first occurence of scanbyte. Refuses to scan any data when src=00000000h or when len>7FFFFFFFh. Returns the address of that first occurence, or 0 if the scanbyte wasn't found.

A(27h) - bcopy(src, dst, len)

Same as "memcpy", but with "src" and "dst" exchanged. That is, the first parameter is "src", the refuse occurs when "src" is 00000000h, and, returns the incoming "src" value (whilst "memcpy" uses "dst" in that places).

A(28h) - bzero(dst, len)

Same as memset, but uses 00h as fixed fillbyte value.

A(29h) - bcmp(ptr1, ptr2, len) - bugged

Same as "memcmp", with exactly the same bugs.

19.17 BIOS String Functions

A(15h) - strcat(dst, src)

Appends src to the end of dst. Searches the ending 00h byte in dst, and copies src to that address, up to including the ending 00h byte in src. Returns the incoming dst value. Refuses to do anything if src or dst is 00000000h (and returns 0 in that case).

A(16h) - strncat(dst, src, maxlen)

Same as "strcat", but clipped to "MaxSrc=(min(0,maxlen)+1)" characters, ie. the total length is max "length(dst)+min(0,maxlen)+1". If src is longer or equal to "MaxSrc", then only the first "MaxSrc" chars are copied (with the last byte being replaced by 00h). If src is shorter, then everything up to the ending 00h byte gets copied, but without additional padding (unlike as in "strncpy").

A(17h) - strcmp(str1, str2)

Compares the strings up to including ending 00h byte. Returns 0 if they are identical, or otherwise [str1+N]-[str2+N], where N is the location of the first mismatch, the two bytes are sign-expanded to 32bits before doing the subtraction. The function rejects str1/str2 values of 00000000h (and returns 0=both are zero, -1=only str1 is zero, and +1=only str2 is zero).

A(18h) - strncmp(str1, str2, maxlen)

Same as "strcmp" but stops after comparing "maxlen" characters (and returns 0 if they did match). If the strings are shorter, then comparision stops at the ending 00h byte (exactly as for strcmp).

A(19h) - strcpy(dst, src)

Copies data from src to dst, up to including the ending 00h byte. Refuses to copy anything if src or dst is 00000000h. Returns the incoming dst address (or 0 if copy was refused).

A(1Ah) - strncpy(dst, src, maxlen)

Same as "strcpy", but clipped to "maxlen" characters. If src is longer or equal to maxlen, then only the first "maxlen" chars are copied (but without appending an ending 00h byte to dst). If src is shorter, then the remaining bytes in dst are padded with 00h bytes.

A(1Bh) - strlen(src)

Returns the length of the string up to excluding the ending 00h byte (or 0 when src is 00000000h).

A(1Ch) - index(src, char)

A(1Dh) - rindex(src, char)

A(1Eh) - strchr(src, char) ;exactly the same as "index"

A(1Fh) - strrchr(src, char) ;exactly the same as "rindex"

Scans for the first (index) or last (rindex) occurence of char in the string. Returns the memory address of that occurence (or 0 if there's no occurence, or if src is 00000000h). Char may be 00h (returns the end address of the string). Note that, despite of the function names, the return value is always a memory address, NOT an index value relative to src.

A(20h) - strpbrk(src, list)

Scans for the first occurence of a character that is contained in the list. The list contains whatever desired characters, terminated by 00h.
Returns the address of that occurence, or 0 if there was none. BUG: If there was no occurence, it returns 0 only if src[0]=00h, and otherwise returns the incoming "src" value (which is the SAME return value as when a occurence did occur on 1st character).

A(21h) - strspn(src, list)

A(22h) - strcspn(src, list)

Scans for the first occurence of a character that is (strspn), or that isn't (strcspn) contained in the list. The list contains whatever desired characters, terminated by 00h. Returns the index (relative to src) of that occurence. If there was no occurence, then it returns the length of src. That silly return values do not actually indicate if an occurence has been found or not (unless one checks for [src+index]=00h or so).
***
"The strcspn() function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by s1 which consists entirely of bytes not from the string pointed to by s2."
"The strspn() function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by s1 which consists entirely of bytes from the string pointed to by s2."

***
Hmmmm, that'd be vice-versa?

A(23h) - strtok(src, list) ;first call

A(23h) - strtok(0, list) ;further call(s)

Used to split a string into fragments, list contains a list of characters that are to be treated as separators, terminated by 00h.
The first call copies the incoming string to a buffer in the BIOS variables area (the buffer size is 100h bytes, so the string should be max 255 bytes long, plus the ending 00h byte, otherwise the function destroys other BIOS variables), it does then search the first fragment, starting at the begin of the buffer. Further calls (with src=00000000h) are searching further fragments, starting at the buffer address from the previous call. The internal buffer is used only for strtok, so its contents (and the returned string fragments) remain intact until a new first call to strtok takes place.
The separate fragments are processed by searching the first separator, starting at the current buffer address, the separator is then replaced by a 00h byte, and the old buffer address is returned to the caller. Moreover, the function tries to skip all continously following separators, until reaching a non-separator, and does memorize that address for the next call (due to that skipping further calls won't return empty fragments, the first call may do so though). That skipping seems to be bugged, if list contains two or more different characters, then additional separators aren't skipped.

Once when there are no more fragments, then 00000000h is returned.

A(24h) - strstr(str, substr) - buggy

Scans for the first occurence of substr in the string. Returns the memory address of that occurence (or 0 if it was unable to find an occurence).
BUG: After rejecting incomplete matches, the function doesn't fallback to the old str address plus 1, but does rather continue at the current str address. Eg. it doesn't find substr="aab" in str="aaab" (in that example, it does merely realize that "aab"\<>"aaa" and then that "aab"\<>"b").

19.18 BIOS Number/String/Character Conversion

A(0Eh) - abs(val)

A(0Fh) - labs(val) ;exactly same as "abs"

Returns the absolute value (if val\<0 then R2=-val, else R2=val).

A(0Ah) - todigit(char)

Takes the incoming character, ANDed with FFh, and returns 0..9 for characters "0..9" and 10..35 for "A..Z" or "a..z", or 0098967Fh (9,999,999 decimal) for any other 7bit characters, or garbage for characters 80h..FFh.

A(25h) - toupper(char)

A(26h) - tolower(char)

Returns the incoming character, ANDed with FFh, with letters "A..Z" converted to uppercase/lowercase format accordingly. Works only for char 00h..7Fh (some characters in range 80h..FFh are left unchanged, others are randomly "adjusted" by adding/ subtracting 20h, and by sign-expanding the result to 32bits).

A(0Dh) - strtol(src, src_end, base)

Converts a string to a number. The function skips any leading "blank" characters (that are, 09h..0Dh, and 20h) (ie. TAB, CR, LF, SPC, and some others) (some characters in range 80h..FFh are accidently treated as "blank", too).
The incoming base value should be in range 2..11, although the function does also accept the buggy values in range of 12..36 (for values other than 2..36 it defaults to decimal/base10). The used numeric digits are "0..9" and "A..Z" (or less when base is smaller than 36).
The string may have a negative sign prefix "-" (negates the result) (a "+" is NOT recognized; and will be treated as the end of the string). Additionally, the string may contain prefixes "0b" (binary/base2), "0x" (hex/base16), or "o" (octal/base8) (only "o", not "0o"), allowing to override the incoming "base" value.
BUG: Incoming base values greater than 11 don't work due to the prefix feature (eg. base=16 with string "0b11" will be treated as 11 binary, and base=36 with string "o55" will be treated as 55 octal) (the only workaround would be to add/remove leading "0"

characters, ie. "b11" or "00b11" or "0o55" would work okay).
Finally, the function initializes result=0, and does then process the digits as "result=result*base+digit" (without any overflow checks) unless/until it reaches an unknown digit (or when digit>=base) (ie. the string may end with 00h, or with any other unexpected characters).
The function accepts both uppercase and lowercase characters (both as prefixes, and as numeric digits). The function returns R2=result, and [src_end]=end_address (ie. usually the address of the ending 00h byte; or of any other unexpected end-byte). If src points to 00000000h, then the function returns r2=0, and leaves [src_end] unchanged.

A(0Ch) - strtoul(src, src_end, base)

Same as "strtol" except that it doesn't recognize the "-" sign prefix (ie. works only for unsigned numbers).

A(10h) - atoi(src)

A(11h) - atol(src) ;exactly same as "atoi" (but slightly slower)

Same as "strtol", except that it doesn't return the string end address in [src_end], and except that it defaults to base=10, but still supports prefixes, allowing to use base2,8,16. CAUTION: For some super bizarre reason, this function treats "0" (a leading ZERO digit) as OCTAL prefix (unlike strtol, which uses the "o" letter as octal prefix) (the "0x" and "0b" prefixes are working as usually).

A(12h) - atob(src, num_dst)

Calls "strtol(str,src_end,10)", and does then exchange the two return values (ie. sets R2=[src_end], and [num_dst]=value_32bit).

A(0Bh) - atof(src) ;USES (ABSENT) COP1 FPU !!!

A(32h) - strtod(src, src_end) ;USES (ABSENT) COP1 FPU !!!

These functions are intended to convert strings to floating point numbers, however, the functions are accidently compiled for MIPS processors with COP1 floating point unit (which is not installed in the PSX, nor does the BIOS support a COP1 software emulation), so calling these functions will produce a coprocessor exception, causing the PSX to lockup via A(40h) SystemErrorUnresolvedException.

Note

On other systems (eg. 8bit computers), "abs/atoi" (integer) and "labs/atol" (long) may act differently. However, on the Playstation, both use signed 32bit values.

19.19 BIOS Misc Functions

A(2Fh) - rand()

Advances the random generator as "x=x*41C64E6Dh+3039h" (aka plus 12345 decimal), and returns a 15bit random value "R2=(x/10000h) AND 7FFFh".

A(30h) - srand(seed)

Changes the current 32bit value of the random generator.

A(B4h) - GetSystemInfo(index) ;not supported by old CEX-1000 version

Returns a word, halfword, or string, depending on the selected index value:

Note: The Date/Version are referring to the Kernel (in the first half of the BIOS). The Intro and Bootmenu (in the second half of the BIOS) may have a different version, there's no function to retrieve info on that portion, however, a version string for it can be usually found at BFC7FF32h (eg. "System ROM Version 4.5 05/25/00 E",0) (in many bios versions, the last letter of that string indicates the region, but not in all versions) (the old SCPH1000 does not include that version string at all).

B(56h) - GetC0Table()

B(57h) - GetB0Table()

Retrieves the address of the jump lists for B(NNh) and C(NNh) functions, allowing to patch entries in that lists (however, the BIOS does often jump directly to the function addresses, rather than indirectly via the list, so patching may have little effect in such cases). Note: There's no function to retrieve the address of the A(NNh) jump list, however, that list is usually/always at 00000200h.

A(31h) - qsort(base, nel, width, callback)

Sorts an array, using a super-slow implementation of the "quick sort" algorithm. base is the address of the array, nel is the number of elements in the array, width is the size in bytes of each element, callback is a function that receives pointers to two elements which need to be compared; callback should return return zero if the elements are identical, or a positive/negative number to indicate which element is bigger.
The qsort function rearranges the contents of the array, ie. depending on the callback result, it may swap the contents of the two elements, for some bizarre reason it doesn't swap them directly, but rather stores one of the elements temporarily on the heap (that means, qsort works only if the heap was initialized with InitHeap, and only if "width" bytes are free). There's no return value.

A(35h) - lsearch(key, base, nel, width, callback)

A(36h) - bsearch(key, base, nel, width, callback)

Searches an element in an array (key is the pointer to the searched element, the other parameters are same as for "qsort"). "lsearch" performs a slow linear search in an unsorted array, by simply comparing one array element after each other. "bsearch" assumes that the array contains sorted elements (eg. via qsort), which is allowing to skip some elements, and to jump back and forth in the array, until it has found the desired element (or the location where it'd be, if it'd be in the array). Both functions return the address of the element (or 0 if it wasn't found).

C(19h) - _ioabort(txt1,txt2)

Displays the two strings on the TTY (in some cases the BIOS does accidently pass garbage instead of the 2nd string though). And does then execute _ioabort_raw(1), see there for more details.

A(B2h) - _ioabort_raw(param) ;not supported by old CEX-1000 version

Executes "longjmp(ioabortbuffer,param)". Internally used to recover from failed I/O operations, param should be nonzero to notify the setjmp caller that the abort has occurred.

A(13h) - setjmp(buf)

This is a somewhat incomplete implementation of posix's setjmp, by storing the ABI-saved CPU registers in the specified buffer (30h bytes):

That type of buffer can be used with "_ioabort", "longjmp", and also "HookEntryInt(addr)".
The "setjmp" function returns 0 when called directly. However, it may return again - to the same return address, and the same stack pointer - with another return value (which should be usually non-zero, to indicate that the state has been restored (eg. _ioabort passes 1 as return value).
Also noteworthy from what a compliant setjmp implementation should be doing is the absence of saving the state of cop0 and cop2, thus making this slightly unsuitable for a typical coroutine system implementation.

A(14h) - longjmp(buf, param)

Restores the R16-R23,GP,SP,FP,RA registers from a previously recorded jmp_buf buffer, and "returns" to that new RA address (rather than to the caller of the longjmp function). The "param" value is passed as "return value" to the code at RA, ie. usually to the caller of the original setjmp call. Noteworthy difference from a conformant longjmp implementation is that the "param" value won't be clamped to 1 if you pass 0 to it. So since setjmp returns 0 on the first call, the caller of longjmp must take care that "param" is non-zero, so the callsite of setjmp can make the difference between the first call and a rollback. See setjmp for further details.

A(53h) - set_ioabort_handler(src) ;PS2 only ;PSX: SystemError

Normally the _ioabort handler is changed only internally during booting, with this new function, games can install their own _ioabort handler. src is pointer to a 30h-byte "savestate" structure, which will be copied to the actual _ioabort structure.

A(06h) or B(38h) - exit(exitcode)

Terminates the program and returns control to the BIOS; which does then lockup itself via A(3Ah) _exit.

A(A0h) - _boot()

Performs a warmboot (resets the kernel and reboots from CDROM). Unlike the normal coldboot procedure, it doesn't display the "\<S>" and "PS" intro screens (and doesn't verify the "PS" logo in the ISO System Area), and, doesn't enter the bootmenu (even if the disk drive is empty, or if it contains an Audio disk). And, it doesn't reload the SYSTEM.CNF file, so the function works only if the same disk is still inserted (or another disk with identical SYSTEM.CNF, such like Disk 2 of the same game).

A(B5h..BFh) B(11h,24h..29h,2Ch..31h,5Eh..FFh) C(1Eh..7Fh) - N/A - Jump 0

These functions jump to address 00000000h. For whatever reason, that address does usually contain a copy of the exception handler (ie. same as at address 80000080h). However, since there's no return address stored in EPC register, the functions will likely crash when returning from the exception handler.

A(57h..5Ah,73h..77h,79h..7Bh,7Dh,7Fh..80h,82h..8Fh,B0h..B1h,B3h), and

C(0Eh..11h,14h) - N/A - Returns 0

No function. Simply returns with r2=00000000h.
Reportedly, A(85h) is CdStop, but that seems to be nonsense?

SYS(00h) - NoFunction()

No function. Simply returns without changing any registers or memory locations (except that, of course, the exception handler destroys k0).

SYS(04h..FFFFFFFFh) - calls DeliverEvent(F0000010h,4000h)

These are syscalls with invalid function number in R4. For whatever reason that is handled by issuing DeliverEvent(F0000010h,4000h). Thereafter, the syscall returns to the main program (ie. it doesn't cause a SystemError).

A(3Ah) - _exit(exitcode)

A(40h) - SystemErrorUnresolvedException()

A(A1h) - SystemError(type,errorcode) ;type "B"=Boot,"D"=Disk

These are used "SystemError" functions. The functions are repeatedly jumping to themselves, causing the system to hang. Possibly useful for debugging software which may hook that functions.

A(4Fh,50h,52h,53h,9Ah,9Bh) B(1Ah..1Fh,21h..23h,2Ah,2Bh,52h,5Ah) C(0Bh) - N/A

These are additional "SystemError" functions, but they are never used. The functions are repeatedly jumping to themselves, causing the system to hang.

BRK(1C00h) - Division by zero (commonly checked/invoked by software)

BRK(1800h) - Division overflow (-80000000h/-1, sometimes checked by software)

The CPU does not generate any exceptions upon divide overflows, because of that, the Kernel code and many games are commonly checking if the divider is zero (by software), and, if so, execute a BRK 1C00h opcode. The default BIOS exception handler doesn't handle BRK exceptions, and does simply redirect them to SystemErrorUnresolvedException().

19.20 BIOS Internal Boot Functions

A(45h) - init_a0_b0_c0_vectors

Copies the three default four-opcode handlers for the A(NNh),B(NNh),C(NNh) functions to A00000A0h..A00000CFh.

C(07h) - InstallExceptionHandlers() ;destroys/uses k0/k1

Copies the default four-opcode exception handler to the exception vector at 80000080h..8000008Fh, and, for whatever reason, also copies the same opcodes to 80000000h..8000000Fh.

C(08h) - SysInitMemory(addr,size)

Initializes the address (A000E000h) and size (2000h) of the allocate-able Kernel Memory region, and, seems to deallocate any memory handles which may have been allocated via B(00h).

C(09h) - SysInitKernelVariables()

Zerofills all Kernel variables; which are usually at [00007460h..0000891Fh].
Note: During the boot process, the BIOS accidently overwrites the first opcode of this function (by the last word of the A0h table), so, thereafter, this function won't work anymore (nor would it be of any use).

C(12h) - InstallDevices(ttyflag)

Initializes the size and address of the File and Device Control Blocks (FCBs and DCBs). Adds the TTY device by calling "KernelRedirect(ttyflag)", and the CDROM and Memory Card devices by calling "AddCDROMDevice()" and "AddMemCardDevice()".

C(1Ch) - AdjustA0Table()

Copies the B(32h..3Bh) and B(3Ch..3Fh) function addresses to A(00h..09h) and A(3Bh.. 3Eh). Apparently Sony's compiler/linker can't insert the addresses in the A0h table directly at compilation time, so this function is used to insert them during execution of the boot code.

19.21 BIOS More Internal Functions

Below are mainly internally used device related subfunctions.

Internal Device Stuff

Device Names

Device Names are case-sensitive (usually lowercase, eg. "bu" for memory cards). In filenames, the device name may be followed by a hexadecimal 32bit non-case-sensitive port number (eg. "bu00:" for selecting the first memory card slot). Accordingly, the device name should not end with a hexdigit (eg. "usb:" would be treated as device "us" with port number 0Bh).
Standard device names are "cdrom:", "bu00:", "bu10:", "tty00:". Other, nonstandard devices are:

19.22 BIOS PC File Server

DTL-H2000

Below BRK's are internally used in DTL-H2000 BIOS for two devices: "mwin:" (Message Window) and "sim:" (CDROM Sim).

Caetla Blurb

Caetla (a firmware replacement for Cheat Devices) supports "pcdrv:" device, the SN systems (=what?) device extension to access files on the drive of the pc. This fileserver can be accessed by using the kernel functions, with the "pcdrv:" device name prefix to the filenames or using the SN system calls.
The following SN system calls for the fileserver are provided. Accessed by setting the registers and using the break command with the specified field.
The break functions have argument(s) in A1,A2,A3 (ie. unlike normal BIOS functions not in A0,A1,A2), and TWO return values (in V0, and V1).

BRK(101h) - PCInit() - Inits the fileserver

No parameters.

BRK(102h) - PCCreat(filename, fileattributes) - Creates a new file on PC

Attributes Bits (standard MSDOS-style):

BRK(103h) - PCOpen(filename, accessmode) - Opens a file on the PC

BRK(104h) - PCClose(filehandle) - Closes a file on the PC

BRK(105h) - PCRead(filehandle, length, memory_destination_address)

Note: PCRead does not stop at EOF, so if you set more bytes to read than the filelength, the fileserver will pad with zero bytes. If you are not sure of the filelength obtain the filelength by PClSeek (A2=0, A3=2, V1 will return the length of the file, don't forget to reset the file pointer to the start before calling PCread!)

BRK(106h) - PCWrite(filehandle, length, memory_source_address)

BRK(107h) - PClSeek(filehandle, file_offset, seekmode) - Change Filepos

seekmode may be from 0=Begin of file, 1=Current fpos, or 2=End of file.

19.23 BIOS TTY Console (std_io)

A(3Fh) - Printf(txt,param1,param2,etc.) - Print string to console

Prints the specified string to the TTY console. Printf does internally use "putchar" to output the separate characters (and expands char 09h and 0Ah accordingly).
The string can contain C-style escape codes (prefixed by "%" each):

Additionally, following prefixes (inserted between "%" and escape code):

The force32bit codes (D,U,O,p,l) are kinda useless since the PSX defaults to 32bit parameters anyways. The force16bit code (h) may be useful as "%hn" (writeback 16bit value), otherwise it's rather useless, unless signed 16bit parameters have garbage in upper 16bit, for unsigned 16bit parameters it doesn't work at all (accidently sign-expands 16bit to 32bit, and then displays that signed 32bit value as giant unsigned value). Printf supports only octal, decimal, and hex (but not binary).

A(3Eh) or B(3Fh) - puts(src) - Write string to TTY

Like "printf", but doesn't resolve any "%" operands. Empty strings are handled in a special way: If R4 points to a 00h character then nothing is output (as one would expect it), but, if R4 is 00000000h then "\<NULL>" is output (only that six letters; without appending any CR or LF).

A(3Dh) or B(3Eh) - gets(dst) - Read string from TTY (keyboard input)

Internally uses "getchar" to receive the separate characters (which are thus masked by 7Fh). The received characters are stored in the buffer, and are additionally sent back as echo to the TTY via std_out_putc.
The following characters are handled in a special way: 09h (TAB) is replaced by a single

SPC. 08h or 7FH (BS or DEL) are removing the last character from the buffer (unless it is empty) and send 08h,20h,08h (BS,SPC,BS) to the TTY. 0Dh or 0Ah (CR or LF) do terminate the input (append 00h to the buffer, send 0Ah to the TTY, which is expanded to 0Dh,0Ah by the std_out_putc function, and do then return from the gets function). The sequence 16h,NNh forces NNh to be stored in the buffer (even if NNh is a special character like 00h..1Fh or 7Fh). If the buffer is full (circa max 125 chars, plus one extra byte for the ending 00h), or if an unknown control code in range of 00h..1Fh is received without the 16h prefix, then 07h (BELL) is sent to the TTY.

A(3Bh) or B(3Ch) - getchar() - Read character from TTY

Reads one character from the TTY console, by internally redirecting to "read(0,tempbuf, 1)". The returned character is ANDed by 7Fh (so, to read a fully intact 8bit character, "read(0,tempbuf,1)" must be used instead of this function).

A(3Ch) or B(3Dh) - putchar(char) - Write character to TTY

Writes the character to the TTY console, by internally redirecting to "write(1,tempbuf, 1)". Char 09h (TAB) is expanded to one or more SPC characters, until reaching the next tabulation boundary (every 8 characters). Char 0Ah (LF) is expanded to 0Dh,0Ah (CR,LF). Other special characters (which should be handled at the remote terminal side) are 08h (BS, backspace, move cursor one position to the left), and 07h (BELL, produce a short beep sound).

C(13h) - FlushStdInOutPut()

Closes and re-opens the std_in (fd=0) and std_out (fd=1) file handles.

C(1Bh) - KernelRedirect(ttyflag) ;PS2: ttyflag=1 causes SystemError

Removes, re-mounts, and flushes the TTY device, the parameter selects whether to mount the real DUART-TTY device (r4=1), or a Dummy-TTY device (r4=0), the latter one sends any std_out to nowhere. Values other than r4=0 or r4=1 do remove the device, but do not re-mount it (which might result in problems).
Caution: Trying to use r4=1 on a PSX that does not has the DUART hardware installed causes the BIOS to hang (so one should first detect the DUART hardware, eg. by writing two different bytes to Port 1F802020h.1st/2nd access, and the read and verify that two bytes).

Activating std_io

The std_io functions can be enabled via C(1Bh) KernelRedirect(ttyflag), the BIOS is unable to detect the presence of the TTY hardware, by default the BIOS bootcode disables std_io by setting the initial KernelRedirect value at [A000B9B0h] to zero, this is hardcoded shortly after the POST(E) output:

assuming that R28=A0010FF0h, the last 3 opcodes of above code can be replaced by:

with that patch, the BIOS bootcode (and many games) are sending debug messages to the debug terminal, via expansion port, see:
EXP2 Dual Serial Port (for TTY Debug Terminal)
Note: The nocash BIOS automatically detects the DUART hardware, and activates TTY if it is present.

B(49h) - PrintInstalledDevices()

Uses printf to display the long and short names from the DCB of the currently installed devices. Doesn't do anything else. There's no return value.

Note

Several BIOS functions are internally using printf to output status information, timeout, and error messages, etc. So, trying to close the TTY file handles (fd=0 and fd=1) would cause such functions to work unstable.

19.24 BIOS Character Sets

B(51h) - Krom2RawAdd(shiftjis_code)

r4 should be 8140h..84BEh (charset 2), or 889Fh..9872h (charset 3).

B(53h) - Krom2Offset(shiftjis_code)

This is a subfunction for B(51h) Krom2RawAdd(shiftjis_code).

Character Sets in ROM (112Kbytes)

The character sets are located at BFC64000h and up, intermixed with some other stuff:

Charset 1 (and Garbage) is NOT included in japanese BIOSes (in the SCPH1000 version that region contains uncompressed program code, in newer japanese BIOSes that regions are zerofilled)
Charset 1 symbols are as defined in JIS-X-0212 char(2661h..2B77h), and EUC-JP char(8FA6E1h..8FABF7h).
Version (and Copyright) string is NOT included in SCPH1000 version (that BIOS includes further japanese 8x15 pix chars in that region).
For charset 2 and 3 it may be recommended to use the B(51h) Krom2RawAdd(shiftjis_code) to obtain the character addresses. Not sure if that BIOS function (or another BIOS function) allows to retrieve charset 1, 4, 5, and 6 addresses?

19.25 BIOS Control Blocks

Exception Control Blocks (ExCB) (4 blocks of 8 bytes each)

Event Control Blocks (EvCB) (usually 16 blocks of 1Ch bytes each)

Thread Control Blocks (TCB) (usually 4 blocks of 0C0h bytes each)

Process Control Block (1 block of 4 bytes)

The PSX supports only one process, and thus only one Process Control Block.

File Control Blocks (FCB) (16 blocks of 2Ch bytes each)

Device Control Blocks (DCB) (10 blocks of 50h bytes each)

19.26 BIOS Versions

Kernel Versions

For the actual kernel, there seem to be only a few different versions. Most PSX/PSone's are containing the version from 1995 (which is kept 1:1 the same in all consoles; without any PAL/NTSC related customizations).

The date and version string can be retrieved via GetSystemInfo(index).
The "CEX-7000/-7001" version was only "temporarily" used (when the kernel/gui grew too large they changed the ROM size from 512K to 1024K; but did then figure out that they could use a self-decompressing GUI to squeeze everything into 512K; but they did

accidentally still use the 1024K setting) (newer consoles fixed that and switched back to the old version from 1995) (aside from the different date/version string, the only changed thing is the opcode at BFC00000h, which initializes port 1F801010h to BIOS ROM size of 1MB, instead of 512KB; no idea if that BIOS does actually contain additional data?).
The "CEX-3000 KT-3" version is already almost same as "CEX-3000/1001/1002", aside from version/date, the only differences are at offset BFC00014h..1Fh, and BFC003E0h (both related to Port 1F801060h).

Bootmenu/Intro Versions

This portion was updated more often. It's customized for PAL/NTSC displays, japanese/english language, and (maybe?) region/licence string checks. The SCPH1000 uses uncompressed Bootmenu/Intro code with "\<S>" intro, but without "PS" intro (or, "PS" is shown only on region matches?), newer versions are using selfdecompressing code, with both intro screens. The GUI in older PSX models looks like a drawing program for children, the GUI in newer PSX models and in PSone's looks more like a modernized bathroom furniture, unknown how the PS2 GUI looks like?
Games are communicating only with the Kernel, so the differences in the Bootmenu/Intro part should have little or effect on compatibility (although some I/O ports might be initialized differently, and although some games might (accidently) read different (garbage) values from the ROM).

The System ROM Version string can be found at BFC7FF32h (except in v1.0).

v2.2j/a/e use exactly the same GUI as v2.1 (only the kernel was changed). v2.2d is almost same as v2.2j (but with some GUI patches or so).
v4.1 and v4.5 use exactly the same GUI code for "A" and "E" regions (the only difference is the last byte of the version string; which does specify whether the GUI shall use PAL or NTSC).
v5.0 is playstation 2 bios (4MB) with more or less backwards compatible kernel.

Character Set Versions

The 16x15 pixel charsets at BFC66000h and BFC69D68h are included in all BIOSes, however, the 16x15 portion for letters with accent marks at BFC64000h is included only in non-japanese BIOSes, and in some newer japanese BIOSes (not included in v4.0j, but they are included in v4.3j).
The 8x15 pixel charset with characters 21h..7Fh is included in all BIOSes. In the SCPH1000, this region is followed by additional 8x15 punctuation marks at char 80h and up, however, this region is missing in PS2 BIOS. Moreover, some BIOSes include an incomplete 8x15 japanese character set (which ends abruptly at BF7FFFFFh), in newer BIOSes, some of theses chars are replaced by the version string at BFC7FF32h, and, the remaining 8x15 japanese chars were removed in the PS2 BIOS version.

19.27 BIOS Patches

The original PSX Kernel mainly consists of messy and unstable compiler generated code, and, to the worst, the \<same> author seems to have attempted to use assembler code in some places. In result, most commercial games are causing a greater mess by inserting patches in the kernel code...
Which has been a nasty surprise when making the nocash PSX bios; which obviously wasn't compatible with these patches. The only solutions would have been to insert hundreds of NOPs to make my bios \<exactly> as bloated as the original bios (which I really didn't want to do), or to create anti-patch-patches.

Patches and Anti-Patch-Patches

As shown below, all known patches are invoked by a B(56h) or B(57h) function call. In the nocash PSX bios, these two functions are examining the following opcodes, if the opcodes are a known patch, then the BIOS reproduces the desired behaviour, and does then continue normal execution after those opcodes. If the opcodes are unknown, then the BIOS simply locks up; and shows an error message with the address of that opcodes in the TTY window; info about any such unknown opcodes would be welcome!

Compatibility

If you want to (or need to) use patches, please use byte-identical opcodes as commercial games do (as shown below; only the "xxxx" address digits are don't care), so the nocash PSX bios (or other homebrewn BIOSes) can detect and reproduce them. Or alternately, don't use the BIOS, and access I/O ports directly, which is much better and faster anyways.

patch_missing_cop0r13_in_exception_handler:

In newer Kernel version, the exception handler reads cop0r13/cause to r2, examines the Excode value in r2, and if the exception was caused by an interrupt, and if the next opcode (at EPC) is a GTE/COP2 command, then it does increment EPC by 4. The GTE commands are executed even if an interrupt occurs simultaneously, so, without adjusting EPC, the command would be executed twice. With some commands that'd just waste some clock cycles, with other commands it may cause data to be written twice to the GTE FIFOs, or may re-use the result from the 1st command execution as input to the 2nd execution.
The old "CEX-1000 KT-3" Kernel version did examine r2, but it "forgot" to previously load cop0r13 to r2, so it did randomly examine a garbage value. The patch inserts the missing opcode, used in elo2 at 80033740h, and in Pandemonium II at 8007F3FCh:

Alternately, same as above, but using k0/k1 instead of r10/r9, used in Ridge Racer at 80047B14h:

Alternately, slightly different code used in metal_gear_solid at 80095CC0h, and in alone1 at 800A3ECCh:

Alternately, a bugged/nonfunctional homebrew variant (used by Hitmen's "minimum" demo):

early_card_irq_patch:

Because of a hardware glitch the card IRQ cannot be acknowledged while the external IRQ signal is still LOW, making it neccessary to insert a delay that waits until the signal gets HIGH before acknowledging the IRQ.
The original BIOS is so inefficient that it takes hundreds of clock cycles between the interrupt request and the IRQ acknowledge, so, normally, it doesn't require an additional delay.

However, the central mistake in the IRQ handler is that it doesn't memorize which IRQ has originally triggered the interrupt. For example, it may get triggered by a timer IRQ, but a newer card IRQ may occur during IRQ handling, in that case, the card IRQ may get processed and acknowledged without the required delay.
Used in Metal Gear Solid at 8009AA5Ch, and in alone1 at 800AE2F8h:

Alternately, elo2 uses slightly different code at 8003961Ch:

Note: The above @@wait_lop's should be more preferably done with timeouts (else they may hang endless if a Sony Mouse is newly connected; the mouse does have /ACK stuck LOW on power-up).

patch_uninstall_early_card_irq_handler:

Used to uninstall the "early_card_irq_vector" (the BIOS installs that vector from inside of B(4Ah) InitCARD2(pad_enable), and, without patches, the BIOS doesn't allow to uninstall it thereafter).
Used in Breath of Fire III (SLES-01304) at 8017E790, and also in Ace Combat 2 (SLUS-00404) at 801D23F4:

Alternately, more inefficient, used in Blaster Master-Blasting Again (SLUS-01031) at 80063FF4h, and Raiden DX at 80029694h:

Note: the above code is same as "patch_install_lightgun_irq_handler", except that it writes to r2+70h, instead of r2+80h.

patch_card_specific_delay:

Same purpose as the "early_card_irq_patch" (but for the command/status bytes rather than for the data bytes). The patch looks buggy since it inserts the delay AFTER the acknowledge, but it DOES work (the BIOS accidently acknowledges the IRQ twice; and the delay occurs PRIOR to 2nd acknowledge).
Used in Metal Gear Solid at 8009AAF0h, and in Legacy of Kain at 801A56D8h, and in alone1 at 800AE38Ch:

Alternately, slightly different code used in elo2 at800396D4h, and in Resident Evil 2 at 800910E4h:

patch_card_info_step4:

The "card_info" function sends an incomplete read command to the card; in order to receive status information. After receiving the last byte, the function does accidently send a further byte to the card, so the card responds by another byte (and another IRQ7), which is not processed nor acknowledged by the BIOS. This patch kills the opcode that sends the extra byte.
Used in alone1 at 800AE214h:

patch_pad_error_handling_and_get_pad_enable_functions:

If a transmission error occurs (or if there's no controller connected), then the Pad handler handler does usually issue a strange chip select signal to the OTHER controller slot, and does then execute the bizarre_pad_delay function. The patch below overwrites that behaviour by NOPs. Purpose of the original (and patched) behaviour is unknown. Used by Perfect Assassin at 800519D4h:

Alternately, same as above, but with inefficient nops, used by Sporting Clays at 8001B4B4h:

Alternately, same as above, but without getting PadEnable functions, used in Pandemonium II (at 80083C94h and at 8010B77Ch):

patch_optional_pad_output:

The normal BIOS functions are only allowing to READ from the controllers, but not to SEND data to them (which would be required to control Rumble motors, and to auto-activate Analog mode without needing the user to press the Analog button). Internally, the BIOS does include some code for sending data to the controller, but it doesn't offer a function vector for setting up the data source address, and, even if that would be supported, it clips the data bytes to 00h or 01h. The patch below retrieves the required SetPadOutput function address (in which only the src1/src2 addresses are relevant, the blah1/blah2 values aren't used), and suppresses clipping (ie. allows to send any bytes in range 00h..FFh).
Used in Resident Evil 2 at 80091914h:

Alternately, more inefficient (with NOPs), used in Lemmings at 80036618h:

patch_no_pad_card_auto_ack:

This patch suppresses automatic IRQ0 (vblank) acknowleding in the Pad/Card IRQ handler, that, even if auto-ack is enabled. Obviously, one could as well disable auto-ack via B(5Bh) ChangeClearPAD(int), so this patch is total nonsense. Used in Resident Evil 2 at 800919ACh:

Alternately, same as above, but more inefficient, used in Sporting Clays at 8001B53Ch:

Either way, no matter if using the patch or if using ChangeClearPAD(int), having auto-ack disabled allows to install a custom vblank IRQ0 handler, which is probably desired for most games, however, mind that the PSX BIOS doesn't actually support the same IRQ to be processed by two different IRQ handlers, eg. the custom handler may acknowledge the IRQ even when the Pad/Card handler didn't process it, so pad input may become bumpy.

patch_install_lightgun_irq_handler:

Used in Sporting Clays at 80027D68h (when Konami Lightgun connected):

Alternately, same as above, but more inefficient, used in DQM (Dragon Quest Monsters 1&2) at 80089390h (install) and 800893F8h (uninstall):

Some lightgun games (eg. Project Horned Owl) do (additionally to above stuff) hook the exception vector at 00000080h, the hook copies the horizontal coordinate (timer0) to a variable in RAM, thus getting the timer0 value "closest" to the actual IRQ execution. Doing that may eliminate some unpredictable timing offsets that could be caused by cache hits/misses during later IRQ handling (and may also eliminate a rather irrelevant 1-cycle inaccuracy depending on whether EPC was pointing to a GTE opcode, and also eliminates constant cycle offsets depending on whether early_card_irq_handler was installed and enabled, and might eliminate timing differences for different BIOS versions).

set_conf_without_realloc:

Used in Spec Ops Airborne Commando at 80070AE8h, and also in the homebrew game Roll Boss Rush at 80010B68h and 8001B85Ch. Purpose is unknown (maybe to override improperly defined .EXE headers).

Cheat Devices

CAETLA detects the PSX BIOS version by checksumming BFC06000h..BFC07FFFh and does then use some hardcoded BIOS addresses based on that checksum. The reason for doing that is probably that the Pre-Boot Expansion ROM vector is called with the normal A0h/B0h/C0h vectors being still uninitialized.
Problems are that the hardcoded addresses won't work with all BIOSes (eg. not with the no$psx bios clone, probably also not with the newer PS2 BIOS), moreover, the checksumming can fail with patched original BIOSes (eg. no$psx allows to enable TTY debug messages and to skip the BIOS intro).
The Cheat Firmwares are probably also hooking the Vblank handler, and maybe also

some other functions.
ACTION REPLAY (at least later versions like 2.81) uses the Pre-Boot handler to set a COP0 hardware breakpoint at 80030000h and does then resume normal BIOS booting (which will then initialize important things like A0h/B0h/C0h tables, and will then break when starting the GUI code at 80030000h).
XPLORER searches opcode 24040385h at BFC06000h and up, and does then place a COP0 opcode fetch breakpoint at the opcode address+10h (note: this is within a branch delay slot, which makes COP0 emulation twice as complicated). XPLORER does also require space in unused BIOS RAM addresses (eg. Xplorer v3.20: addr 7880h at 1F002280h, addr 017Fh at 1F006A58h).

Note

Most games include two or three patches. The only game that I've seen so far that does NOT use any patches is Wipeout 2097.

20. Arcade Cabinets

PSX-Based Arcade Boards

CPU

Same as in PSX. Except, one board is said to be having the CPU clocked at 48MHz, instead of 33MHz???

GPU

Same as in PSX. Except, most or all boards are said to have 2MB VRAM instead of 1MB. Unknown how the extra VRAM is accessed... maybe as Y=512..1023... (though the PSX VRAM transfer commands are limited to 9bit Ysiz values, but maybe Y coordinates can be 10bit wide).

ROM vs CDROM

Arcade games are stored on ROM (or FLASH) instead of using CDROM drives.

SPU

Most PSX-based arcade boards are having the PSX-SPU replaced by a different sound chip (with each arcade manufacturer using their own custom sound chip, often controlled by a separate sound CPU).

Controls

Arcade boards are typically having digital joysticks instead of joypads, with differently named buttons (instead of /\,[],(),>\<), probably accessed via custom I/O ports (instead of serial transmission)? Plus additional coin inputs and DIP switches.

Note: There's no documentation for those arcade boards yet, however, it might be possible to extract that info from MAME source code.

21. Cheat Devices

Action Replay, GameShark, Gamebuster, GoldFinger, Equalizer (Datel/clones)

The Datel devices exist in various official/cloned hardware revisions, the DB25 connector requires a special Comms Link ISA card (or a "FiveWire" mod for making it compatible with normal PC parallel ports). Later "PAR3" models are said to not require Comms Link, and do thus probably work directly with normal parallel ports).
Cheat Devices - Datel I/O
Cheat Devices - Datel DB25 Comms Link Protocol
Cheat Devices - Datel Chipset Pinouts
Cheat Devices - Datel Cheat Code Format

Xplorer/Xploder/X-Terminator (FCD/Blaze)

The FCD/Blaze devices are all same hardware-wise (with some cosmetic PCB revisions, and with extra SRAM and bigger FLASH installed in some carts). The DB25 connector can be directly connected to a PC parallel port.
Cheat Devices - Xplorer Memory and I/O Map
Cheat Devices - Xplorer DB25 Parallel Port Function Summary
Cheat Devices - Xplorer DB25 Parallel Port Command Handler
Cheat Devices - Xplorer DB25 Parallel Port Low Level Transfer Protocol
Cheat Devices - Xplorer Versions
Cheat Devices - Xplorer Chipset Pinouts
Cheat Devices - Xplorer Cheat Code Format
Cheat Devices - Xplorer Cheat Code and ROM-Image Decryption

FLASH Chips (for both Xplorer and Datel)

Cheat Devices - FLASH/EEPROMs

http://gamehacking.org/faqs/hackv500c.html - cheat code formats
http://doc.kodewerx.org/hacking_psx.html - cheat code formats
http://xianaix.net/museum.htm - around 64 bios versions
http://www.murraymoffatt.com/playstation-xplorer.html - xplorer bioses

Separating between Gameshark and Xplorer Codes

Codebreaker
Megacom Power Replay III Game Enhancer

21.1 Cheat Devices - Datel I/O

Datel Memory and I/O Map (for PAR2 or so)

Datel PAR1

Original PAR1 might have supported only 128K FLASH (?) if so, the I/O ports are probably same as above, but without the "second 128K" FLASH area.

Datel PAR3

The PAR3 version is said to work with parallel ports (not needing the Comms Link IDA card), and said to support more FLASH with bankswitching, so the I/O ports must work somehow entirely different as described above.
Some notes from a (poorly translated) japanese document:
PAR3 Memory:

PAR3 I/O:

21.2 Cheat Devices - Datel DB25 Comms Link Protocol

Boot Command Handler

The Boot Command Handler is executed once at Pre-Boot, and also (at least in some firmware versions) once before displaying the GUI. Following command(s) can be sent from PC side:

Data is always transferred as byte-pair (send one byte, receive one byte), 16bit/32bit values are transferred MSB first (with ECHO after each byte).
The upload/exec command is supported by both Datel and Caetla, the upload/flash command is supported by Datel only (but it's totally bugged in PAR1.99, and might also have upwards compatiblity issues in later versions, so it's better to upload a custom flash function via upload/exec instead of using upload/flash).

The 16bit checksum is all DATA[len] bytes added together, and then ANDed with 0FFFh (ie. actually only 12bit wide).

Menu/Game Command Handler

There must be some further command handler(s) after the Boot Command Handler, with support for additional cheat related commands, and (at least in Caetla) with support for uploading EXE files with Kernel functions installed (the Boot Command Handler at Pre-Boot time can also upload EXE code, but doesn't have Kernel installed).
Original Datel commands for Menu/Game mode are unknown. The Caetla commands are documented in japanese, and there are also two english translations:
http://www.psxdev.net/forum/viewtopic.php?f=49&t=370 - good (though incomplete)
http://www.psxdev.net/forum/viewtopic.php?f=53&t=462#p6849 - very bad (beware)

21.3 Cheat Devices - Datel Chipset Pinouts

There appear to be numerous Datel hardware revisions (and possibly numerous Datel clones). So this chapter is unlikely to cover all hardware revisions.

DATEL REF1288 board (with DATEL ASIC1 chip)

The ASIC1 chip is found in this hardware:

The ASIC1 is basically same as the PAL/GAL on other boards, with the 74HC245 transceiver intergrated; the ASIC1 is using a 44pin PLCC package, with pin1 being upper-middle, and pin7 being upper-left. Pinouts are:

D0 is wired to both pin7 and pin29. The /MODE pin is NC (but could be GNDed via the two solder points in middle of the PCB). The SWITCH has 10K pullup (can can get GNDed depending on switch setting).

PALCE20V8 Cuthbert Action Replay schematic (from hitmen webpage)

Charles MacDonald Game Shark schematic

Uhm, schematic shows "PAR.ACK" instead of "BUF.DIR" as transceiver direction?
The 24pin PAL in Charles schematic does actually seem to be a 28pin PLCC GAL in actual hardware (which has four NC pins, hence the 24pin notation in the schematic).
The three PIC pins connect to a 28pin PIC16C55 microprocessor (unknown purpose).
Most of the PIC pins are NC (apart from the above three signals, plus supply, plus OSC ... derived from some oscillator located "behind" the DB25 connector?).

Charles MacDonald Gold Finger schematic

Note: This is a datel clone, without "BUF.DIR" signal (instead, the transceiver DIR pin is wired to "PAR.ACK"; it's probably functionally same as real datel hardware, assuming that "PAR.ACK" is only a short pulse during writing; then reading should be possible anytime else).

Charles MacDonald Comms Link schematic

PAL

The JP1/JP2 pins allow to select Port 300h,310h,320h,330h via two jumpers. The /IRQ pin could be forward to ISA./IRQ2..7 via six jumpers (but the feature is ununsed and the six jumpers aren't installed at all).

DB25 Connector

nocash FiveWire mod (for connecting datel expansion cart to parallel port)

21.4 Cheat Devices - Datel Cheat Code Format

PSX Gameshark Code Format

Below for v2.2 and up only

Below for v2.41 and up only

Below probably v2.41, too (though other doc claims for v2.2)

Below probably v2.41, too (though other doc claims for ALL versions)

Below from Caetla .341 release notes

These are probably caetla-specific, not official Datel-codes. In fact, Caetla .341 itself might be an inofficial hacked version of Caetla .34 (?) so below might be totally inofficial stuff:

Notes

A maximum of 30 increment/decrement codes can be used at a time.
A maximum of 60 conditionals can be used at a time (this includes Cx codes).
Increment/decrement codes should (must?) be used with conditionals.

Unknown if greater/less conditionals are signed or unsigned.
Unclear if greater/less compare dddd by [aaaaaa], or vice-versa.
Unknown if 16bit codes do require memory alignment.

21.5 Cheat Devices - Xplorer Memory and I/O Map

Xplorer Memory Map

FLASH can be 256Kbyte (normal), or 512Kbyte (in FX versions). When programming FLASH chips: Observe that the carts can be fitted with chips from different manufacturers, and, Xplorer carts can have either one or two 256K chips, or one 512K chip.
SRAM can be 0Kbyte (normal/none), or 128Kbyte (in FX versions). The PCB supports max 512K SRAM (but there aren't any carts having that much memory installed).

Xplorer I/O Map

21.6 Cheat Devices - Xplorer DB25 Parallel Port Function Summary

Xplorer Parallel Port Commands (from PC side)

Function names starting with "Game/Menu" and/or "New/Mid/Old" are working only in Game/Menu mode, or only in New/Old xplorer firmware versions (new commands exist in v4.52, old commands exist in v1.091, mid commands exist in v2.005, but neither in v1.091 nor v4.52, unknown when those new/mid/old commands have been added/ removed exactly, in which specific versions).

The only useful command is SetMemAndExecute, which works in ALL versions, and which can be used to do whatever one wants to do (unfortunately, most of the official & inoffical tools are relying on other weird commands, which are working only with specific xplorer firmware versions).

21.7 Cheat Devices - Xplorer DB25 Parallel Port Command Handler

The command handler is called once and then during booting, during xplorer GUI, and during Game execution.
Each call to the command handler does allow to execute ONLY ONE command, however, the "Freeze" command can be used to force the xplorer to stay in the command handler, so one can send MORE commands, until leaving the command handler by sending the "Unfreeze" command.
The command handling can vary depending on current boot phase (see below cautions on Pre-Boot, Mid-Boot, and In-Game phases).

Pre-Boot Handler

This is called shortly after the kernel has done some basic initialization, and after the xplorer has relocated its EEPROM content to RAM (which means it may called about a second after reset when using official PSX kernel and Xplorer Firmware).

Observe that the Kernel function vectors at A0h, B0h, and C0h aren't installed at this point. If you want to upload an EXE with Kernel vectors installed: send THREE dummy commands (eg. Unfreeze) to skip the above early command handling. On the other hand, the ReBootKernel command can be used if you WANT to upload something during Pre-Boot (the ReBootKernel command works only in MENU mode though, ie. during Xplorer GUI, but not during Game).

Mid-Boot Handler (Xplorer GUI)

The Xplorer GUI is called only if the Pre-Boot handler has installed it (eg. if the SWITCH was ON). The handler is called alongsides with joypad reading (which does NOT take place during the Xplorer intro, so there will be a long dead spot between Pre-Boot and Mid-Boot command handling).

Observe that the GUI may have smashed various parts of the Kernel initialization, so you can upload EXE files, and can use Kernel functions, but the EXE won't get booted in same state as when booting from CDROM. The boot state can also vary dramatically depending on the Xplorer Firmware version.

Post-Boot Handler (at start of CDROM booting)

This is called when starting CDROM booting.

In-Game Handler (after CDROM booting) (...probably also DURING booting?)

This is called via the hooked B(17h) ReturnFromException() handler.

Observe that GAME mode doesn't support all commands. And, above will work only if the game does use B(17h), eg. when using non-kernel exception handling, or if it has crashed, or disabled exceptions. Some internal kernel functions are using

ReturnFromException() directly (without going through the indirect B(17h) function table entry; so the hook cannot trap such direct returns).

21.8 Cheat Devices - Xplorer DB25 Parallel Port Low Level Transfer Protocol

All 16bit/24bit/32bit parameters are transferred MSB first.

Tx(Data) - transmit data byte(s)

Rx(Data) - receive data byte(s)

RxFast(Data) for TurboGetMem - slightly faster than normal Rx(Data)

First, for invoking the Turbo transfer:

Thereafter, receive the actual Data byte(s) as so:

The /ACK transitions can be sensed by polling the parallel port IRQ flag on PC side.

RxFaster(Data) for OptimalGetMem - much faster than normal Rx(Data)

First, for invoking the Turbo transfer:

Thereafter, receive the actual Data byte(s) as so:

BUG: The first received byte will be garbage with upper and lower 4bit both containing the lower 4bits (the bugged firmware does explicitely want DATA=00h before transfer, although DATA=00h is also 'confirming' that the upper 4bit can be 'safely' replaced by lower 4bit).

TxRxChksum for SetMem/GetMem functions

The 16bit checksum is all bytes in Data[Len] added together. The two final response bytes should be "OK"=Okay, or, if the transmitted chksum didn't match, either "CF"=ChecksumFail (for SetMem functions) or "BG"=BadGetChecksum (for GetMem functions). MenuSetMemAndBurnFirm is a special case with three response codes: "OF"=FlashOkay, "CF"=ChecksumFail, "FF"=FlashWriteFail.

TxFilename for Memcard (bu) functions

This is internally using the standard "SetMem" function; preceeded by Rx(Addr32). Whereas Addr is the target address for the filename (just pass the Rx'ed address to the Tx part), Len should be max 38h, Data should be the filename with ending zero (eg. "bu10:name",00h).

TxFiledata for Memcard (bu) WriteFile

This is also using the standard "SetMem" function, plus some obscure extra's. The filedata is split into fragments, Len should be max 2000h per fragment.

RxDataFFEEh for Memcard (bu) ReadFile and GetWhatever

Memcard ReadFile does transfer N fragments of Len=2000h (depending on filesize). The GetWhatever function transfers one fragment with Len=80h, followed by N*6 fragments with Len=40Ah.

RxTurbo for Memcard (bu) GetDirectory/GetFileHeader functions

This is internally using the standard "TurboGetMem" function; preceeded by Rx(Addr32). Whereas Addr is the source address of the actual data (just pass the Rx'ed address to the Tx part).
For GetDirectoy, Len should be max 800h (actual/data data is only 4B0h bytes, ie. 258h bytes per memcard, aka 28h bytes per directory entry). For GetFileHeader, Len should be max 80h.

21.9 Cheat Devices - Xplorer Versions

Xplorer names

Xplorer suffices

The V1/V2/V3 suffix does just indicate the pre-installed firmware version (so that suffices become meaningless after software upgrades).
The FX suffix (or DX in japan) indicates that the PCB contains more memory and an extra resistor (the memory/resistor are intended for use with the "X-Assist" add-on device).

Xplorer PCB types

Xplorer Compatibility Issues

The three PCB versions are functionally identical, and do differ only by cosmetic changes for alternate/smaller chip packages.
However, some things that can make difference in functionality are the installed components and installed firmware version:

X-Assist add-on for Xplorer carts

The X-Assist is a quity huge clumsy controller with DPAD, plus 4 buttons, plus small LCD screen. The thing connects to the Xplorer's DB25 connector, allowing to enter/search cheat codes without using a PC.
The device works only with "FX" Xplorer boards (which contain an extra resistor for outputting supply power on the DB25 connector, plus more memory which is somewhat intended for use by the X-Assist thing).

21.10 Cheat Devices - Xplorer Chipset Pinouts

Xplorer Pinout GAL20V8 (generic array logic)

The GALs are programmed nearly identical for all Xplorer versions, some small differences are: One or two EEPROM chip selects (depending on EEPROM chipset), and extra ports at 1F060005h, 1F060006h, 1F060007h (used in v4.52).
Note: The 28pin PLCC GAL has same pinout as the 24pin chip, but with four NC pins inserted (at pin 1,8,15,22, whereof, there is a wire routed "through" pin 8, so that pin isn't literally NC).

Xplorer Pinout 74373 (8bit tristate latch)

Xplorer Pinout 74245 (8bit bus transceiver)

Xplorer Pinout 7805 (voltage regulator)

Xplorer Pinout SWITCH (on/off)

Xplorer Pinout DB25 (parallel/printer port)

EEPROM.pin1 is NC on 256Kx8 chip (however it is wired to A18 for use with 512Kx8 chips).
EEPROM.pin30 is A17 from GAL.pin21 (not from PSX.A17), accordingly GAL.pin21 is EEPROM.A17 (not A14).
Boards with solder pads for TWO EEPROMs are leaving A18 not connected on the 2nd EEPROM (but do connect A18 to the first EEPROM, so one could either use one 512K chip or two 256K chips).
DB25.pin15./ERR is VCC via 0.47ohm (installed only on carts with SRAM, intended as supply for the X-ASSIST thing).
SRAM (if any) is wired to GAL.pin17 (/CE), 74373.Q6 (A17 or CE2), 74373.Q7 (A18 or NC), other SRAM pins are wired straight to D0-D7, A0-A16, /RD, /WR.

VCC is 5V, derived from a 7805 voltage converter (with 7.5V used as input).
Existing boards seem to have 128K SRAM (if any), so SRAM A17/A18 aren't actually used (unless a board would have 512K SRAM), however, for 128K SRAMs one should switch SRAM CE2 (aka A17) high.

21.11 Cheat Devices - Xplorer Cheat Code Format

PSX Xplorer/Xploder Code Format

The second code digit (t) contains encryption type (bit0-2), and a "default on/off" flag (bit3: 0=on, 1=off; whatever that means, it does probably require WHATEVER actions to enable codes that are "off"; maybe via the Ftaaaaaa dddd code).

break_type (cccc) (aka MSBs of cop0r7 DCIC register)

The CPU supports one data breakpoint and one instruction breakpoint (though unknown if the Xplorer does support to use both simultaneously, or if it does allow only one of them to be used).
If the break_type/address/mask to match up with CPU's memory access actions... then

"something" does probably happen (maybe executing a sub-function that consists of the d0,d1,d2,etc-bytes, if so, maybe at a fixed/unknown memory address, or maybe at some random address; which would require relocatable code).

Notes

The "Slide" code shall be used only with even addresses, unknown if other 16bit/32bit codes do also require aligned addresses.

21.12 Cheat Devices - Xplorer Cheat Code and ROM-Image Decryption

decrypt_xplorer_cheat_code:

decrypt_xplorer_fcd_rom_image:

21.13 Cheat Devices - FLASH/EEPROMs

FLASH/EEPROM Commands

Below commands should work on all chips (for write: page size may vary, eg. 1 byte, 128 bytes, or 256 bytes). Some chips do have some extra commands (eg. an alternate older get id command, or sector erase commands, or config commands), but those extras aren't needed for basic erase/write operations.

Above addresses are meant to be relative to the chip's base address (ie. "5555h" would be 1F005555h in PSX expansion ROM area, or, if there are two flash chips, then it would be 1F045555h for the 2nd chip in xplorer and datel carts; whereas, that region is using bank switching in xplorer carts, so one must output some FLASH address bits I/O ports, and the others via normal CPU address bus; whilst datel carts have noncontinous FLASH areas at 1F000000h and 1F040000h, with a gap at 1F020000h).
Observe that the chips will output status info (instead of FLASH data) during write/erase/id mode (so program code using those commands must execute in RAM, not in FLASH memory).

FLASH/EEPROM Wait Busy

Waiting is required after chip erase and page write (after writing the last byte at page end), and on some chips it's also required after enter/exit id mode. Some chips indicate busy state via a toggle bit (bit6 getting inverted on each 2nd read), and/or by outputting a value different than the written data, and/or do require hardcoded delays (eg. AM29F040). Using the following 3-step wait mechanism should work with all chips:

Whereas, "addr" should be the last written address (or 0000h for erase and enter/exit id mode). And "data" should be the last written data (or FFh for erase, or "don't care" for enter/exit id mode).

Board and Chip Detection

First of, one should detect the expansion board type, this can be done as so:

Next, detect the Chip ID for the (first) FLASH chip:

Finally, one needs to check if there's a second FLASH chip, there are two such cases:

In both cases, the 2nd chip would be mapped at 1F400000h, and one can test the following four combinations:

For each combination compare 400h bytes at 1F000000h vs 1F400000h.

In the latter case, do Chip ID detection at 1F400000h to see if there's really another chip there, and which type it is (if present, then it should be usually the same type as the 1st chip; and if it's not present, then there might be just open bus garbage instead of valid ID values).

FLASH/EEPROM Chip IDs

The above Atmel/SST/Winbond chips are commonly used in Datel or Xplorer carts (or both). The CATALYST chip is used in some Datel clones (but seems to require 12 volts, meaning that it can't be properly programmed on PSX, nethertheless, it's reportedly working "well enough" to encounter flash corruption upon programming attempts). The two ST/AMD chips aren't really common in PSX world (except that I've personally used them in my PSones).

22. PSX Dev-Board Chipsets

Sony DTL-H2000 CPU Board

Sony DTL-H2000 PIO Board

JP715 must be either connected to an external CDROM drive, or to some of "terminator" plug (which shortcuts Pin23 and Pin26 with each other; software may hang upon certain I/O operations without that terminator).

Sony DTL-H2500 Dev board (PCI bus)

Newer revision of the DTL-H2000 board. Consists of a single PCI card (plus tiny daughterboard with Controller ports).

Sony DTL-H2700 Dev board (ISA bus) (CPU, ANALYZER ...?)

Another revision of the DTL-H2000/DTL-H2500 boards. Consists of a single ISA card stacked together with two huge daughterboards, and probably additionally having a small connector daughterboard. Exact chipset is unknown (there might be components on both sides of the PCBs, most of them not visible due to the PCB stacking, so taking photos/scans of the PCBs would require advanced techniques with screwdrivers).
Currently the only known chip name is an EPROM (MX 27C1000DC-90, with sticker "Title=DTL-H2700, Ver=1.00, Date=96.12.4, Sum=046B No."). The ISA card is having markings: "SONY HCD MWB-7? MADE IN JAPAN, PA47 1-589-003-01 1642E03A0".
One uncommon feature is an extra connector for a "trigger switch" (foot pedal), which is reportedly used for activating performance analyzer logging.

Sony DTL-H201A / DT-HV - Graphic Artist Board (IBM PC/ATs to NTSC video)

DTL-S2020 aka Psy-Q CD Emu

Note: There's also a similar ISA cart (DTL-S510B) with less chips and less connectors.
Note: The SN Systems carts seem to have been distributed by Sony (with "DTL-Sxxxx"

numbers), and also distributed by Psygnosis. The external SCSI connectors can be possibly also used with Psy-Q Development Systems for SNES and Sega Saturn?

PSY-Q Development System (Psygnosis 1994)

Sony DTL-H800 Sound Artist Board (with optical fibre audio out)

Note: There's also a similar board (DTL-H700) for MAC/NuBus instead of PCI bus.

Sony COH-2000 (unknown purpose)

Unknown what COH-2000 was used for. One theory was that it's related to PSX-based arcade cabinets. The 68pin connector might be also related to the 68pin PSX "Parallel I/O" expansion port.

Sony DTL-H2010 (Black External CDROM Drive for DTL-H2000, CD-R compatible)

External front loading CDROM drive with Eject button. Connects to the blue 40pin connector on DTL-H2000 boards.

The required cable consists of a Yamaichi NFS-40a female connector (blue connector on DTL-H2000 side), 0.635mm pitch ribbon cable, and 3M Sub-D MDR40 connector (silver connector on DTL-H2010 side). But caution: the odd/even pins on the cable are somewhat swapped, on DTL-H2000 side the wires should be ordered 1,2,3,4,..,39,40, but on DTL-H2010 side they should be ordered 2,1,4,3,..,40,39.

Sony DTL-H2510 (Gray Internal CDROM Drive)

This is some sort of a mimmicked front loading PC CDROM drive (consisting of a tray that contains a normal (top-loading) PSX cdrom drive unit).

There is no eject button, unknown if there's some eject motor, or if one needs to push/ pull the drive tray manually.

Sony SCPH-9903 (Gray SCEx-free Playstation)

A rare SCEx-free Playstation that can boot from CDR's without SCEx strings; maybe intended for beta-testers. Marked "Property of Sony Computer Entertainment", "U/C".

23. Hardware Numbers

Sony's own hardware (for PSX) (can be also used with PSone)

Sony's own hardware (for PSone)

Sony's own devkits

SN System / Psy-Q devkit add-ons / SCSI cards

Sony Licensed Hardware (Japan)

And, maybe unlicensed (they don't have official SLPH numbers, still they are listed as official controllers on PSX CDROM back covers):

And whatever:

Sony Licensed Hardware (Europe)

And, maybe unlicensed:

Sony Licensed Hardware (USA)

Sony Licensed Hardware (Asia)

Newer hardware add-ons?

Note

Early SLEH/SLUH devices used 4-digit numbers (eg. the "official" name for SLEH-00003 is SLEH-0003; unlike as shown in the above list).

Software (CDROM Game Codes)

Note: Multi-disc games have more than one game code. The game code for Disc 1 is also printed on the CD cover, and used in memory card filenames. The per-disk game codes are printed on the discs, and are used as boot-executable name in SYSTEM.CNF file. There is no fixed rule for the multi-disc numbering; some games are using increasing numbers of XNNNN or NNNNX (with X increasing from 0 upwards), and some are randomly using values like NNNXX and NNNYY for different discs.

24. Pinouts

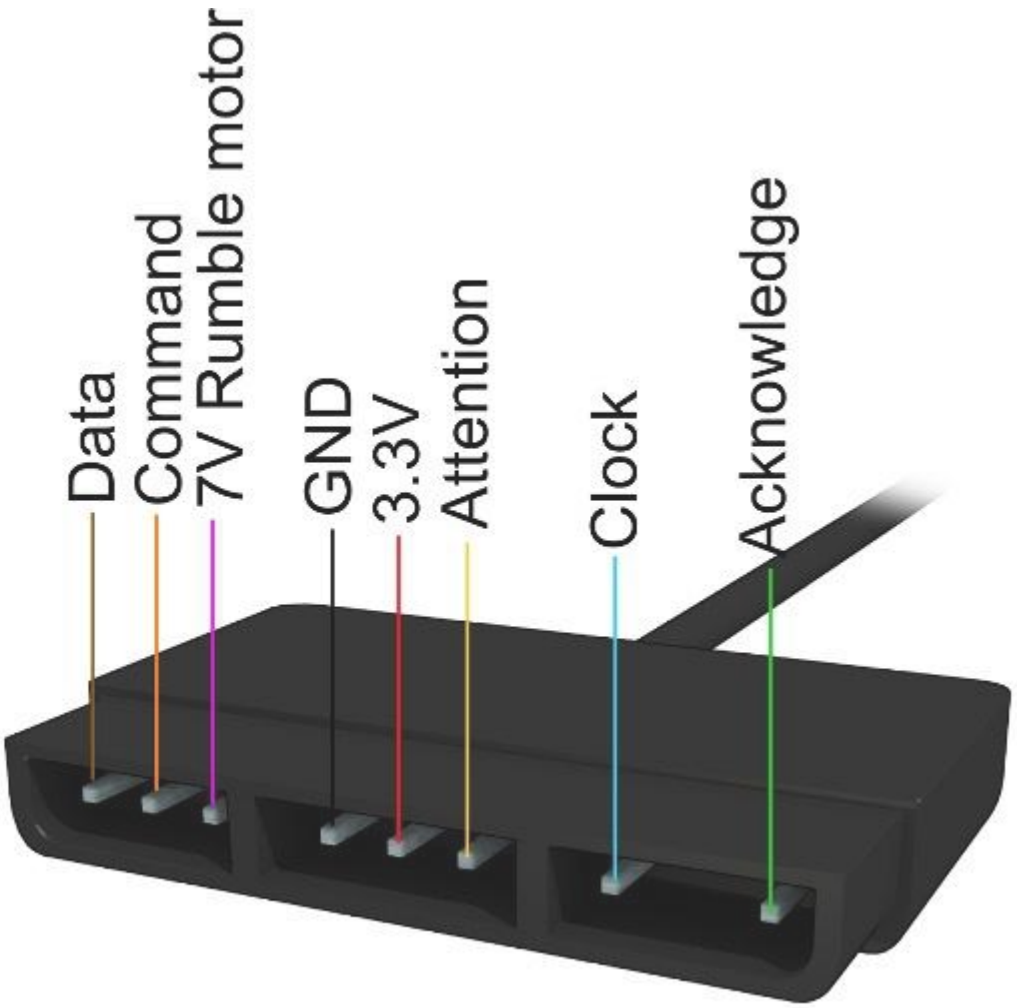
External Connectors

Internal Pinouts

Mods/Upgrades

24.1 Pinouts - Controller Ports and Memory-Card Ports

Controller Ports and Memory-Card Ports



/JOYn are two separate signals (/JOY1 for left card/pad, /JOY2 for right card/pad) (whether it is an card or pad access depends on the first CMD bit). All other signals are exactly the same on all four connectors (except that pin8 and shield are missing on the card slots).

Pin8 (/IRQ10)

Most or all controllers leave pin8 unused, the pin can be used as lightpen input (not sure if the CPU is automatically latching a timer somewhere?), if there's no auto-latched timer, then the interrupt would be required to be handled as soon as possible; ie. don't disable interrupts, and don't "halt" the CPU for longer periods (as far as I understood, the GTE can halt the CPU when trying to read results of incomplete operations; to avoid that, one could wait by software, eg. inserting NOPs, before reading GTE results...?)
(Some (or maybe all?) existing psx lightguns are reportedly connected to the Video output on the Multiout port for determining the current cathode ray position though).

24.2 Pinouts - Audio, Video, Power, Expansion Ports

AV Multi Out (Audio/Video Port)

The standard AV-cable connects only to Pins 7,9,10,11,12,Shield (with pin 1 and 3 and Shield shortcut with each other, used for both audio and video ground).
The plug on that cable does have additional sparings for pin 1,3,5 (though without any metal-contacts installed in there) (pin 3,5 would be used as supply for external RF modulators) (no idea what pin 1 could be used for though?).
RGB displays may (or may not) be able to extract /SYNC from the Composite signal, if that doesn't work, note that /SYNC (and separate /VSYNC, /HSYNC signals) are found on the GPU pinouts, moreover, the GPU outputs 24bit digital RGB.
Not sure if a VGA monitor can be connected? The SYNC signals are there (see GPU pinputs), but the vertical resolution is only 200/240 lines... standard VGA displays probably support only 400/480 lines (or higher resolutions for newer multisync SVGA

displays) (as far as I know, the classic 200 lines VGA mode is actually outputting 400 lines, with each line repeated twice).

Parallel Port (PIO) (Expansion Port) (CN103)

This port exists only on older PSX boards (not on newer PSX boards, and not on PSone boards).
The parallel port is used by Gameshark, Game Enhancer II, and Gold Finger cheat devices (not used by the Code Breaker CDROM cheat software).

Lots of pins are still unknown?

Internal Power Supply (PSX)

The PSX contains an internal power supply, however, like the PSone, it's only having a "Standby" button, which merely disconnects 3.5V and 7.9V from the mainboard. The actual power supply remains powered, and wastes energy day and night, thanks Sony!

External Power Supply (PSone)

24.3 Pinouts - SIO Pinouts

Serial Port

That port exists only on original Playstation (not on the PSone). The shape of the Serial Port is identical to the 12pin Multiout (audio/video) port, but with only 8pins.

Can be used to communicate with another PSX via simple cable connection. With an external RS232 adaptor (for voltage conversion) it could be also used to communicate with a PC, a mouse, a modem, etc.

PSone Serial Port

The PSone doesn't have an external serial connector, however, easy to use soldering points for serial port signals are found as cluster of 5 soldering points (below CPU pin52), and a single soldering point (below CPU pin100), arranged like so (on PM-41 boards) (might be different on PM-41(2) boards):

The three outputs (RTS,DTR,TXD) are left floating, the RXD input is wired via a 1K ohm pull-up resistor to 3.5V, the other two inputs (CTS,DSR) are wired via 1K ohm pull-down resistors to GND.
If you want to upgrade the PSone, remove that resistors, and then install the PSX-style serial circuit (as shown below), or, think of a more simplified circuit without (dis-)inverted signals.

PSX Serial Port Connection (PU-23 board) (missing on PM-41 board)

The PSX serial circuit basically consists of a few transistors, diodes, and resistors. The relevant part is that most of the signals are inverted - compared with RS232 signals, the CPU uses normal high/low levels (of course with 0V and 3.5V levels, not -12V and +12V), and the signals at the serial port socket are inverted. Ie. if you want to built a RS232 adaptor, you must either externally undo the inversion, or, disconnect the transistors, and wire your circuit directly to the CPU signals.

All six signals are passed through fuses (or loops or so). The three inputs have 1K ohm pull-ups, and diodes as protection against negative voltages, two of the inputs are inverted via transistors, with 470 ohm pull-ups at the CPU side, the other input is passed through 22 ohm to the CPU. The three outputs are also passed through 22 ohm, one of them having a diode as negative voltage protection, the other two are inverted via transistors (which may also serve as negative voltage protection).
Note that there is no positive voltage protection (ie. +12V inputs would do no good, also strong -12V inputs might overheat the diodes/fuses, so if you want to use RS232 voltages, better use a circuit for voltage conversion).

Serial RS232 Adaptor

The PSX serial port uses 0V/3.5V logic, whilst RS232 uses -5V/+5V...-15V/+15V logic.
An example circuit for converting the logic levels would be:

Parts List: 1 or 2 MAX232 chips (voltage conversion), 0 or 1 7400 (NAND, used as inverter), 4 or 8 1uF/16V capacitors, 1x 10uF/16V capacitor, 1x 9pin male SubD plug.
The four inverters are needed only for external adapters (which need to undo the transistor inversion on the PSX mainboard) (ie. the inverters are not needed when when connecting the circuit directly to the PSX CPU).
The second MAX232 chip is needed only if DTR/DSR "not ready" conditions are required (for an "always ready" condition: DSUB.4.DTR can be wired to -8.5V, which is available at Pin6 of the first MAX232 chip, and PSX.DSR can be wired to +3.5V).
With the above DSUB pin numbers, peripherals like mice or modems can be connected directly to the circuit. For connection to another computer, use a "null modem" cable (with crossed RXD/TXD, RTS/CTS, DTR/DSR wires).
The circuit works with both VCC=5V (default for MAX232) and with VCC=3.5V (resulting

in slightly weaker signals, but still strong enough even for serial mice; which are misusing the RS232 signals as power supply).

24.4 Pinouts - Chipset Summary

PSX/PSone Mainboards

There are at least two revisions of the "PM-41" board:

The "incomplete" board reportedly requires to solder one wire to the multiout port to make it fully functional... though no idea which wire... looks like the +5V supply? Also, the capacitors near multiout are arranged slightly differently.

CPU chips

These chips contain the MIPS CPU, COP0, and COP2 (aka GTE), MDEC and DMA.

GPU chips - Graphics Processing Unit

SPU chips - Sound Processing Unit

IC106 CPU-RAM / Main RAM chips

GPU-RAM / Video RAM chips

Note: The older 64pin VRAM chips are special dual-ported DRAM, the newer 100pin VRAM chips are just regular DRAM.
Note: The PM-41 board uses a 2MB VRAM chip (but allows to access only 1MB)
Note: The PM-41(2) board has on-chip RAM in the GPU (no external memory chip)

IC310 - SPU-RAM - Sound RAM chips

Note: The PM-41(2) board has on-chip RAM in the SPU (no external memory chip)

BIOS ROM

Oscillators and Clock Multiplier/Divider

Voltage Converter (for +7.5V to +5.0V conversion)

Pulse-Width-Modulation Power-Control Chip

The PM-41 board has locations for both IC606 and IC607, some boards have the bigger IC606 (10mm) installed, others the smaller IC607 (5mm), both chips have exactly the same pinouts, the only difference is the size.

Reset Generator

CDROM Chips

Note: The SUB-CPU contains an on-chip BIOS (which does exist in at least seven versions, plus US/JP/PAL-region variants, plus region-free debug variants).

RGB Chips

MISC

Controller/Memory Card Chips

24.5 Pinouts - CPU Pinouts

CPU Pinouts (IC103)

Pin5-68 = Main RAM bus. Pin 95-152 = System bus. Pin 102,153,159-206 = Video bus.

CPU Pinout Notes

Pin 3,4: 67MHz is Pin3/old or Pin4/new (with Pin3=NC/new or Pin4=GND/old)
Pin 43,45..49,100,101,125(A0!),201,203..206 are connected via 22 ohm.
Pin 77,80,81,83 are connected via 470 ohm.
Pin 82,84,89 are connected via 47 ohm.
Pin 95,96,97 are connected via 100 ohm.
Pin 44: goes LOW for a short time once every N us (guessed: maybe /REFRESH ?)
Pin 4: 67MHz (from IC204.pin5)
Pin 87/88: SPU-DMA related (/SPUW also permanent LOW for Manual SPU-RAM Write)
Pin 154: 33MHzS (via 22ohm and FB102 to SPU) (and TESTPOINT near MainRAM pin70)
Pin 160: DOTCLK (via 22ohm), and IC502.Pin41 (without 22ohm)
Pin 56,58 are maybe additional address lines for the addressable 8MB RAM.
The System Bus address lines are latched outputs (containing the most recently used / BIOS /EXP /SPU /CD address) (not affected by Main RAM and GPU addressing).

24.6 Pinouts - GPU Pinouts (for old 160-pin GPU)

Old 160-pin GPU is used on PU-7 boards and EARLY-PU-8 boards.

IC203 - Sony CXD8514Q - Old 160pin GPU for use with Dual-ported VRAM

Unlike the later 208pin GPU's, the old 160pin GPU has less supply pins, and, it doesn't have a 24bit RGB output (nor any other video output at all), instead, it's used with a RGB D/A converter that reads the video data directly from the Dual-ported VRAM chips (ie. from special RAM chips with two data busses, one bus for GPU read/write access, and one for the RGB video output).

Pin 1-63,148,160 = CPU Bus, Pin 66-139 = VRAM Bus (two chips, A and B), Pin 142-155 = Misc (CXA and RGB chips), Pin 18-19,156-157 = Test points.
Pin 3,5,6,11,98,99,102,103,108,148,160 via 22 ohm. Pin 104,105,106 via 100 ohm. Pin 107 via 220 ohm. Pin 155 via 2200 ohm. Pin 145 via 220+2200 ohm.

IC207 - SONY CXD2923AR - Digital VRAM to Analog RGB Converter (for old GPU)

This chip is used with the old 160pin GPU and two Dual-ported VRAM chips. The 2x16bit databus is capable of reading up to 32bits of VRAM data, and the chip does then extract the 15bit or 24bit RGB values from that data (depending on the GPU's current color depth).
The RGB outputs (pin 5,7,9) seem to be passed through transistors and capacitors... not sure how the capacitors could output constant voltage levels... unless the RGB signals are actually some kind of edge-triggering PWM pulses rather than real analog levels(?)

Pin 5,7,9 = RGB outputs (via transistors and capacitors?), Pin 18-22 = GPU, Pin 25-59 = VRAM (chip A and B), Pin 1-2,11-13,63 = Test points.

IC201 - 64pin NEC uPD482445LGW-A70-S or SEC KM4216Y256G-60 (VRAM 256Kx16)

IC202 - 64pin NEC uPD482445LGW-A70-S or SEC KM4216Y256G-60 (VRAM 256Kx16)

These are special Dual-ported VRAM chips (with two data busses), the D0-D15 pins are wired to the GPU (for read/write access), the Q0-Q15 pins are wired to the RGB D/A converter (for sequential video output).

The 8bit /LWE and /UWE write signals are shortcut with each other and wired to the GPU's 16bit /WE write signal.

IC501 24pin "SONY CXA1645M" Analog RGB to Composite (older boards only)

Used only on older boards (eg. PU-7, PU-8, PU-16), newer boards generate composite signal via 48pin IC502.
Pin7 (NPIN): NTSC=VCC, PAL=GND. Pin6 (SCIN aka FSC): Sub Carrier aka PAL/NTSC color clock, which can be derived from three different sources:

for the color clocks from GPU pins, the GPU does try to automatically generate PAL or NTSC clock depending on current frame rate, which is resulting in "wrong" color clock when chaning between 50Hz/60Hz mode).

24.7 Pinouts - GPU Pinouts (for new 208-pin GPU)

New 206-pin GPU is used LATE-PU-8 boards and up.

GPU Pinouts (IC203)

Pin 77..150 = Video RAM Bus. Pin 156..189 = Video Out Bus. Other = CPU Bus. Pin 153: Sub Carrier (NC on newer boards whick pick color clock from IC204).

GPU Pinout Notes

Pin 1,3,4,9,122..128,199,206 are connected via 22 ohm.
Pin 18 has a 4K7 ohm pullup to 3.5V
Pin 77..118 data lines (DQ0..DQ31) are connected via 82 ohm.
Pin 192/196: via 220 ohm to IC204.pin1 (53MHz)
At RAM Side: CKE via 4K7 to 3.5V, and, A8 is GROUNDED!
DQM0 is wired to both DQM0 and DQM2, DQM1 is wired to both DQM1 and DQM3.
CLK is wired to both GPU pin 131 and 134.
RGBnn = IC502 pin nn
/VSYNC, /HSYNC, (and BLANK?) are test points (not connected to any components).
/SYNC = (/VSYNC AND /HSYNC). BLANK = (VBLANK OR HBLANK).

IC202 44pin "Philips TDA8771H" Digital to Analog RGB (older boards only)

Region Japan+Europe: TDA8771AN
Region America+Asia: MC151854FLTEG or so?

Used only LATE-PU-8 boards (and PU-16, which does even have two TDA8771AH chips: one on the mainboard, and one on the VCD daughterboard).
Earlier boards are generating analog RGB via 64pin IC207, and later boards RGB via 48pin IC502.

IC502 48pin "SONY CXA2106R-T4" - 24bit RGB video D/A converter

Pin 3..8 (analogue outputs) are passed via external 75 ohm resistors.
Pin 6,7 additionally via 220uF. Pin 8 additionally via smaller capacitor.
Pin 10 (YTRAP) wired via 2K7 to 5.0V.
Pin 1,44,46,48 (can) connect via capacitors to ground (only installed for 44).
The 4.4MHz clock is obtained via 2K2 from IC204.Pin6.
The /PAL pin can be reportedly GROUNDED to force PAL colors in NTSC mode, when doing that, you may first want to disconnect the pin from the GPU.
Note: Rohm BH7240AKV has same pinout (XXX but with pin7/pin8 swapped?)

Beware

Measuring in the region near GPU Pin10 is the nocash number one source for blowing up components on the mainboard. If you want to measure that signals while power is on, better measure them at the CPU side.

24.8 Pinouts - SPU Pinouts

IC308 - SONY CXD2922Q (SPU) (on PU-7, EARLY-PU-8 boards)

IC308 - SONY CXD2925Q (SPU) (on LATE-PU-8, PU-16, PU-18, PU-20 boards)

Pin 1..36 = MIPS-CPU bus. Pin 45..87 = SPU-RAM bus (A0,A10-A15,/WE1,OE1=NC). Pin 91..99 = Digital serial audio in/out (A=CDROM, B=EXP, O=OUT).

IC732 - SONY CXD2941R (SPU+CDROM+SPU_RAM) (on PM-41(2) boards)

IC732 - SONY CXD2938Q (SPU+CDROM) (on newer boards) (PM-41 boards)

Pin 74..102 = SubCPU. Pin 103..144 = MainCPU. Pin 160..192 = Sound RAM Bus.
Pin 21 and 53..59 = Drive Motor Control (IC722).
Pin 1..47 are probably mainly CDROM related.
Pin 39 "TE9" = IC723.Pin16 - CL709, and via 15K to SPU.39
Pin 66 connects via 4K7 to IC723.Pin19.
Pin 67 not connected (but there's room for an optional capacitor or resistor)
The (tst) pins are wired to test points (but not connected to any components)

CXD2938Q SPU Pinout Notes

Pin 74,75,76,119,120 are connected via 22 ohm.
Pin 103,104 are connected via 100 ohm.
ZZnn = IC405 Pin nn (analog audio related, L/R/MUTE).
Pin 103..142 = System Bus (BIOS,CPU). Pin 160..192 = Sound RAM Bus.
Pin 178 used for both /CASL and /CASH (which are shortcut with each other).
Pin 146 and 151 are 3.48V (another supply, not 3.5V).

Pin 147 and 150 are connected via capacitors.
Pin 195 and 197 testpoints are found below of the pin 206/207 testpoints.

SPU.Pin5 connects to MANY modchips
SPU.Pin42 connects to ALL modchips
SPU.Pin42 via capacitor to SPU.Pin41, and via resistor?/diode? to IC723.10

CXD2938Q CDROM clocks

(*) these frequencies are twice as fast in double speed mode.

CXD2938Q CDROM signals

CXD2938Q CDROM/SPU Testpoints (on PM-41 board)

IC402 - 24pin AKM AK4309VM (or AK4309AVM/AK4310VM) - Serial 2x16bit DAC

Used only on older boards (eg. PU-8), newer boards seem to have the DAC in the 208pin SPU.
No 24pin AK4309VM datasheet exists (however it seems to be same as 20pin AK4309B's, with four extra NC pins at pin10-14).

IC405 - "2174, 1047C, JRC" or "3527, 0A68" (on newer boards)

Called "NJM2174" in service manual. Audio Amplifier with Mute.

Audio amplifier, for raising the signals to 5V levels.

IC405 - "NJM2100E (TE2)" Audio Amplifier (on older PU-8 and PU-22 boards)

24.9 Pinouts - DRV Pinouts

IC304 - 52pin/80pin - Motorola HC05 8bit CPU

Pinouts - HC05 Pinouts

IC305 - SONY CXD1815Q - CDROM Decoder/FIFO (used on PU-8, PU-16, PU-18)

Pin 1..20 to HC05 CPU, pin 22..42 to MIPS cpu, pin 43..75 to SRAM cd-buffer.
The pinouts/registers in CXD1199AQ datasheet are about 99% same as CXD1815Q.
Note: Parity on the 8bit data busses is NC. SRAM is 32Kx8 (A15+A16 are NC). Later boards have this integrated in the SPU.

ICsss - SONY CXA1782BR - CDROM Servo Amplifier (used on PU-8 boards)

Datasheet exists. Later boards have CXA1782BR+CXD2510Q integrated in CXD2545Q, and even later boards have it integrated in the SPU.

IC309 - SONY CXD2510Q - CDROM Signal Processor (used on PU-8, PU-16 boards)

Datasheet exists. Later boards have CXA1782BR+CXD2510Q integrated in CXD2545Q, and even later boards have it integrated in the SPU.

IC701 - SONY CXD2545Q - Signal Processor + Servo Amp (used on PU-18 boards)

Datasheet exists. The CXD2545Q combines the functionality of CXA1782BR+CXD2510Q from older boards (later boards have it integrated in the SPU). XTAI/XTAO input is 16.9344MHz (44.1kHz*180h), with XTSL=GND. Clock outputs are FSTO=16.9344MHz/3, FSOF=16.9344MHz/4, C16M=16.9344MHz/1.

IC101 - SONY CXD2515Q - Signal Processor + Servo Amp (used on DTL-H2010)

Pinouts are same as CXD2545Q, except, three pins are different: Pin24=ADII (instead of ADIO), Pin25=ADIO (instead of RFC), Pin68=C4M (instead of FSOF).

IC720 - 144pin SONY CXD1817R (=CXD2545Q+CXD1815Q) ;PU-20

IC701 - 8pin chip (on bottom side, but NOT installed) (PU-7 and EARLY-PU-8)

IC722 "BA5947FP" or "Panasonic AN8732SB" - IC for Compact Disc Players

Drive Motor related.

Additionally to the above 28pins, the chip has two large grounded pins (between pin 7/8 and 21/22) for shielding or cooling purposes.

IC703 - 20pin - "SONY CXA1791N" (RF Amplifier) (on PU-18 boards)

Datasheet for CXA1791N does exist. Later boards have IC703 replaced by IC723. Older PU-7/PU-8 boards appear to have used a bunch of smaller components (8pin chips and/or transistors) instead of 20pin RF amplifiers.

IC723 - 20pin - "SONY CXA2575N-T4" (RF (Matrix?) Amplifier) (PU-22..PM-41(2))

Used only on PU-22 .. PM-41(2) boards (PU-18 boards used IC703 "CXA1791N", and even older boards... maybe had this in CXA1782BR... or maybe had it in a bunch of 8pin NJMxxxx chips?).
There is no CXA2575N datasheet (but maybe some signals do resemble CXA2570N/CXA2571N/CXA1791N datasheets).

CN702 CDROM Data Signal socket (PU-23 and PM-41 board)

PU-23 and PM-41 board seem to be using exactly the same Drive, the only difference is the length (and folding) of the attached cable.

CN701 CDROM Motor socket (PU-8, PU-18, PU-23, PM-41 boards)

CLnnn - Calibration Points (PU-23 and PM-41 boards)

Probably test points for drive calibration or so.

24.10 Pinouts - VCD Pinouts

SCPH-5903 Video CD PlayStation

VCD Mainboard "PU-16, 1-655-191-11" Component List

The overall design is very close to LATE-PU-8 boards (1-658-467-2x). Changed components are IC102/IC304 (different kernel and cdrom firmware), C318/C325/C327 (height reduced capacitors for mounting the daughterboard above of them). Plus some extra components: Three triple multiplexors (for switching between PSX and VCD audio/video), and the daughterboard connector.

VCD Daughterboard "MP-45, 1-665-192-11" Component List

VCD Daughterboard Connector

IC104 "Sony CXD1852AQ" (MPEG-1 Decoder for Video CD) (120 pin)

The Hxxx pins are for the Host (the 8bit CXP CPU), the Mxxx for the RAM chips, the R/G/B pins are 24bit RGB video. Pin36 can be /CAS2 or MA9 (and, the VCD daughterboard has alternate solderpads for one large RAM instead of two small RAMs).

IC107 "6230FV" (OSD chip, similar to BU6257AFV-E2) (20 pin)

SIO pin1/2/3 are wired to CXP pin38/37/36. OSCIN is the RGB DAC CLK divided by two (from H74 chip pin5). OSD/SYNC on pin15-20 connect to the MPEG1 decoder chip.
No datasheet (but pinouts are same/similar as for BU6257AFV, documented in several service manuals for tape decks with vcd player: HCD-V5500, HCD-V8900/V8900AV, HCD-V909AV).

IC111 "Sony CXP10224-603R" (8bit SPC700 CPU) (64pin LQFP)

Pin 3-15,45,46,50 connect to MPEG1 decoder. Pin 36-38 to OSD. Pin 47-49 to HC05.PortF. Pin 27 is /RESET from PSU. Pin 29,31 are SUBQ from CXD2510Q. The "TP" pins connect to test points (but seem to be NC otherwise).
Pinouts are same as in CXP811P24 datasheet (which uses SPC700 instruction set; that instruction set is also used by SNES sound CPU).

IC109 "TLC2932" (PLL) (14pin)

Used to generate the CLK for the TDA chip (that is, the dotclk, paused during VSYNC, or so?). The same CLK, divided by two, is also used as OSD.OSCIN.

IC112 "74HCT32" (Quad OR gate) (14pin)

Used to sharpen the output from the PLL chip, and to level-shift signals for the two PLL inputs from 3.5V to 5V. The input-pairs for the OR gates are shortcut with each other, so the chip isn't actually ORing anything.

IC113 "H74 7H" (single D-type flip-flop; OSD clock divider) (8 pin)

Used to divide the RGB DAC CLK by two. CLK comes from TDA.pin31, D and /Q are shortcut with each other, /RES and /SET are wired to VDD, and Q goes to OSD.OSCIN.

ICnnn "4053C" (Triple multiplexor, for Audio LRCK,BCLK,DATA) (16pin)

The three SEL pins are wired to HC05.PortF3, the three SPU pins are wired via 10Kohm.

ICnnn "4053C" (Triple multiplexor, for Video FSC,CSYNC) (16pin)

The three SEL pins are wired to HC05.PortF3, the two OUTx pins are wired via 2.2Kohm.

ICnnn "NJM2283" (Triple multiplexor, for Video R,G,B) (16pin)

The three SEL pins are wired to HC05.PortF3, the six INxx pins wired through resistors and capacitors, the three OUTx pins are wired through capacitors.

24.11 Pinouts - HC05 Pinouts

Motorola HC05 chip versions for PSX cdrom control

The early DTL-H2000 devboard is also using a 80pin CPU (with piggyback EPROM socket), but that CPU is a Sony CXP82300 SPC700 CPU, not a Motorola HC05 CPU.

IC304 - "C 3060, SC430943PB, G63C 185" (PAL/PSone) - CDROM Controller

Called "MC68HC05G6PB" in service manual (=8bit CPU).

This chip isn't connected directly to the CPU, but rather to a Fifo Interface, which is then forwarding data to/from the CPU. On older PSX boards, that Fifo Interface has been located in a separate chip, on newer PSX boards and PSone boards, the Fifo stuff is contained in the SPU chip. The CDROM has a 32K buffer, which is also implemeted at the Fifo Interface side.

OSC input (internally HC05 is running at OSC/2, ie. around 2MHz):

HC05 - 80pin version (pinout from MC68HC05L16 datasheet)

HC05 - 32pin/64pin Versions

Sony's Digital Joypad and Mouse contain 32pin CPUs, which are probably also HC05's:
Pinouts - Component List and Chipset Pin-Outs for Digital Joypad, SCPH-1080
Moreover, some old memory cards contain a 64pin Motorola SC419510FU (probably also a HC05) with separate Atmel AT29LV010A (128Kx8 FLASH).

24.12 Pinouts - MEM Pinouts

IC102 - BIOS ROM (32pin, 512Kx8, used on LATE-PU-8 boards, and newer boards)

Uses standard EPROM pinouts, VCC is 3.5V though, when replacing the ROM by an EPROM, it may be required to replace the supply by 5V. Note that, on PM-41 boards at least, Pin 1 is connected to A19 (allowing to install a 1MB BIOS chip on that board, however, normally, a 512KB BIOS chip is installed, and, the CPU is generating an exception when trying to access more than 512KB, but that 512K limit can be disabled via memory control registers).
Datasheet for (MS-)M534031E does exist.

IC102 - BIOS ROM (40pin, 512Kx8, used on PU-7 boards, and EARLY-PU-8 boards)

The chip supports 8bit/16bit mode, on the PSX D0-D14 are actually wired, but A0/D15 is wired to A0, and /BYTE is wired to GND, so 16bit mode doesn't work.
Datasheet for MX23L4100 does exist.

IC102 - BIOS ROM (44pin, 1Mx8, used on P16-boards, ie. VCD console)

Pinouts are from OKI MSM538032E datasheet.

CPU-RAM (four 28pin chips) (older boards)

Unknown.
Note: The newer 70pin RAM comes up without external /REFRESH signal, but maybe the 28pin RAMs required refresh (the CPU has some odd delays once and when).

IC106 - CPU-RAM (single 70pin chip, on newer boards)

"Samsung K4Q153212M-JC60" (70pin, 512Kx32) (newer boards)
"Toshiba T7X16" (70pin, 512Kx32) (newer boards, too)

Notes: Pin23 must NC or VSS. In the PSone, /OE is wired to GND.
Datasheet for K4Q153212M-JC60 does exist (the chip supports 27ns Hyper Page mode access, which seems to be used for DMA).

IC106/IC107/IC108/IC109 - CPU-RAM (four 28pin chips, on PU-8, PU-18 boards)

SEC KM48V514BJ-6 (DRAM 512Kx8) (four pieces = 512Kx32 = 2Mbyte)

Datasheet for KM48V514B-6 and BL-6 exist (though none for BJ-6). The chips support 25ns Hyper Page mode access.

IC310 - SPU-RAM (512Kbyte)

EliteMT M11B416256A-35J (256K x 16bit) (40pin SOJ, PM-41 boards)
Nippon Steel NN514256ALTT-50 (256K x 16bit) (40pin TSOP-II, PU-23 boards)
Toshiba TC51V4260DJ-70 (40pin, PU-8 board) (PseudoSRAM)

Note: SPU-RAM supply can be 3.5V (PU-8), or 5.0V (PU-22 and PM-41).
Note: The /CASL and /CASH pins are shortcut with each other on the mainboard, both wired to the /CAS pin of the SPU (ie. always accessing 16bit data at once).
Note: The TSOP-II package (18mm length, super-flat and with spacing between pin 10/11 and 30/31) is used on PU-23 boards. The pinouts and connections are identical for SOJ and TSOP-II.
Note: Nippon Steels NN514256-series is normally 256Kx4bit, nethertheless, for some bizarre reason, their 256Kx16bit chip is marked "NN514256ALTT"... maybe that happened accidently in the manufacturing process.
Note: The PM-41(2) board has on-chip RAM in the SPU (no external memory chip).

IC303 - CDROM Buffer (32Kbyte)

"HM62W256LFP-7T" (SRAM 32Kx8) (PCB bottom side) (PU-8)
"SONY CXK5V8257BTM" 32Kx8 SRAM (PU-18)

Used only on older boards (eg. PU-8, PU-18), newer boards seem to have that RAM included in the 208pin SPU chip.

IC201 - GPU-RAM (1MByte) (or 2MByte, of which, only 1MByte is used though)

Samsung KM4132G271BQ-10 (128K x 32bit x 2 Banks, Synchronous Graphic RAM) 1MB
Samsung K4G163222A-PC70 (256K x 32bit x 2 Banks, Synchronous Graphic RAM) 2MB

Newer boards often have 2MB VRAM installed (of which only 1MB is used, apparently the 2MB chips became cheaper than the 1MB chips). At the chip side, the only difference is that Pin30 became an additional address line (that, called A8, and, accordingly, the old A8,A9 pins were renamed to A9,A10). At the mainboard side, the connection is exactly the same for both 1MB and 2MB chips; Pin30 is grounded on both PU-23 boards (which typically have 1MB) and PM-41 boards (which typically have 2MB).
Note: The PM-41(2) board has on-chip RAM in the GPU (no external memory chip).

24.13 Pinouts - CLK Pinouts

The "should-be" CPU clock is 33.868800 Hz (ie. the 44100Hz CDROM/Audio clock, multiplied by 300h). However, the different PSX/PSone boards are using different oscillators, multipliers and dividers, which aren't exactly reaching that "should-be" value.
The PSone are using a single oscillator for producing CPU/GPU clocks, and for producing the TV/color signal:

PSone/PAL - IC204 8pin - "CY2081, SL-509" or "2294A, 1913"

Clock Multiplier/Divider

PSone/NTSC - IC204 8pin "CY2081 SL-500" (PSone, and PSX/PU-20 and up)

Unknown. Uses a 14.318MHz oscillator, so multiply/divide factors must be somehow different.

The "optimal" conversion would be (hardware is barely able to do that):

So, maybe it's doing

PSX/PAL

PU-7 and PU-8 boards are using three separate oscillators:

PU-18 does have same X101/X201 as above, but doesn't seem to have X302.

PSX/NTSC

PU-7 and PU-8 boards are using three separate oscillators:

PU-20 works more like PSone (a single oscillator, and CY2081 SL-500 divider)

24.14 Pinouts - PWR Pinouts

Voltage Summary

Fuses

There are a lot of SMD elements marked FBnnn, these are NOT fuses (at least they don't seem to blow-up whatever you do). The actual fuses are marked PSnnn, found near the power switch and near the power socket.

IC601 3pin +5.0V "78M05, RZ125, (ON)"

IC602 - Audio/CDROM Supply

Called "LP29851MX-3.5" in service manual.

IC002/IC003 - Reset Generator (PM-41 board)

/RES is connected via 330 ohm to GPU/CPU, and via 5K6 SPU/IC722/IC304.
Note: Either IC002 or IC003/Q004 can be installed on PM-41 boards. Most or all boards seem to contain IC003/Q004.
Note: PSX consoles have something similar on the Power Supply boards (IC101: M51957B).

IC606/IC607 - TL594CD - Pulse-Width-Modulation Power-Control Chip

Q602

CN602 - PU-8, PU-9 board Power Socket (to internal power supply board)

Purpose of the standy-by voltage is unknown... maybe to expansion port?

CN602 - PU-18, PU-23 board Power Socket (to internal power supply board)

CN102 - Controller/memory card daughter-board connector (PU-23 board)

24.15 Pinouts - Component List and Chipset Pin-Outs for Digital Joypad, SCPH-1080

Digital Joypad Component List (SCPH-1080)

Digital Joypad Connection Cable:

Digital Joypad 32pin SC401800 Chip Pin-Outs

Digital Joypad 14pin BA10339F Chip Pin-Outs

24.16 Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-1150

This applies for two controller versions:

Both are using the same PCB, and the same SD657 chip. The difference is that the motor, transistors, and some resistors aren't installed in SCPH-1180.

Analog Joypad Component List (SCPH-1150, single motor)

Analog Joypad Connection Cables (SCPH-1150)

CN1 (cable to PSX controller port) (same for SCPH-1150 and SCPH-1200)

CN2 (ribbon cable to analog-joystick daughterboard) (SCPH-1150)

J3 (ribbon cable to R-1, R-2 button daughterboard) (SCPH-1150)

J2 (ribbon cable to L-1, L-2 button daughterboard) (SCPH-1150)

J1 wires to small rumble motor (SCPH-1150)

Analog Joypad Chipset Pin-Outs (SCPH-1150)

U1 42pin "SD657, 9702K3006"

U2 (probably reset signal related)

Q1 "BQ03" or so (motor post-amp)

Q2 "S6","SG","9S" or so (motor pre-amp)

Motor

Left/Single Motor (SCPH-1150)

24.17 Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-1200

Analog Joypad Component List (SCPH-1200, two motors)

Note: There's also a different SCPH-1200 revision, which having a smaller mainboard with analog joysticksonboard, plus a single sided PCB for the digital buttons (that is, similar to SCPH-110, but with the single sided PCB instead of membrane foil).

Analog Joypad Connection Cables (SCPH-1200)

CN1 (cable to PSX controller port) (same for SCPH-1150 and SCPH-1200)

CN2 (ribbon cable to analog-joystick daughterboard) (SCPH-1200)

J1 (ribbon cable to R-1, R-2 button daughterboard) (SCPH-1200)

J2 (ribbon cable to L-1, L-2 button daughterboard) (SCPH-1200)

M1 wires to big rumble motor (SCPH-1200)

M2 wires to small rumble motor (SCPH-1200)

Analog Joypad Chipset Pin-Outs (SCPH-1200)

U1 SONY CXD8771Q

U2 PST9329 (System Reset with 2.9V detection voltage)

U3 NJM2904 (Dual Operational Amplifier)

Q1 (transistor for big M1 motor)

Q2 (transistor for small M2 motor)

Motors

24.18 Pinouts - Component List and Chipset Pin-Outs for Analog Joypad, SCPH-110

Analog Joypad Component List (SCPH-110, two motors, PSone-design)

Analog Joypad Connection Cables (SCPH-110)

CN1 (cable to PSX controller port)

J1 (ribbon cable with membrane/foil with digital buttons)

M1 wires to left/big rumble motor (SCPH-110)

M2 wires to right/small rumble motor (SCPH-110)

U1 ("SD707, 039 107")

Misc

```
VR1..VR4 -- analog inputs
R1..R5 -- signals to/from psx
R6 ?
R7 M1
R8
R9
R10
JP1
C1 3.5V to GND3 (22uF)
C2 3.5V to GND3 (U1)
C3 VR1 to GND3
C4 VR2 to GND3
C5 VR3 to GND3
C6 VR4 to GND3
C7 M2+ to M2-
```

C8 M1+ to M1-
C9 M1 related
S5
S6

Motors

D1 is probably a Z-diode with R7 as pull-up, creating a reference/source voltage at U1.13 for the analog output at U1.12.

24.19 Pinouts - Component List and Chipset Pin-Outs for Namco Lightgun, NPC-103

Schematic

http://www.nicolaselectronics.be/reverse-engineering-the-playstation-g-con45/

Namco Lightgun "NPC-103, (C) 1996 NAMCO LTD." Component List

PCB "DNP-0500A, NPC10300, namco, CMK-P3X"

PCB "DN-P-0501"

PCB "DN-P-0502"

Other Components

Cable Pinouts

U1 "NAMCO103P" Pinouts (44pin, arranged as 4x11pin)

U2 "7071" Pinouts (=BA7071F, Sync Separator IC with AFC) (2x4pin)

24.20 Pinouts - Component List and Chipset Pin-Outs for Multitap, SCPH-1070

Multitap Component List

Cables from Multitap PCB1 to PCB2:

plus a bunch of SMD capacitors and around 70 SMD resistors.

Multitap PSX Controller Port Cable

Multitap CARD A/B/C/D Slots

Multitap JOY A/B/C/D Slots

Multitap IC02 8pin "7W14, 5K" some tiny SMD chip

Multitap "SONY CXD103-166Q" Chip Pin-Outs (Multitap CPU)

24.21 Pinouts - Memory Cards

Sony Playstation Memory Card (SCPH-1020)

The "SONY CXD8732AQ" chip is installed on memory cards with "SPC02K1020B" boards, however, the text layer on the board says that it's an "LC86F8604A" chip. So, the CXD8732AQ is most probably a standard LC86F8604A chip (more on that below) with a Sony Memory Card BIOS ROM on it.
The "SONY CXD8732AQ" comes in a huge 64pin package, but it connects only to:

Aside from that chip, the board additionally contains some resistors, capacitors, z-diodes (for protection against too high voltages), a 6MHz oscillator (for the CPU), and a 5pin

reset generator (on the cart edge connector, the supply pins are slightly longer than the data signal pins, so when inserting the cartridge, power/reset gets triggered first; the 7.5V supply pin is left unconnected, only 3.5V are used).
Caution: The "diagonal edge" at the upper-left of the CXD8732AQ chip is Pin 49 (not pin 1), following the pin numbers on the board (and the Sanyo datasheet pinouts), pin 1 is at the lower-left.

Sanyo LC86F8604A

8bit CPU with 132Kbyte EEPROM, 4Kbyte ROM, 256 bytes RAM, 2 timers, serial port, and general purpose parallel ports. The 132K EEPROM is broken into 128K plus 4K, the 4K might be internally used by the CPU, presumably containing the BIOS (not too sure if it's really containing 4K EEPROM plus 4K ROM, or if it's meant to be only either one).

Ports P10..P17 have multiple functions (I/O port, data bus, serial, etc):

In March 1998, Sanyo has originally announced the LC86F8604A as an 8bit CPU with "2.8V FLASH, achieved for the first time in the industry", however, according to their datasheet, what they have finally produced is an 8bit CPU with "3.5V EEPROM". Although, maybe the 3.5V EEPROM version came first, and the 2.8V FLASH was announced to be a later low-power version of the old chip; namely, otherwise, it'd be everyones guess what kind of memory Sony used in memory cards before 1998?

Note

For the actual pin-outs of the cart-edge connector, see
Pinouts - Controller Ports and Memory-Card Ports

24.22 Mods - Nocash PSX-XBOO Upload

Nocash PSX-XBOO Connection (required)

Nocash PSX-BIOS Connection (required)

Nocash BIOS "Modchip" Feature (optional)

The nocash PSX bios outputs the "data" signal on the A20 address line, so (aside from the BIOS chip) one only needs to install a 1N4148 diode and two wires to unlock the CDROM. For more variants, see:
CDROM Protection - Chipless Modchips

Composite NTSC/PAL Mod (optional)

Mods - PAL/NTSC Color Mods

Component List

PSX-XBOO Upload BIOS

The required BIOS is contained in no$psx (built-in in the no$psx.exe file), the Utility menu contains a function for creating a standalone ROM-image (file PSX-XBOO.ROM in no$psx folder; which can be then burned to FLASH or EPROM).

Pinouts

Note

Instead of the above internal mod, the nocash kernel clone can be also installed on cheat devices, which do also include DB25 connectors for parallel port uploads, too. For DB25 parallel port uploads, do the following mods to the cheat device:

24.23 Mods - PAL/NTSC Color Mods

The PSX hardware is more or less capable of generating both PAL and NTSC signals. However, it's having the bad habbit to do this automatically depending on the game's frame rate. And worse, it's doing it regardless of whether the board is having matching oscillators installed (eg. a PAL board in 60Hz mode will produce NTSC encoding with faulty NTSC color clock).

RGB Cables

RGB cables don't rely on composite PAL/NTSC color encoding, and thus don't need any color mods (except, see the caution on GNDed pins for missing 53.20MHz/53.69MHz oscillators below).

Newer Consoles (PU-22, PU-23, PM-41, PM-41(2))

These consoles have 17.734MHz (PAL) or 14.318MHz (NTSC) oscillators with constant dividers, so the color clock will be always constant, and one does only need to change the color encoding:

This forces the console to be always producing the desired composite color format (regardless of whether the GPU is in 50Hz or 60Hz mode).
That works for NTSC games on PAL consoles (and vice-versa). However, it won't work for NTSC consoles with PAL TV Sets (for that case it'd be easiest to install an extra oscillator, as done on older consoles).

Older Consoles (PU-7, PU-8, PU-16, PU-18, PU-20)

These consoles have 53.20MHz (PAL) or 53.69MHz (NTSC) oscillators and the GPU does try to change the clock divider depending on the frame rate (thereby producing a nonsense clock signal that's neither PAL nor NTSC). Best workaround is to install an extra 4.43361875MHz (PAL) or 3.579545MHz (NTSC) oscillator (with internal amplifier, ie. in 4pin package, which resembles DIP14, hence the pin 1,7,8,14 numbering):

Caution: Many mainboards have solder pads for both 53.20MHz and 53.69MHz oscillators, the missing oscillator is either GNDed or shortcut with the installed oscillator (varies from board to board, usually via 0 ohm resistors on PCB bottom side). If it's GNDed, remove that connection, and instead have it shortcut with the installed oscillator.
Alternately, instead of the above mods, one could also install the missing oscillator (and remove its 0 ohm resistor), so the board will have both 53.20MHz and 53.69MHz installed; that will produce perfect PAL and NTSC signals in 50Hz and 60Hz mode accordingly, but works only if the TV Set recognizes both PAL and NTSC signals.

Notes

External 4.433MHz/3.579MHz osciallors won't be synchronized with the GPU frame rate (normally you don't want them to be synchronized, but there's some small risk that they might get close to running in sync, which could result in static or crawling color artifacts).
For the CXA1645 chip modded to a different console region, one should also change one of the resistors (see datasheet), there's no noticable difference on the TV picture though.

Region Checks

Some kernel versions contain regions checks (additionally to the SCEx check), particulary for preventing NTSC games to run on PAL consoles, or non-japanese games on japanese consoles. Some PAL modchips can bypass that check (by patching the region byte in BIOS). Expansions ROMs or nocash kernel clone could be also used to avoid such checks.

25. About & Credits

Credits

GPU.TXT by doomed/padua; based on info from K-communications & Nagra/Blackbag
GTE.TXT by doomed@c64.org / psx.rules.org
SPU.TXT by doomed@c64.org / psx.rules.org
CDINFO.TXT by doomed with big thanks to Barubary, who rewrote a large part
SYSTEM.TXT by doomed with thanx to Herozero for breakpoint info
PS_ENG.TXT PlayStation PAD/Memory Interface Protocol by HFB03536
IDT79R3041 Hardware User's Manual by Integrated Device Technology, Inc.
IDTR3051, R3052 RISController User's Manual by Integrated Device Technology
PSX.* by Joshua Walker (additional details in various distorted file formats)
LIBMIRAGE by Rok; info/source code for various cdrom-image formats
psxdev.ru; cdrom sub-cpu decapping

PSXSPX homepage

http://problemkaputt.de/psx.htm no$psx emulator/debugger
http://problemkaputt.de/psx-spx.htm psx specs in html formal
http://problemkaputt.de/psx-spx.txt psx specs in text formal

Contact

http://problemkaputt.de/email.htm (spam-shielded)

25.1 Index

[extracted from no$psx v2.0 documentation]

26. CDROM Video CDs (VCD)

VCDs are Video CDs with MPEG compression, yielding a playtime of 72 minutes per disc (whole movies usually being stored on two CDs). VCDs are popular in asia (as opposed to VHS tapes used in western world).

VCDs on Playstation

For the Playstation, the asian SCPH-5903 model includes a special daughterboard with MPEG decoding hardware for playing VCDs.
CDROM - Video CD Commands
Pinouts - VCD Pinouts
Without that hardware it has been widely believed to be impossible to play VCDs on Playstations, although, as of 2017, it turned out that the Playstation's CPU and MDEC decoder are fast enough for that purpose (when skipping B-frames, rendering the movie in monochrome without colors, and reducing audio output to 11kHz/mono).

ISO Filesystem (Track 1)

VCD ISO Basic Files (INFO, ENTRIES, AVSEQnn, ISO Filesystem)
VCD ISO Playback Control PBC Files (PSD, LOT, ITEMnnnn)
VCD ISO Search Files (SCANDATA, SEARCH, TRACKS, SPICONTX)
VCD ISO Misc files (CAPTnn, AUDIOnn, KARINFO, PICTURES, CDI)

MPEG Streams (Track 2 and up)

VCD MPEG-1 Multiplex Stream
VCD MPEG-1 Video Stream
XXX MPEG-1 Macroblocks
VCD MP2 Audio Stream

VCD Versions & Variants

XXX

26.1 VCD ISO Basic Files (INFO, ENTRIES, AVSEQnn, ISO Filesystem)

Primary Volume Descriptor (00:02:16)

VCDs are having a standard ISO Primary Volume Descriptor, with some VCD specific entries:

There are some more differences to normal CDROMs:

Due to the fixed sector numbers, VCDs players can completely ignore the ISO filesystem with filenames and folders, and just address everything via sector numbers (though accessing files in EXT and CDI folders seem to require using the filesystem).

VCD\INFO.VCD or SVCD\INFO.SVD (00:04:00) (800h bytes, one sector)

InfoStatusFlags at [02Bh] describes certain characteristics of the disc:

Note: Bit5/6 are used only if the next disc has the same Album ID (eg. the feature allows to skip copyright messages if the same message was already shown on another disc).
First_segment_addr: The location of the first sector of the Segment Play Item Area [that is... the first ITEMnnnn.DAT file?], in the form mm:ss:00. Must be 00:00:00 if PSD size is zero. If PSD size is nonzero, but no segments used: Usually set to 00:02:00.

VCD\ENTRIES.VCD or SVCD\ENTRIES.SVD (00:04:01) (800h bytes, one sector)

Version;

Sys_prof_tag;

MPEGAV\AVSEQnn.DAT (pointers to max 98 MPEG-1 Tracks, nn=01..98) (for VCDs)

MPEG2\AVSEQnn.MPG (pointers to max 98 MPEG-2 Tracks, nn=01..98) (for SVCDs)

MPEGAV\AVSEQnn.MPG (pointers to WHATEVER) (as so on some SVCDs or VCD30?)

These filesystem entries contain pointers to the video tracks (that is, outside of the ISO area on Track 1).
Commercially made SVCDs can reportedly contain 7 folders: Autorun, Data, Ext, Mpegav, Segment, Svcd and Vmp (ie. there's no MPEG2 folder on all SVCDs? though that MPEGAV folder is said to contain a .MPG file instead of .DAT file).

26.2 VCD ISO Playback Control PBC Files (PSD, LOT, ITEMnnnn)

Playback Control (PBC) is an optional feature that allows to define menues, pictures or text pages (whereas all those is internally just consisting of MPEG compressed bitmaps; rather than of text characters).
Presence of the PBC feature is indicated by PSD.VCD filesize entry (in INFO.VCD) being nonzero. PBC seems to be supported by most VCDs (except older discs from around 1997), however, many VCDs are merely including a single PlayList entry for the movie track, without any further menues/extras.

VCD\PSD.VCD or SVCD\PSD.SVD (00:04:34 and up) (max 256 sectors)

The Descriptors in this file can be considered as being "program code". The program is usually stuck on "executing" the current descriptor (eg. playing a movie, or showing a selection menu) without automatically increasing the program counter. Actual program flow occurs only if the user presses a button (or upon selection timeouts), causing the program to "goto" to a new PsdOffset. And, something does probably happen upon end-of-track/item... maybe that does automatically trigger the Next button handler?

Delay values in "1s/10s" units (for PlayList[0Ch,0Dh]):

Item numbers (0..599 or 1000..2979) can be:

PsdOffset values can be:

For whatever reason, some PsdOffsets are specified as ListID (lid), these ListID values must be translated to actual PsdOffset via the ListID Offset Table (aka LOT.VCD/LOT.SVD file).

VCD\LOT.VCD or SVCD\LOT.SVD (00:04:02..33) (64Kbyte, 32 sectors)

The ListID Offset Table (LOT) allows to translate ListIDs to PsdOffsets. The file is always 64Kbyte in size (unused entries should be set to FFFFh).
The PSD.VCD file does also assign ListIDs to each descriptor (ie. instead of using the LOT.VCD file, one could also scan all descriptors in PSD.VCD when searching a specific ListID).

Note: ListID#1 is used as entrypoint to PSD.VCD when inserting a new disc (or when inserting another disc of the SAME movie, the entrypoint can be ListID#2, depending on the Next Disc flag in INFO.VCD).

SEGMENT\ITEMnnnn.DAT (Pictures, Menu screens) (nnnn=0001..1980)

These files contain Pictures/Menu screens referenced from PSD.VCD. The files seem to be stored in FORM2 sectors (not FORM1). Unknown if the files are located on Track 1. The content of the files seems to resemble short MPEG video clips (with only one picture frame, or eventually with a few frames for short animations, including audio in some cases). Still images are said to be allowed to use twice the resolution of MPEG videos.

EXT\PSD_X.VCD or EXT\PSD_X.SVD (extended version of PSD.VCD)

EXT\LOT_X.VCD or EXT\LOT_X.SVD (extended version of LOT.VCD)

The "extended" files are often identical to the normal PSD/LOT files. The difference is that, if disc uses SelectionLists, then PSD should use the normal descriptor (18h), and PSD_X should use the extended descriptor (1Ah), the latter one seems to be intended to allow to highlight the current menu selection (particulary useful when using +/- buttons instead of Numeric Keypad input). Note: Nethertheless, Muppets from Space uses descriptor 18h in PSD_X.
Unknown if SVCDs do really have "extended" files, too (theoretically the VCD extension should be a default feature for SVCDs).

Playback Control Issues

Although PBC was intended as "nice extra feature", many VCDs are containing faulty PSD files. In general, VCD players should either leave PBC unsupported (or provide an option for disabling it).
Red Dragon from 2003 uses extended selection lists, but crops PSD_X.VCD to the same filesize as PSD.VCD.
Muppets from Space from 1999 assigns weird functions to Prev/Next buttons (Next wraps from Last Track to First Track, but Prev doesn't wrap from First to Last; default Non-PBC Prev/Next functions are more user friendly).
Sony's SCPH-5903 console refuses to display the HH:MM:SS playback time when using PBC (instead it does only display a "PBC" logo).

26.3 VCD ISO Search Files (SCANDATA, SEARCH, TRACKS, SPICONTX)

Below files can help searching I-frames, and provide some info about the content of Tracks and Segments.
Essentially, searching I-frames is possible without these files - however, if present, then the files may be useful in two cases: For discs with variable bitrates (which isn't allowed on VCDs though), and, for CDROM firmwares that don't support "inaccurate" seeking (like telling it to start reading anywhere NEAR some MM:SS:FF value, so one could skip sectors till reaching an I-frame) (ie. if the firmware insists on a "accurate" seek position, then it's best to give it a known I-frame address).

Caution: Overlapping Sectors (!?!)

Reportedly the new SVCD files TRACKS.SVD and SEARCH.DAT are on these sectors:

If that's correct, then the files would overlap with PSD.SVD (when PSD.SVD is bigger than one sector), that would be weird, but possible (ie. the "PsdOffset" in PSD.SVD would need to "skip" the region used by those two files).

EXT\SCANDATA.DAT (12+3*N bytes for VCD 2.0) (or 16+3*N+2*X+3*Y+3*Z for SVCD)

This file fulfills much the same purpose of the SEARCH.DAT file except that this file is mandatory only if the System Profile Tag of the INFO.SVD file is 0x01 (HQ-VCD) and also that it contains sector addresses also for each video Segment Play Items in addition to the regular MPEG tracks.

SVCD\SEARCH.DAT (13+3*N bytes)

This file defines where the scan points are. It covers all mpeg tracks together. A scan point at time T is the nearest I-picture in the MPEG stream to the given time T. Scan points are given at every half-second for the entire duration of the disc.

Note: This SVCD file is about same as the old EXT\SCANDATA.DAT file on VCDs (with one extra entry for Time Interval). Whilst, SVCDs are storing some different stuff in EXT\SCANDATA.DAT (despite of the identical filename).

SVCD\TRACKS.SVD (11+4*N bytes) (or rarely:11+5*N bytes)

The TRACKS.SVD file contains a series of structures, one for each track, which indicates the track's playing time (in sectors, not actually real time) and contents. SVCD\TRACKS.SVD is a mandatory file which describes the numbers and types of MPEG tracks on the disc.

SVCD\SPICONTX.SVD (1000h bytes, two sectors)

Unknown if/when/where/why this file exists, possibly only on VCD30?
Note: The same info can be stored in INFO.SVD at offsets [038h..7F3h].

Content Flags for Segments and Tracks

For SVCD\INFO.SVD and SVCD\TRACKS.SVD (on SVCD) these are encoded in 1 byte:

For SPICONTX.SVD and SVCD\TRACKS.SVD (on VCD30) these are encoded in 2 bytes:

26.4 VCD ISO Misc files (CAPTnn, AUDIOnn, KARINFO, PICTURES, CDI)

EXT\CAPTnn.DAT (Closed Caption data, aka subtitles) (SVCD only?)

VCDs with subtitles are usually/always having the subtitles encoded directly in the picture frames (ie. in the MPEG macroblocks, rather than using the Closed Caption

feature).
These CAPTnn.DAT files are intended for Closed Captions (eg. subtitles in different languages and/or for deaf people).
Alternately, the "user_data_cc" flag in INFO.VCD?/INFO.SVD can indicate to store Closed Captions in MPEG User Data (with START_CODE=000001B2h=User Data) instead of in EXT\CAPTnn.DAT. Either way, the format of those Closed Captions is unknown.
Moreover, Content can be flagged to have Overlay Graphics/Text (OGT), whatever that is: it might be related to Closed Captions.
Note: Reportedly CAPTnn.DAT can exist on VCDs and SVCDs (although the same person reported that VCDs do not support subtitles, so that info sounds wrong).

CDDA\AUDIOnn.DAT (pointers to uncompressed CD Audio Tracks)

These filesystem entries contain pointers to uncompressed audio tracks tracks (that is, outside of the ISO area on Track 1).
Most VCDs don't have audio tracks (though some VCDs do contain empty CDDA folders).
Maybe the feature is occassionally used the other way around: Music discs containing VCD clips as bonus feature?

KARAOKE\KARINFO.xxx (whatever)

The KARAOKE folder exists on many VCDs (about 50%), but it's usually/always empty on all discs.
Reportedly the folder can contain "KARINFO.xxx" files, but the purpose/format of that files is unknown.
Reportedly there are Midi VCDs (MVCDs) for karaoke, maybe those discs have "KARINFO.xxx" files(?)

PICTURES*.* (whatever)

Unknown purpose. The PICTURES folder has been spotted on one VCD (Wallace and Gromit), but the folder was just empty.

CDI*.* (some kind of GUI/driver for Philips CDI Players)

The CDI folder is some relict for Philips CDI Players, it isn't used by normal VCD players, however, the CDI folder & files are included on most or all VCDs.
The path/name for the CDI executable is stored at offset 23Eh in the ISO Primary Volume Descriptor (usually "CDI/CDI_APPL.VCD;1" or "CDI/CDI_VCD.APP;1") (or accidentally "CDI_CDI_VCD.APP;1" on homebrew Nero discs).

The files in the CDI folder are usually just some standard files (without any customizations), however, there are some different revisions of these files:

CDI_VCD.CFG is some ASCII text file (with uncommon 0Dh,0Dh,0Ah line breaks), the file could be customized to change things like GUI colors, but most or all discs seem to contain the same file with CRC32=D1C6F7ADh. Note: The CFG file is missing on the homebrew DemoVCD.
CDI_IMAG.RTF is seen as 1510028 byte file under windows (that is, with a windows RIFF header, and with data area containing the whole 930h bytes from each sector; this includes the MM:SS:FF values from the sector header, so the RTF file may look slightly different depending on which sectors it has been stored on, although the files are usually exactly same apart from those MM:SS:FF values). Note: The RTF file is cropped to 1324220 bytes (instead of 1510028) on the homebrew DemoVCD (apart from that, the file is same as normal).
CDI_ALL.RTF and CDI_BUM.RTF cannot be read/copied under Windows 7 (which is weirdly reporting them to use an "invalid MS-DOS function"; some people also reported having CDI_IMAG.RTF files with similar problems). The reason is unknown, maybe windows doesn't fully support the CD filesystem, or some VCDs are violating the filesystem specs, or whatever... maybe windows is mis-identifying certain RTF files as Rich Text Format files and tries to prevent virus-infections by throwing a faked "MS-DOS" error message.

26.5 VCD MPEG-1 Multiplex Stream

Multiplex Stream & Sector Boundaries

The Multiplex stream is some higher level stream, intended to help to distinguish between Audio- and Video-streams (which are enclosed in the Multiplex stream). MPEG's are somewhat organized in "sectors", with sector size varying for normal .mpg files and VCDs:

Sectors are always beginning with a Multiplex Packet (and Multiplex Packets are never crossing sector boundaries). If the amount of video data exceeds the sector size, then it's split into several Multiplex packets, whereas, that may happen anywhere in the video stream (ie. there can be Multiplex Headers occurring even in the middle of Video packet).

MPEG-1 Multiplex Pack (sector header) (12 bytes)

The Pack Header is found at the begin of the stream (on VCDs, it's also found at the begin of each sector). The SCR values might help on identifying the current playback position, and, with the bitrate value, this could be also used to compute the distance to another position (though there are other ways to determine the position/bitrate, so the Pack is kinda useless).

MPEG-1 Multiplex System Header (12+N*3 bytes)(optionally)(at start of stream)

The System Header is usally found after the first Pack at the begin of the stream.

Followed by N*3 bytes for the streams (each with first bit=set):

Terminated by a value with first bit=cleared (eg. next 000001xxh value).

MPEG-1 Multiplex Video/Audio/Special Packets (7..24 bytes, plus data)

These packets are encapsulating the lower-level Video/Audio streams.

If (and while) next two bits are 11b (0..16 padding bytes):

If next two bits are 01b (buffer size info):

Always:

If PTS Flag set:

If DTS Flag set (in this case PTS Flag must be also set):

If PTS and DTS Flags are both zero:

Always:

Note: The first Multiplex Video Packet would usually start with a Sequence Header Code (000001B3h), and the first Multiplex Audio Packet should always start with an Audio Sync Word (FFFh).
However, the size of the Multiplex packets does usually differ from the size of the packets in the audio/video stream, so new Multiplex Packets may occur anywhere in the middle of those streams (eg. in the middle of a video slice, the next Multiplex Video packet would then begin with the remaining slice bytes, rather than with a 000001xxh code; it's also possible that a Multiplex Audio packet gets inserted in the middle of the video slice).

The best (or easiest) way to get continous data for the lower level streams might be to memcopy the data from Multiplex packets to separate Audio & Video buffers.

MPEG-1 Multiplex End Code (4 bytes)

This should occur at the end of the video. On a VCD it does also occur at the end of each video track.

26.6 VCD MPEG-1 Video Stream

The Video stream is part of the Multiplex stream, meaning that the Video packets preceeded (and interrupted) by Multiplex headers. Ie. before processing the Video packets, one must first extract the video snippets from the Multiplex stream (see previous chapter).

MPEG-1 Video Sequence Header (12, 76, or 140 bytes, ie. 12+N*64)

Next 64byte only when above bit was set:

Next 64byte only when above bit was set:

Aspect Ratio values:

Frame Rate values:

MPEG-1 Video Group of Pictures (GOP) (8 bytes) XXX...

MPEG-1 Video Picture Header XXX...

If Coding Type is 2 or 3 (P-Frame or B-Frame):

If Coding Type is 3 (B-Frame):

If (and while) next bit is set:

End of Extra:

Coding Type values:

Frame Order

The B-fames require to know the next P- (or I-) frame in advance, for that reason, the frames are stored as "PBBB" (although being played as "BBBP"):

MPEG-1 Video Slice

Slices are containing the actual 16x16 pixel Macro Blocks. Usually a Slice contains one horizontal line - although, theoretically, it could be longer or shorter, ie. a slice could wrap to next line, or a line could be split into several slices (with the leading "MBA Increment" value greater than 1 to define the horizontal start offset).

If (and while) next bit is set:

End of Extra:

If (and while) next 23bit are nonzero (ie. until next 000001xxh):

Final padding:

MPEG-1 Video Group/Sequence Extension Data (reserved)

MPEG-1 Video User Data (optional)

User Data can contain Closed Captions (see flag in VCD\INFO.VCD or SVCD\INFO.SVD). User Data contains 11h-byte "Created with Nero" in some homebrew discs.

MPEG-1 Video Sequence End Code (4 bytes)

MPEG-1 Video 4:2:0 Macroblock

Aka...

26.7 VCD MP2 Audio Stream

VCD video discs and .mpg movie files are having the MP2 Audio Stream enclosed in the Multiplex stream (whilst .mp2 audio files may contain raw MP2 data without Multiplex stream).

Each MP2 frame is starting with a FFFh syncword (which is always located on a byte boundary). Unfortunately, the value FFFh can also occur anywhere in the audio data (eg. a 16bit sample with value 3FFCh).
So, when starting mid-stream, one will need some guessing when searching a valid syncword. The best method is to compute the frame size (based on the supposed frame header), and then to check if supposed frame begins AND ends with a sync word. Moreover, one could check for invalid sample rate values in the frame header, or invalid "groupings" in the frame's data part.
VCDs are conventionally having three audio frames encoded in one CDROM sector, so the first syncword can be simply found right after the multiplex packet header (though that might differ in some cases: VCD2.0 allows different audio bitrates, and a CDROM sector could be theoretically shared for Audio and Video data).

Overall MP2 Frame Format

MP2 Header

MP2 Checksum (optional)

Allocation Information

Scale Factor Selector Information

Scale Factors

Data

26.8 Inflate

Inflate/Deflate is a common (de-)compression algorithm. In the PSX world, it's used by the .CDZ cdrom-image format.

Inflate - Core Functions
Inflate - Initialization & Tree Creation
Inflate - Headers and Checksums

26.9 Inflate - Core Functions

tinf_uncompress(dst,src)

tinf_inflate_uncompressed_block()

tinf_inflate_compressed_block()

tinf_decode_symbol(tree)

tinf_read_bits(num) ;get N bits from source stream

tinf_getbit() ;get one bit from source stream

tinf_align_src_to_byte_boundary()

26.10 Inflate - Initialization & Tree Creation

tinf_init()

tinf_build_bits_base(bits,base,delta,base_val)

tinf_build_fixed_trees()

tinf_decode_dynamic_trees()

tinf_build_tree(tree, first, num)

tinf_data

26.11 Inflate - Headers and Checksums

tinf_gzip_uncompress(void *dest, *destLen, *source, sourceLen)

tinf_zlib_uncompress(dst, destLen, src, sourceLen)

tinf_adler32(src, length)

27. CDROM Internal Info on PSX CDROM Controller

PSX software can access the CDROM via Port 1F801800h..1F801803h (as described in the previous chapters). The following chapters describe the inner workings of the PSX CDROM controller - this information is here for curiosity only - normally PSX software cannot gain control of those lower-level stuff (although some low level registers can be manipulated via Test commands, but that will usually conflict with normal operation).

Motorola MC68HC05 (8bit single-chip CPU)

The Playstation CDROM drive is controlled by a MC68HC05 8bit CPU with on-chip I/O ports and on-chip BIOS ROM. There is no way to reprogram that BIOS, nor to tweak it to execute custom code in RAM.
CDROM Internal HC05 Instruction Set
CDROM Internal HC05 On-Chip I/O Ports
CDROM Internal HC05 I/O Port Usage in PSX
CDROM Internal HC05 Motorola Selftest Mode
The PSX can read HC05 I/O Ports and RAM via Test Commands:
CDROM - Test Commands - Read HC05 SUB-CPU RAM and I/O Ports

Decoder/FIFO (CXD1199BQ or CXD1815Q)

This chip handles error correction and ADPCM decoding, and acts as some sort of FIFO interface between main/sub CPUs and incoming cdrom sector data. On the MIPS Main CPU it is controlled via Port 1F801800h..1F801803h.
CDROM Controller I/O Ports
On the HC05 Sub CPU it is controlled via Port A (data in/out), Port E (address/index), and Port D (read/write/select signals); the HC05 doesn't have external address/data bus, so one must manually access the CXD1815Q via those ports.
CDROM Internal CXD1815Q Sub-CPU Configuration Registers
CDROM Internal CXD1815Q Sub-CPU Sector Status Registers
CDROM Internal CXD1815Q Sub-CPU Address Registers
CDROM Internal CXD1815Q Sub-CPU Misc Registers
The PSX can read/write the Decoder I/O Ports and SRAM via Test commands:
CDROM - Test Commands - Read/Write Decoder RAM and I/O Ports

The sector buffer used in the PSX is 32Kx8 SRAM. Old PU-7 boards are using CXD1199BQ chips, later boards are using CXD1815Q, and even later boards have the stuff intergrated in the SPU. Note: The CXD1199BQ/CXD1815Q are about 99% same as described in CXD1199AQ datasheet.

Signal Processor and Servo Amplifier

Older PSX mainboards are using two separate chips:
CDROM Internal Commands CX(0x..3x) - CXA1782BR Servo Amplifier
CDROM Internal Commands CX(4x..Ex) - CXD2510Q Signal Processor
Later PSX mainboards have the above intergrated in a single chip, with some extended features:
CDROM Internal Commands CX(0x..Ex) - CXD2545Q Servo/Signal Combo
Later version is CXD1817R (Servo/Signal/Decoder Combo).
Even later PSX mainboards have it integrated in the Sound Chip: CXD2938Q (SPU+CDROM) with some changed bits and New SCEx transfer:
CDROM Internal Commands CX(0x..Ex) - CXD2938Q Servo/Signal/SPU Combo
Finally, PM-41(2) boards are using a CXD2941R chip (SPU+CDROM+SPU_RAM), unknown if/how far the CDROM part of that chip differs from CXD2938Q.
Some general notes:
CDROM Internal Commands CX(xx) - Notes
CDROM Internal Commands CX(xx) - Summary of Used CX(xx) Commands
The PSX can manipulate the CX(..) registers via some test commands:
CDROM - Test Commands - Test Drive Mechanics
Note: Datasheets for CXD2510Q/CXA1782BR/CXD2545Q do exist.

CDROM Pinouts

Pinouts - DRV Pinouts
Pinouts - HC05 Pinouts

27.1 CDROM Internal HC05 Instruction Set

ALU, Load/Store, Jump/Call

Operands can be...

Read-Modify-Write

Operands can be...

CLR includes a dummy-read-cycle, whilst TST does omit the dummy-write cycle.
The ",slow" RMW opcodes are smaller, but slower than equivalent ALU opcodes.

Bit Manipulation and Bit Test with Relative Jump (to $+3+/-dd)

Branch (Relative jump to $+2+/-nn)

Control/Misc

MUL isn't supported in original "M146805 CMOS" family (MUL is used/supported in PSX cdrom controller).

Registers

Pushed on IRQ are:

Addressing Modes

Notes:

Exception Vectors

Exception vectors are 16bit BIG-ENDIAN values at FFF0h-FFFFh (or at FFE0h-FFEFh when running in Motorola Bootstrap mode).

Directives/Pseudos (used by a22i assembler; in no$psx utility menu)

27.2 CDROM Internal HC05 On-Chip I/O Ports

HC05 Port 3Eh - MISC - Miscellaneous Register (R/W)

Note: For PSX, OSC is 4.0000MHz (PU-7/PU-8), 4.2336MHz (PU-18 and up). SysClk is usually set to OSC/2, ie. around 2MHz.

HC05 Port OPTM=0:00h - PORTA - Port A Data Register (R/W)

HC05 Port OPTM=0:01h - PORTB - Port B Data Register (R)

HC05 Port OPTM=0:02h - PORTC - Port C Data Register (R/W)

HC05 Port OPTM=0:03h - PORTD - Port D Data Register (R/W)

HC05 Port OPTM=0:04h - PORTE - Port E Data Register (R/W)

HC05 Port OPTM=0:05h - PORTF - Port F Data Register (R) (undoc: R/W)

These are general purpose I/O ports (controlling external pins). Some ports are Input-only, and some can be optionally used for special things (like IRQs, SPI-bus, or as Timer input/output).

HC05 Port OPTM=1:00h - DDRA - Port A Data Direction Register (R/W)

HC05 Port OPTM=1:02h - DDRC - Port C Data Direction Register (R/W)

HC05 Port OPTM=1:03h - DDRD - Port D Data Direction Register (R/W)

HC05 Port OPTM=1:04h - DDRE - Port E Data Direction Register (R/W)

HC05 Port OPTM=1:05h - DDRF - Port F Data Direction Register (undoc)

Officially, there are no DDRB and DDRF registers (Port B and F are always Inputs). Although, actually, Motorola's Bootstrap RAM \<does> manipulate DDRF.

HC05 Port OPTM=1:08h - RCR1 - Resistor Control Register 1 (R/W)

HC05 Port OPTM=1:09h - RCR2 - Resistor Control Register 2 (R/W)

HC05 Port OPTM=1:0Ah - WOM1 - Open Drain Output Control Register 1 (R/W)

HC05 Port OPTM=1:0Bh - WOM2 - Open Drain Output Control Register 2 (R/W)

==== Interrupts =====

HC05 Port OPTM=0:08h - INTCR - Interrupt Control Register (R/W)

HC05 Port OPTM=0:09h - INTSR - Interrupt Status Register (R and W)

HC05 Port OPTM=1:0Eh - KWIE - Key Wakeup Interrupt Enable Register (R/W)

==== SPI Bus ====

HC05 Port OPTM=0:0Ah - SPCR1 - Serial Peripheral Control Register 1 (R/W)

HC05 Port OPTM=0:0Bh - SPSR1 - Serial Peripheral Status Register 1 (R)

Note: SPSR1.7 appears to be reset after reading SPSR1 (probably same for SPSR1.6, and maybe also same for whatever SPI IRQ signal).

HC05 Port OPTM=0:0Ch - SPDR1 - Serial Peripheral Data Register 1 (R/W)

==== Time Base / Config ====

HC05 Port 10h - TBCR1 - Time Base Control Register 1 (R/W)

HC05 Port 11h - TBCR2 - Time Base Control Register 2 (R/W, some bits R or W)

HC05 Port OPTM=1:0Fh - MOSR - Mask Option Status Register (R)

Reading this register returns A0h (on PSX/PSone with 52pin chips).

==== Timer 1 ====

HC05 Port 12h - TCR - Timer 1 Control Register (R/W)

HC05 Port 13h - TSR - Timer 1 Status Register (R)

HC05 Port 14h - ICH - Timer 1 Input Capture High (undoc)

HC05 Port 15h - ICL - Timer 1 Input Capture Low (undoc)

HC05 Port 16h - OC1H - Timer 1 Output Compare 1 High (undoc)

HC05 Port 17h - OC1H - Timer 1 Output Compare 1 Low (undoc)

HC05 Port 18h - TCNTH - Timer 1 Counter 1 High (undoc)

HC05 Port 19h - TCNTL - Timer 1 Counter 1 Low (undoc)

HC05 Port 1Ah - ACNTH - Alternate Counter High (undoc)

HC05 Port 1Bh - ACNTL - Alternate Counter Low (undoc)

==== Timer 2 ====

HC05 Port 1Ch - TCR2 - Timer 2 Control Register (R/W)

HC05 Port 1Dh - TSR2 - Timer 2 Status Register (R/W)

HC05 Port 1Eh - OC2 - Timer 2 Output Compare Register (R/W)

HC05 Port 1Fh - TCNT2 - Timer 2 Counter Register (R) (W=Set Counter to 01h)

==== Reserved ====

HC05 Port 3Fh - Unknown/Unused

Reading this port via Sony's test command returns 20h (same as openbus), but reading it via Motorola's selftest function returns 00h (unlike openbus), so it seems to have some unknown/undocumented function; bit5 might indicate selftest mode, or it might reflect initialization of whatever other ports.

HC05 Port OPTM=0:06h..07h,0Dh..0Fh - Reserved

HC05 Port OPTM=1:01h,06h..07h,0Ch..0Dh - Reserved

HC05 Port 20h..3Dh - Reserved

These ports are unused/reserved. Trying to read them on a PSone does return 20h (possibly the prefetched next opcode value from the RAM test command). Other HC05 variants contain some extra features in these ports:
CDROM Internal HC05 On-Chip I/O Ports - Extras
The PSX CDROM BIOS doesn't use any of these ports - execpt, it is writing [20h]=2Eh (possibly to disable unused LCD hardware; which might be actually present in the huge 80pin HC05 chips on old PU-7 mainboards).

HC05 Openbus

Openbus values can be read from invalid memory locations, on PSX with 52pin chips:

The returned openbus value depends on the opcode's memory operand:

27.3 CDROM Internal HC05 On-Chip I/O Ports - Extras

HC05 Port OPTM=0:0Dh - SPCR2 - Serial Peripheral Control Register 2 (R/W)

HC05 Port OPTM=0:0Eh - SPSR2 - Serial Peripheral Status Register 2 (R)

HC05 Port OPTM=0:0Fh - SPDR2 - Serial Peripheral Data Register 2 (R/W)

This is a second SPI channel, works same as first SPI channel, but using the lower 3bits of Port G (instead of Port C) for the SPI signals.

HC05 Port OPTM=0:06h - PORTG - Port G Data Register (R/W)

HC05 Port OPTM=0:07h - PORTH - Port H Data Register (R/W)

HC05 Port 3Ch - PORTJ - Port J Data Register (R/W)

HC05 Port OPTM=1:06h - DDRG - Port G Data Direction Register (R/W)

HC05 Port OPTM=1:07h - DDRH - Port H Data Direction Register (R/W)

HC05 Port 20h - LCDCR - LCD Control Register (R/W)

HC05 Port 21h..34h - LCDDR1..20 - LCD Data Register 1..20 (R/W)

HC05 Port 34h - PWMCR - PWM Pulse Width Modulation Control Register (R/W)

HC05 Port 35h - PWMCNT - PWM Counter Register (R) (W=Set Counter to FFh)

HC05 Port 36h - PWMDR0 - PWM Duty Register 0 (R/W)

HC05 Port 37h - PWMDR1 - PWM Duty Register 1 (R/W)

HC05 Port 38h - PWMDR2 - PWM Duty Register 2 (R/W)

HC05 Port 39h - PWMDR3 - PWM Duty Register 3 (R/W)

HC05 Port 3Ah - ADR - A/D Data Register (R)

HC05 Port 3Bh - ADSCR - A/D Status and Control Register (R/W)

HC05 Port 3Dh - PCR - Program Control Register (R/W) (for EPROM version)

27.4 CDROM Internal HC05 I/O Port Usage in PSX

Port A - Data (indexed via Port E)

Port B - Inputs

Port C - Inputs/Outputs

Port D - Outputs

Port E - Index (for data on Port A)

Port F - Motorola Bootstrap Serial I/O (not used in cdrom bios)

Other HC05 I/O Ports

Note: The PSX has the HC05 clocked via 4.00MHz oscillator (older boards), or via 4.3MHz signal from SPU (newer boards); internally, the HC05 is clocked at half of those frequencies (ie. around 2 MHz).

27.5 CDROM Internal HC05 Motorola Selftest Mode

52-pin HC05 chips (newer psx cdrom controllers)

52-pin chips are used on LATE-PU-8 boards, and on later boards ranging from PU-18 up to PM-41(2).
CDROM Internal HC05 Motorola Selftest Mode (52pin chips)

80-pin HC05 chips (older psx cdrom controllers)

80-pin chips are used PU-7, EARLY-PU-8, and PU-9 boards.
CDROM Internal HC05 Motorola Selftest Mode (80pin chips)

32-pin HC05 chips (joypad/mouse)

Sony's Digital Joypad and Mouse are using 32pin chips (with TQFP-32 package), which are probably containing Motorola HC05 CPUs, too. Unknown if/how those chips can be switched into bootstrap/dumping modes.

Pinouts

Pinouts - HC05 Pinouts

27.6 CDROM Internal HC05 Motorola Selftest Mode (52pin chips)

Motorola Bootstrap ROM

The Motorola MC68HC05 chips are including a small bootstrap ROM which gets activated upon /RESET when having two pins strapped to following levels:

Moreover, two pins are needed on /RESET for selecting a specific test mode:

The selectable four modes are:

The upload/download functions are using following additional pins:

RX/TX are RS232-like serial signals (but using other voltages, 0=0V and 1=3.5V). Transfer format is 8-N-1, ie. one startbit(0), 8 databits LSB first, no parity, one stopbit(1). Baudrate is OSC/2/208 (ie. 9616 bps for 4.000MHz, or 10176 bps for 4.2336MHz clock derived from CXD2545Q/CXD2938Q).
Note: Above pins may vary on some chips (namely on chips that don't have PortF). The pins for entering bootstrap mode (PortC in this case) should be described in datasheets; but transfer protocol and mode selection (PortB) and transmission (PortF) aren't officially documented.

Mode2: Upload 200h bytes to RAM & jump to 0040h

This mode is very simple and powerful: After /RESET, you send 200h bytes to the RX input (without any response on TX output), the bytes are stored at 0040h..023Fh in RAM, and the chip jumps to 0040h after transferring the last byte. The uploaded program can contain a custum highspeed dumping function, or perform hardware tests, etc. A custom dumping function for PSX/PSone can be found at:

After uploading the 200h-byte dumping function it will respond by send 4540h bytes (containing some ASCII string, the 16.5Kbyte ROM image, plus dumps for RAM and (banked) I/O port region, plus openbus tests for unused memory and I/O regions.

Wiring for Mode2 on PSX/PSone consoles with 52-pin HC05 chips

Good places to pick 3.5V and 7.5V from nice solder pads are:

The SCOR trace on Pin31, connects to Signal Processor...

cut that trace (preferably on the PCB between two vias or test points, so you can later repair it easily) (better don't try to lift Pin31, it breaks off easily)
Note: Mode2 also requires Pin16=Low, and Pin30=High (but PSX/PSone boards should have those pins at that voltages anyways).

Mode3: Download ROM as ASCII hexdump

This mode is very slow and not too powerful. But it may useful if you can't get Mode2 working for whatever reason. Wiring for Mode3 is same as above, plus PortB.0=3.5V. In this mode, the chip will send one 0Dh,0Ah," AAAA DD " string immediately after /RESET (with 16bit address "AAAA" (initially 1000h), and 8bit data "DD"). Thereafter the chip will wait for incoming commands:

Basic setup would be wiring RX to GND (the chip will treat that as infinite stream of start bits with chr(00h), so it will respond by sending data from increasing addresses automatically; the increment wraps from 4FFFh to FE00h (skipping the gap between Main ROM and Bootstrap ROM), and also wraps from FFFFh to 0000h; transfer is ultraslow: 13 characters needed per dumped byte: chr(00h) to chip, chr(00h) echo from chip, and 0Dh,0Ah," AAAA DD " from chip.

27.7 CDROM Internal HC05 Motorola Selftest Mode (80pin chips)

80pin Sony 4246xx chips

And for anyone else planning to try this, these are the connections:

In bootstrap mode, Port A is used as follows:

The selectable testmodes are:

RX/TX are plain binary (non-ASCII), baudrate is 9600 (when using 4.000MHz oscillator), transfer format is 8,N,2 (aka 8,N,1 with an extra pause bit).

Wiring for Upload/Download on PSX consoles with 80-pin HC05 chips

Good places to pick 3.5V and 7.5V from nice solder pads are:

Credits to TriMesh for finding the 80pin chip's bootstrap signals.

Other 80pin chips

DTL-H100x uses 80pin chip with onchip PROM (chip text "(M) MC68HC705L15", instead of "Sony [...] 4246xx"), wiring for serial dumping on that is unknown (the bootstrap ROM may be a little different because it should contain PROM burning functions). PU-9 boards boards seem to use a similar PROM (with some sticker on it).
DTL-H2000 uses 80pin CXP82300 chip with socketed piggyback 32pin EPROM - that chip is a Sony SPC700 CPU, not a Motorola HC05 CPU. Accordingly there's no Motorola Bootstrap mode in it, but of course one could simply dump the EPROM with standard eprom utilities, but nobody did do so yet).

27.8 CDROM Internal CXD1815Q Sub-CPU Configuration Registers

00h - DRVIF - Drive Interface (W)

01h - CONFIG 1 - Configuration 1 (W)

02h - CONFIG 2 - Configuration 2 (W)

03h - DECCTL - Decoder Control (W)

07h - CHPCTL - Chip Control (W)

27.9 CDROM Internal CXD1815Q Sub-CPU Sector Status Registers

00h - ECCSTS - ECC Status (R)

01h - DECSTS - Decoder Status (R)

02h - HDRFLG - Header/Subheader Error Flags for HDR/SHDR registers (R)

Error flags for current sector are probably stored straight in this register (ie. these flags are probably available even without using 9bit SRAM).
Or maybe not... if the chip supports receiving newer sectors during time-consuming error corrections... then those newer would need to be stored in SRAM, and would thus require 9bit SRAM for the error flags?

03h - HDR - Header Bytes (R)

04h - SHDR - Subheader Bytes (R)

The contents of the HDRFLG, HDR, SHDR registers indicate:

Unknown when 1st..4th read indices are reset for HDR and SHDR (maybe on access to certain I/O ports, or maybe anytime when receiving a new sector), also unknown what happens on 5th read and up.

27.10 CDROM Internal CXD1815Q Sub-CPU Address Registers

Drive Address -- used for storing incoming CDROM sectors in Buffer RAM
Host Address -- used for transferring Buffer RAM to (or from) Main CPU
ADPCM Address -- used for Real-time ADPCM audio output from Buffer RAM

05h - CMADR - Drive Current Minute Address (R)

Indicates the start address of the most recently decoded sector (called "Minute Address" because it points to the MM byte of the MM:SS:FF:MODE sector header). Normally, CMADR should be forwarded to Host:

Whereas, offset would be usually 00h, 04h, or 0Ch (to start reading from the begin of the sector, or to skip 4-byte MODE1 header, or 12-byte MODE2 header). And length would be usually 800h (normal data sector), or 924h (entire sector, excluding the leading 12 sync-bytes). Length bit14 is undocumented/reserved, but the PSX CDROM BIOS does set that bit for whatever reason.
Alternately, the sector can be forwarded to the Real-time ADPCM decoder:

19h - ADPMNT - ADPCM "MNT" Address (W)

04h - DLADR-L, Drive Last Address, bit0-7 (W)

05h - DLADR-M, Drive Last Address, bit8-15 (W)

06h - DLADR-H, Drive Last Address, bit16 (W)

10h - DADRC-L - Drive Address Counter, bit0-7 (W)

11h - DADRC-M - Drive Address Counter, bit8-15 (W)

12h - DADRC-H - Drive Address Counter, bit16 (W)

0Eh - DADRC-L - Drive Address Counter, Bit0-7 (R)

0Fh - DADRC-M - Drive Address Counter, Bit8-15 (R)

0Ch - HXFR-L - Host Transfer Length, bit0-7 (W)

0Dh - HXFR-H - Host Transfer Length, bit8-11 and stuff (W)

0Eh - HADR-L - Host Transfer Address, bit0-7 (W)

0Fh - HADR-M - Host Transfer Address, bit8-15 (W)

0Ah - HXFRC-L - Host Transfer Remain Counter, bit0-7 (R)

0Bh - HXFRC-H - Host Transfer Remain Counter, bit8-11, and other bits (R)

0Ch - HADRC-L - Host Transfer Address Counter, bit0-7 (R)

0Dh - HADRC-M - Host Transfer Address Counter, bit8-15 (R)

> "This counter keeps the addresses which write or read the data with host into/from the buffer.
> When data from the host is written into the buffer or data to the host is read from the

buffer, the HADRC value is output from MA0 to 16. HADRC is incremented each time one byte of data from the drive is read from the buffer (BFRD is high) or written into the buffer (BFWR is high)."

Note

When reading from SRAM, data seems to go through a 8-byte data fifo, the HXFRC (remain) and HADRC (addr) values aren't aware of that FIFO (ie. if there's data in the fifo, then addr will be 8 bigger and remain 8 smaller than what has arrived at the host).

Unclear Notes

"If sound map data is to be transferred before the data is transferred (immediately after the host has set the BFRD and BFWR bits (bits 7 and 6) of the HCHPCTL register high)":

"At any other time":

Unknown what the above crap is trying to say exactly.
"At any other time" does apparently refer to cases when transfers get started (whilst during transfer, the address/remain values are obviously increasing/decreasing).
For sound map, theoretically, the SMEN bit should be set, but above does somewhat suggest that BFRD or BFWR (or actually: both BFRD and BFWR) need to be set...?

Sector Buffer Memory Map (32Kx8 SRAM)

27.11 CDROM Internal CXD1815Q Sub-CPU Misc Registers

16h - HIFCTL - Host Interface Control (W)

11h - HIFSTS - Host Interface Status (R)

0Ah - CLRCTL - Clear Control (W)

07h - INTSTS - Interrupt Status (R) - (0=No, 1=IRQ)

09h - INTMSK - Interrupt Mask (W) - (0=Disable, 1=Enable)

0Bh - CLRINT - Clear Interrupt Status (W) - (0=No change, 1=Clear/Ack)

12h - HSTPRM - Host Parameter (R)

HIFSTS.4 goes off when all bytes read.
Said to have 8-byte FIFO in CXD1199AQ datasheet.
But, PSX has 16-byte Parameter FIFO...!?!

13h - HSTCMD - Host Command (R)

Command should be ack'ed via CLRINT.1 and CLRCTL.6.

17h - RESULT - Response FIFO (W)

Said to have 8-byte FIFO in CXD1199AQ datasheet.
But, PSX has 16-byte Response FIFO...!?!

08h - ADPCI - ADPCM Coding Information (R)

Unknown if ADPCI is affected by configurations by Main-CPU's Sound Map ADPCM or by Sub-CPU's Real-time ADPCM (or by both)?
Note: Bit5,7 are semi-undocumented in the datasheet (mentioned in the ADPCI description, but missing in overall register summary).

1Bh - RTCI - Real-time ADPCM Coding Information (W)

06h,09h,10h,14h..1Fh - Reserved (R)

Of these, 09h and 10h are officially unused/reserved. And addresses 06h and 14h..1Fh aren't officially mentioned to exist at all.
Trying to read these registers on a PSone returns Data=C0h for 06h, 09h, 10h, 15h-16h, 18h-1Fh, and Data=FFh for 14h, and Data=DEh for 17h.

08h,13h-15h,18h,1Ah,1Ch-1Fh - Reserved (W)

Of these, 09h,13h-15h,18h,1Ah are officially unused/reserved. And addresses 1Ch-1Fh aren't officially mentioned to exist at all.

27.12 CDROM Internal Commands CX(0x..3x) - CXA1782BR Servo Amplifier

CXA1782BR - CX(0x) - Focus Servo Control - "FZC" FocusZeroCross at SENS pin

For Focus Search: Keep FS1=on, and toggle FS2 on and off (this will generate a waveform, and SENS will indicate when reaching a good focus voltage).

CXA1782BR - CX(1x) - Tracking/Brake/Sled - "DEFECT" at SENS pin

Note: The PSX CDROM BIOS does use the "Undoc" setting (ie. bit17=1), but the effect is undoc/unknown?

Note: CX(1x) works different on CXD2545Q (some bits are moved around, and the "SledKickHeight" bits are renamed to "SledKickLevel" and moved to different/new CX(3X) commands.

CXA1782BR - CX(2x) - Tracking and Sled Servo Control - "TZC" at SENS pin

CXA1782BR - CX(3x) - "Automatic Adjustment Comparator Output" at SENS pin

Note: CX(3x) is extended and works very different on CXD2545Q.

CXA1782BR Command 4x..7x - "HIGH-Z" at SENS pin

CXA1782BR Command 8x..Fx - "UNSPECIFIED???" at SENS pin

Note

The Servo Amplifier isn't directly connected to the CPU. Instead, it's connected as a slave device to the Signal Processor. There are two ways to access the Servo Amplifier:
1) The CPU can send CX(0X..3X) commands to the Signal Processor (which will then forward them to the Servo Amplifier).
2) The Signal Processor can send CX(0X..3X) commands to the Servo Amplifier (this happens during CX(4xxx) Auto Sequence command).

27.13 CDROM Internal Commands CX(4x..Ex) - CXD2510Q Signal Processor

CXD2510Q - CX(4xxx) - Auto Sequence

Values for AS (Auto Sequence Command):

CXD2510Q - CX(5x) - Blind,Brake,Overflow Timer

CXD2510Q - CX(6xx) - SledKick,Brake,Kick Timer

CXD2510Q - CX(7xxxx) - Track jump count setting (for Auto Sequence Command)

CXD2510Q - CX(8xx) - MODE Specification

CXD2510Q - CX(9xx) - Function Specification

CXD2510Q - CX(Axx) - Audio Control

Normal: SQSO outputs... WHAT?
PeakMeter: SQSO outputs highest peak ever on any channel (bit15: usually 0)
LevelMeter: SQSO outputs recent peak (with bit15 toggled: 0=Right, 1=Left)

CXD2510Q - CX(Bxxxx) - Traverse Monitor Counter Setting

CXD2510Q - CX(Cxx) - Spindle Servo Coefficient Setting

CXD2510Q - CX(Dx) - CLV Control

CXD2510Q - CX(Ex) - CLV Mode

Values for CM (CLV Mode):

N/A - CX(F) - Reserved

SUBQ Output

L/R is toggled after each SUBQ reading, however the PSX Report mode does usually forward SUBQ only every 10 frames, so it stays stuck in one setting (but may toggle after one second; ie. every 75 frames). And, peak is reset after each read, so 9 of the 10 frames are lost.

CXD2510Q - SENS output

Whereas,

27.14 CDROM Internal Commands CX(0x..Ex) - CXD2545Q Servo/Signal Combo

CXD2545Q - CX(0x) and CX(2x) - same as CXA1782BR Servo Amplifier

CDROM Internal Commands CX(0x..3x) - CXA1782BR Servo Amplifier

CXD2545Q - CX(4x..Ex) - same as CXD2510Q Signal Processor

CDROM Internal Commands CX(4x..Ex) - CXD2510Q Signal Processor
One small difference is that the CXD2545Q supports a new "M Track Move" function as part of the CX(4xxx) command. And, some "don't care" bits are now reserved (ie. some commands need to be padded with additional leading "0" bits).

CXD2545Q - CX(1x) - Anti Shock/Brake/Tracking Gain/Filter

CXD2545Q - CX(30..33) - Sled Kick Level

CXD2545Q - CX(34xxxx) - Write to Coefficient RAM

Allows to change the default preset coefficient values,
CDROM Internal Coefficients (for CXD2545Q)

CXD2545Q - CX(34Fxxx) - Write to Special Register

CXD2545Q - CX(35xxxx) - FOCUS SEARCH SPEED/VOLTAGE/AUTO GAIN

CXD2545Q - CX(36xxxx) - DTZC/TRACK JUMP VOLTAGE/AUTO GAIN

CXD2545Q - CX(37xxxx) - FZSL/SLED MOVE/Voltage/AUTO GAIN

CXD2545Q - CX(38xxxx) - Level/Auto Gain/DFSW (Initialize)

VCLM,FLM,RFLM,TCLM are accepted every 2.9ms.

CXD2545Q - CX(39xx) - Select internal RAM/Registers for serial readout

Serial Readout Addresses:

CXD2545Q - CX(3Ax000) - Focus BIAS addition enable

CXD2545Q - CX(3Bxxxx) - Operation for MIRR/DFCT/FOK

CXD2545Q - CX(3Cxxxx) - TZC for COUT SLCT HPTZC (Default)

CXD2545Q - CX(3Dxxxx) - TZC for COUT SLCT DTZC

CXD2545Q - CX(3Exxxx) - Filter

CXD2545Q - CX(3Fxxxx) - Others

CXD2545Q feedback on 39xx: see pg. 77 (eg. 390C = VC AVRG)

XXX

CXD2545Q - SENS output

*1 $38 outputs AGOK during AGT and AGF command settings, and XAVEBSY during AVRG measurement.
SSTP is output in all other cases.

27.15 CDROM Internal Commands CX(0x..Ex) - CXD2938Q Servo/Signal/SPU Combo

Most commands are same as on CXD2545Q. New/changed commands are:

CXD2938Q - CX(349xxxxx) - New SCEx

Older PSX consoles have received the "SCEx" string at 250 baud via HC05 PortB.bit1, which allowed modchips to injected faked "SCEx" signals to that pin. To prevent that, the CXD2938Q contains some new 32bit commands that allow to receive somewhat encrypted "SCEx" strings via SPI bus. The used commands are:

The relevant command is NewScexRandomKey(xy) which does send a random value (x=01h..0Fh, and y=01h), and does then receive a 12-byte response via SPI bus (which is normally used to receive SUB-Q data).

The 12-byte packet does contain one SCEx character encoded in 4th..10th byte corresponding to Flags in "xy" bit 7..1 (in that order):
All bytes with Flag=1 are ORed together to compute a Character byte (those bytes could be all set to 53h for "S", or if more than one flag is set, it could be theoretically split to something like 41h and 12h).
All bytes with Flag=0 are ORed together to compute a Dummy byte. If the Character byte is same as the Dummy byte, then it gets destroyed by setting Character=00h (to avoid that, one could set all dummies to 00h, or set one or more dummies to FFh, for example).
Finally, "xy" bit0=0 indicates that the resulting character byte is inverted (XORed by FFh), however, the CDROM BIOS does always use bit0=1, so the inversion feature is never used.
For the whole SCEx string, there must be at least one 00h byte inserted between each character (or some Char=Dummy mismatch, which results in Char=00h either), and there should be a few more 00h bytes preceeding the first character ("S").
Note: Modchips didn't bother to reproduce that new SCEx transfers, instead they have simply bypassed it by injecting the 250 baud SCEx string to some analog lower level signal.

CXD2938Q - CX(3Bxxxx) - Some Changed Bits

Same as in older version, but initialized slightly differently: CXD2545Q used CX(3B2250) whilst CXD2938Q is using CX(3B7250).

CXD2938Q - CX(3Cxxxx) - TzcCoutSelect with New/Changed Extra Bits

The CXD2545Q used two 8bit commands, CX(3C) and CX(3D), for TzcOut selection, which are now replaced by a single 24bit command, CX(3Cxxxx), and which do include a new mode related to New SCEx.

CXD2938Q - CX(8xxxxx) - Disc Mode with New/Changed Extra Bits

Command CX(8xx) has been 12bit wide on CXD2545Q, and is now expanded 24bit width (with some changed/unknown bits).

CXD2938Q - CX(9xx000) - Normal/Double Speed with New Extra Bits

Command CX(9xx) has been 12bit wide on CXD2545Q (with bit12=reserved/zero), and is now expanded 24bit width (with bit12=unknown/one and bit11-0=unknown/zero).

CXD2938Q - CX(Dx0000) and CX(Ex0000) - New Zero Padding

Commands CX(Dx) and CX(Ex) have been 8bit wide on CXD2545Q, and are now zeropadded to 24bit width, ie. CX(Dx0000) and CX(Ex0000). Unknown if the extra bits are hiding any extra features. In practice, the CDROM BIOS is always setting them zero (except in some test commands which are accidently still using the old 8bit form, resulting in garbage in lower 16bits).

27.16 CDROM Internal Commands CX(xx) - Notes

Serial Command Transmission (for Signal Processor and Servo Amplifier)

Commands are sent serially LSB first via DATA,CLOK,XLAT pins: DATA+CLOK are used to send commands to the chip, command execution is then applied by dragging XLAT low for a moment.
Commands can be up to 24bits in length, but unused LSBs (the "don't care" bits) can be omitted; the PSX BIOS clips the length to 8bit/16bit where possible (due to the LSB-first transfer order, the chip does treat the most recently transferred bit as MSB=bit23, and there's no need to transfer the LSBs if they aren't used).
Aside from being used as command number, the four most recently transferred bits are also used to select SENS status feedback (for the SENS selection it doesn't matter if the four bits were terminated by XLAT or not).

Sled Motor / Track Jumps / Tracking

The Sled motor moves the drive head to the current read position. On a Compact Disc, the term "Track" does normally refer to Audio tracks (songs). But the drive hardware uses the terms "Track" and "Tracking" for different purposes:
Tracking appears to refer to moving the Optics via magnets (ie. moving only the laser/lens, without actually moving the whole sled) (this is done for fine adjustments, and it seems to happen more or less automatically; emulators can just return increasing sectors without needing to handle special tracking commands).
Track jumps refer to moving the entire Sled, one "track" is equal to one spiral winding (1.6 micrometers). One winding contains between 9 sectors on innermost windings, and 22.5 sectors on outermost windings (the PSX cdrom bios is translating the sector-distance to non-linear track-distance, and emulators must undo that translation; otherwise the sled doesn't arrive at the intended location; the cdrom bios will retry seeking a couple of times and eventually settle down at the desired location - but it will timeout if the sled emulation is too inaccurate).
The PSX hardware uses two mechanisms for moving the Sled:
Command CX(4xxx) Forward/Reverse Track Jump: allows to move the sled by 1..131070 tracks (ie. max 210 millimeters), and the hardware does stop automatically after reaching the desired distance.
Command CX(2x) Forward/Reverse Fast Sled: moves the sled continously until it gets stopped by another command (in this mode, software can watch the COUT signal, which gets toggled each "N" tracks; whereas "N" can be selected via Command CX(Bxxxx),

which is configured to N=100h in PSX).
The PSX cdrom bios is issuing another Fast Sled command (in opposite direction) after Fast Sled commands, emulators must somehow interprete this as "sled slowdown" (rather than as actually moving the sled in opposite direction, which could move the sled miles away from the target). For some reason vC1 BIOS is using a relative short slowdown period, whilst vC2/vC3 are using much longer slowdown periods (probably related to different SledKickHeight aka SledKickLevel settings and/or to different Sled Move Voltage settings).

Focus / Gain / Balance

The hardware includes commands for adjusting & measuring focus/gain/balance. Emulators can just omit that stuff, and can always signalize good operation (except that one should emulate failures for Disc Missing; and eventually also for cases like Laser=Off, or Spindle=Stopped).
Focus does obviously refer to moving the lens up/down. Gain does probably refer to reflection level/laser intensity. Balance might refer to tracking adjustments or so.

27.17 CDROM Internal Commands CX(xx) - Summary of Used CX(xx) Commands

The PSX CDROM BIOS versions vC1, vC2, and vC3 are using following CX() commands:

Note: for vC2, some CX(38xxxx) values may differ depending on "set_mid_lsb_to_140Eh".
For vC2, CX(Dx) and CX(Ex) should be officially zero-padded to CX(Dx00) and CX(Ex00), but the vC2 BIOS doesn't do that, it still uses short 8bit form.
For vC2, CX(Dx) and CX(Ex) should be apparently zero-padded to CX(Dx0000) and CX(Ex0000), at least, the vC3 BIOS is doing so (except on some test comannds that do still use the CX(Ex) short form).

Used Sense Values

27.18 CDROM Internal Coefficients (for CXD2545Q)

The CXD2545Q contains Preset Coefficients in internal ROM, which are copied to internal Coefficient RAM shortly after Reset. CX(34xxxx) allows to change those RAM settings, and CX(39xxxx) allows to readout some of those values serially.

CXD2545Q - Coefficient Preset Values