libHdd Reference Manual

hdd utility library by:
Nicholas Van Veen

hdd drivers by:
Nicholas Van Veen,
Florin Sasu,
Marcus R. Brown,
Vector

fileXio file IO library by:
adresd

Documentation by:
Nicholas Van Veen

Development of parts of libhdd (specifically, the utiliy library, APA driver and PFS driver) was sponsored by **DMS Technologies**, the makers of the excellent **DMS3** PS2 modchip - http://www.dms3.com

Table of Contents

Introduction to libHdd

libHdd is a set of drivers and source code which provides access to a hard disk drive (hdd) connected to your PS2. The library allows you to manage partitions and provides access to filesystems on the hdd. Partitions are organized on the hdd using what is known as the “aligned partition allocation” system, also known as “APA”. The filesystem type supported by libHdd is called the “Playstation Filesystem”, also known as “PFS”. Both PFS and APA are described in more detail later on in this document.

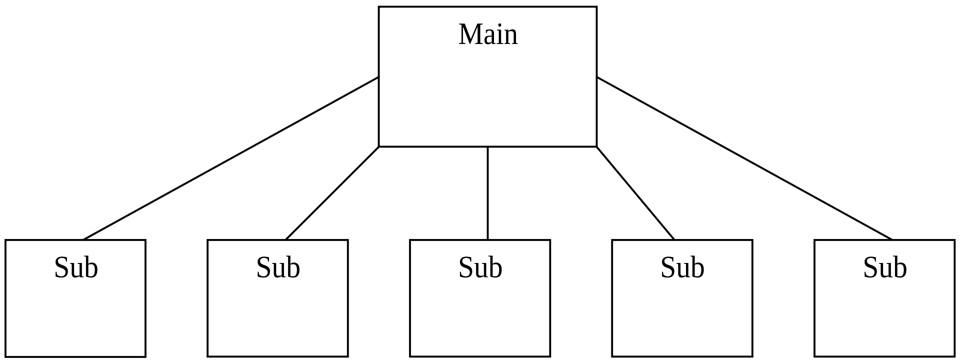
libHdd is made up of several distinct parts:

- APA driver, manages APA partitions on the hdd.
- PFS driver, manages access to PFS filesystems stored within APA partitions.
- Utility library, sits on top of the APA and PFS drivers and simplifies access to the hdd.

Something important to note is that Playstation 2 hdd-enabled games use the same APA partition system which libHdd supports, so both hdd-enabled games and homebrew software can access the hdd without conflicts. However, libHdd cannot be used to access partitions/ filesystems created by these games.

Introduction to APA

The hard disk drive (hdd) on the PlayStation 2 is organized using a custom partitioning system known as aligned partition allocation (APA). In this system there are two types of partitions – main partitions and sub partitions. Each main partition can have up to 64 sub partitions linked to it, as shown by the illustration below:



Sub partitions are very similar to main partitions; the main difference is in the way sub partitions are treated by the Playstation Filesystem (see introduction to PFS). The main partition and all sub partitions attached to it are treated as a single “block device” by the filesystem. Due to this it is possible to create filesystems of various sizes using different combinations of sub partition sizes. It is also possible to expand the size of a filesystem after the initial formatting by adding additional sub partitions to the main partition.

Partition sizes are restricted to a power of 2 (from 128mb to 32GB). However, the APA driver imposes a limitation regarding the size of a newly created partition. The size is restricted to approximately 1/32 the size of the entire hdd, so if for example if the hdd size is 40GB (the size of the official Sony hdd), then the maximum partition size will be 1GB.

Partitions are arranged on the hdd using the aligned partition allocation method. When a partition is placed on the hdd, it will be placed starting on a sector which is aligned with the partition size. The sector which is aligned with the partition size becomes the header for the partition. The header is 1kb large.

Main partitions have an “extended attributes area” placed after the partition header. This area is approximately 4mb large. You are able to read and write to this area; however it is not managed by a filesystem. Read and write access is provided by standard read/ write/ lseek calls (see hdd.irx reference), however you are restricted to reading/ writing in units of 512 bytes. This area is generally only used to store icon files to be displayed by a hdd-enabled PS2 browser replacement. There are currently no details on exactly how the attributes area is used to store icon files etc, so any information regarding this would be appreciated.

The layout of partitions is described below:

Offset	Data
0kb	Partition header
1kb	Reserved area
4kb	Extended attribute area
4mb	Start of filesystem

Table: Layout for main partition

Offset	Data
0kb	Partition header
1kb	Reserved area
4kb	Start of filesystem

Table: Layout for sub partition

Introduction to PFS

Playstation 2 software also uses a custom filesystem which sits on top of the APA partition system. This filesystem is known as the “Playstation Filesystem” or “PFS” for short. PFS is a 64-bit journaling filesystem, with similar characteristics to *nix filesystems such as EXT2. The maximum possible size of the filesystem is technically 2TB, though it’s unlikely anybody will ever create a PFS filesystem of that size. The maximum file size is also 2TB. It is also important to note that while PFS is a 64-bit filesystem, the unit of a single read/write/seek operation is 32-bits.

PFS features

- **Logical volumes:** PFS allows multiple partitions to be treated as a single filesystem. That is, where there are sub-partitions attached to a main partition, the combination of the main partition and all its sub-partitions are treated as a single filesystem, or logical volume. Even after building a filesystem, it is possible to increase its size as needed by adding additional sub-partitions. The next time the filesystem is mounted, the driver will recognize the additional partitions and adjust the filesystem accordingly (i.e. format the new partitions and adjust the free space etc).

- **Directories:** PFS uses a hierarchical directory structure similar to conventional filesystems such as EXT2. The size of a directory is of a variable length and the number of files which can be placed in each directory is unlimited. However, currently there is no directory cache implemented in the PFS driver so the more files in each directory, the slower the performance will be when accessing those files. If possible, keep the number of files in each directory to a minimum.

- **Filenames:** You are free to use any character in filenames except for ‘/ ’. The maximum length of the name of a single file is 255 characters, and the maximum length of a path name is 1024 characters.

- **File modes:** PFS supports file modes such as execution/ write/ read rights to individual files and directories. However, currently in Playstation 2 software there is no such concept as the “user”, so currently in the PFS driver the user id and group id for each file is fixed.

- **Journalling:** PFS performs a process called “journalling” that enables the recovery of a filesystem which has become corrupted in the event of a system crash or power failure. When an operation such as creating/modifying a file is performed, in addition to updating the actual file contents its is necessary to update the metadata (i.e. inodes, directory entries) on the hdd. If there is a failure while writing the metadata back to the hdd, then the metadata will become inconsistent and it may not be possible to read back the file’s contents properly. In most conventional filesystems you would be required to run the filesystem check utility in order to attempt to repair the corrupt filesystem, and even then a full recovery may not be possible. In PFS, metadata is first written to the journal area and then written to its actual destination on the hdd after it has been written to the journal area. In the event of a failure, corrupted metadata can be recovered from the journal area. In PFS, journalling is performed only for metadata – file data is written to the hdd immediately.

PFS driver performance

- **Memory efficiency:** The PFS driver is designed to reduce memory usage on IOP as much as possible, while still achieving impressive performance. It is also designed so that the memory usage is completely customizable. When loading the module you can specify the number of buffers/caches to use. The more buffers you use, the greater the performance will be, but this will also result in more memory consumption. Currently, the size of a single buffer is fixed at 1052 bytes.

- **File location:** When files are placed on the hdd, the driver tries to allocate a continuous area as large as possible to store the file contents so that file contents can be read back at a maximum speed. Due to this, write speed is relatively slower than reading but only when the area is being allocated. It is possible to improve write performance by allocating space for a file in advance, if you have an idea of how large the file will be.

- **Setting the "Zone" size:** Like most filesystems, in PFS files are constructed from small fragments. In PFS these fragments are called "Zones". A zone corresponds to a block in EXT2. When formatting the filesystem you are required to select a zone size (which must be a power of 2, in the range of 2kb to 128kb). You can increase performance by selecting a zone size which suits the data you will be storing, i.e. if you will be storing a number of smaller files then you will be better off using a small zone size, while if storing a small number of larger files you will be better off using a large zone size.

Overview of the proposed homebrew hdd usage guidelines

I believe that in order to maintain consistency in homebrew applications which utilize the hdd, a set of usage guidelines needs to be put in place. I have drafted a simple set of guidelines which are explained below.

- **Logical filesystems:** PFS supports filesystems/ logical volumes made up of multiple partitions. Applications should treat a main partition and any sub partitions attached to the main as a single filesystem.

- **Filesystem groups:** Filesystems on the hdd can be broken into three distinct groups: Application, System and Common. Application filesystems are those created by hdd-enabled Playstation 2 titles, or homebrew applications which require their own dedicated filesystem. System filesystems are those created and used by system software such as the Playstation 2 browser. Common filesystems are those created by the user (with a tool such as the DMS hdd format tool). They can be created by the user as they wish, for whatever purpose they wish. Applications which allow loading files from the hdd, for instance an emulator which loads rom files or a media player which loads movie/audio files, would let the user browse through any of the common filesystems on the hdd. An example of this is PGEN which allows you to browse through any common filesystems on the hdd when selecting a game to play.

- **Partition naming:** System partitions are prefixed with "__" (double underscore), i.e. "__boot". Common partitions are prefixed with "+", i.e. "+Media". Application partitions have no prefix.

- **Boot filesystem:** The format function from libHdd (see below) creates a system filesystem named "__boot". This filesystem is reserved for storing homebrew applications which may be executed by a loader application. The details for the layout of this filesystem have not been finalized at present. I would like to get feedback from other developers as to how this partition should be used, then a standard can be released which can be followed by anybody who wishes to create a loader application. Then any loader applications can share the boot partition without problems.

The above guidelines are currently enforced by the DMS hdd format tool. They are still a draft, so I would like to hear back from other developers with any feedback.

libHdd utility library

The libHdd utility library provides functions for listing/removing/creating filesystems on the hdd, formatting the hdd, checking for the presence of a hdd and handling the power-off procedure. It adheres to the draft homebrew hdd usage guidelines specified above.

Much of the library relies on the ps2drv project, and in particular the fileXio module of ps2drv. fileXio is a replacement file IO manager written by adresd, which is used by the utility library and is used to access the APA and PFS drivers. In order to build and use libHdd you must have a copy of ps2drv.

Something to note is that when the hdd drivers are resident, the PS2's reset button no longer functions as normal. The resident software must detect when the reset button is pressed, make any preparations required before the system shuts down (such as closing all files on the hdd to prevent corruption) and then finally power the system down through software. libHdd is bundled with a default power-off handler which will close all PFS files and power off the PS2, when the reset button has been pressed. This is described in more detail in the following section.

Structures

```
typedef struct {
        char name[32];char filename[40];
        u32 size;
        int formatted;
        u32 freeSpace;
        int fileSystemGroup;
 } t_hddFilesystem;
```

The t_hddFilesystem structure is used to hold information for a filesystem.

'name'	The name of the filesystem (with any prefix characters stripped). I.e. a filesystem with the filename "hdd0:+Media" would have the name "Media".
'filename'	The actual filename for the filesystem which can be used to open the filesystem's main partition using the file IO library.
'size'	Total size of the filesystem, in mega-bytes
'formatted'	TRUE if filesystem is formatted, otherwise FALSE
'freeSpace'	The filesystem's free space, in mega-bytes
'fileSystemGroup'	The filesystem group (one of: FS_GROUP_SYSTEM, FS_GROUP_COMMON or FS_GROUP_APPLICATION)

```
typedef struct {
        int hddSize;
        int hddFree;
        int hddMaxPartitionSize;
 } t_hddInfo;
```

The t_hddInfo structure is used to hold information about the current state of the hdd.

'hddSize'	Total capacity of the hdd in mega-bytes
'hddFree'	Amount of free space (space not currently used by partitions) on the hdd
'hddMaxPartitionSize'	The maximum size allowed for a single partition, in mega-bytes

Functions

```
int hddCheckPresent();
```

Checks for the presence of a supported hdd.

Returns 0 if a supported hdd is found, otherwise -1.

```
int hddCheckFormatted();
```

Checks if the connected hdd is properly formatted.

Returns 0 if the connected hdd is properly formatted, otherwise -1.

```
int hddFormat();
```

Formats the connected hdd with APA and creates the “boot” system filesystem.

Returns 0 on success, -1 times errno if an error occurred.

```
void hddGetInfo(t_hddInfo *info);
```

‘info’	Pointer to a t_hddInfo structure where the current state of the HDD will be stored.

Fills the structure pointed to by **info** with the current state of the hdd (i.e, information regarding the total hdd capacity, free space etc).

```
int hddGetFilesystemList(t_hddFilesystem hddFs[], int maxEntries);
```

‘hddFs’	Pointer to an array of t_hddFilesystem structures where list will be stored.
‘maxEntries’	The max number of list entries which will be written.

Retrieves a list of filesystems present on the hdd. The list is stored in the buffer pointed to by **hddFs**. At most, **maxEntries** structures are filled.

Returns the number of t_hddFilesystem structures filled on success, -1 times errno if an error occurred.

```
int hddMakeFilesystem(int fsSizeMB, char *name, int type);
```

'fsSizeMB'	Size of the new filesystem in mega-bytes, must be a multiple of 128mb.
'name'	Name of the new filesystem
'type'	Filesystem type. One of: FS_GROUP_SYSTEM, FS_GROUP_COMMON or FS_GROUP_APPLICATION

Creates a new filesystem on the hdd. The function will first create a main partition as large as possible, then attempt to create sub-partitions until the sum of main and sub partition sizes meets the desired filesystem size.

Currently, a system filesystem may only be created if named "boot". Any attempt to create a system filesystem with a different name will fail.

Returns the size of the created filesystem in mega-bytes, or -1 times errno if an error occurred. You should check that the returned value is equal to the value you pass for fsSizeMB, as it is possible that the size of the filesystem which was created is not as big as you requested.

```
int hddRemoveFilesystem(t_hddFilesystem *fs);
```

'fs'	Pointer to a t_hddFilesystem structure which holds information corresponding to the filesystem which is to be deleted.

Removes a filesystem from the hdd.

Returns 0 on success, -1 times errno if an error occurred.

```
int hddExpandFilesystem(t_hddFilesystem *fs, int extraMB);
```

'fs'	Pointer to a t_hddFilesystem structure which holds information corresponding to the filesystem to be resized.
'extraMB'	The amount of mega-bytes to increase the filesystem's size by. Must be a multiple of 128mb.

Expands a filesystem by adding additional sub-partitions.

Returns the amount the filesystem was expanded by, or -1 times errno if an error occurred. As with the hddMakeFilesystem function, you should check that the return value matches the value passed for extraMB.

```
void hddPreparePoweroff();
```

Installs the default power-off handler. The default poweroff hander will close all PFS files then power off the PS2 when the reset button is pressed. For this operation it is required that poweroff.irx (included with libHdd) is loaded on IOP. This can actually be loaded before or after the call is made to this function.

```
void hddSetUserPoweroffCallback(void (*user_callback)(void *arg), void
*arg);
```

‘user_callback’	Pointer to the callback function
‘arg’	Pointer to a buffer which will be passed as an argument to the callback function.

Sets a user callback function to be executed during power-off handling (ie: when the PS2 reset button has been pressed). ‘user_callback’ is a pointer to the callback function, and ‘arg’ is a pointer which will be passed to the callback function when it is called.

```
void hddPowerOff();
```

Activates the power-off handler, which first closes all PFS files then powers down the system.

The APA driver

The APA driver (hdd.irx) manages access to the hdd at the partition level. It also provides support for a number of miscellaneous tasks such as flushing the hdd's cache, shutting down DEV9 etc. All access to the APA driver is done through the "fileXio" file IO manager.

The APA driver requires that ps2dev9.irx and ps2atad.irx are loaded first. On loading the APA driver, you can pass arguments to control the performance and memory consumption of the driver. The possible arguments are as follows:

-o <#>	- Maximum number of partitions which may be opened simultaneously. In the current implementation of the driver, each open consumes 564 bytes of memory.
-n <#>	- Number of buffers to use. The speed of operation may be improved by increasing the number of buffers. By default 3 buffers are used, and each consumes 1048 bytes of memory.

Throughout the remained of this chapter, usage of the APA driver via fileXio is documented.

Structures

```
typedef struct {
        unsigned int    mode;
        unsigned int    attr;
        unsigned int    size;
        unsigned char   ctime[8];
        unsigned char   atime[8];
        unsigned char   mtime[8];
        unsigned int    hisize;
        unsigned int    private_0;
        unsigned int    private_1;
        unsigned int    private_2;
        unsigned int    private_3;
        unsigned int    private_4;
        unsigned int    private_5;
} iox_stat_t;
```

'mode'	Filesystem type of the partition. Possible values include FS_TYPE_PFS, FS_TYPE_EXT2 or FS_TYPE_EXT2_SWAP.
'attr'	Specifies if the partition is a main or sub partition. Possible values include ATTR_SUB_PARTITION and ATTR_MAIN_PARTITION
'size'	Number of sectors in the partition.

'ctime'	Creation time of the partition.
	
	ctime[1] = seconds
	ctime[2] = minutes
	ctime[3] = hours
	ctime[4] = day
	ctime[5] = month
	ctime[6 & 7] = year (4 digits)

'private_0'	For a main partition, this field holds the number of sub partitions. For a sub partition, holds the sub partition number.

```
typedef struct {
        iox_stat_t      stat;
        char            name[256];
        unsigned int    unknown;
} iox_dirent_t;
```

'stat'	Partition status, as described above.
'name'	Partition name

Functions

```
int fileXioFormat(const char *dev, const char *blockdev, const char *args, int arglen);
```

‘dev’	Device name (currently fixed at “hdd0:”)
‘blockdev’	Must be NULL
‘args’	Must be NULL
‘arglen’	Must be 0

Formats the hdd with the APA partition format. The required system partitions are created, however they are not formatted. The hdd will not be recognized by hdd enabled PS2 games until these system partitions have been formatted.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioOpen(const char *name, int flags, int modes);
```

‘name’	Partition identifier string
‘flags’	One or more (via logical OR) of the following: O_RDONLY – Open as read only O_RDWR – Open as read/write O_CREAT – Create a new partition
‘modes’	Must be 0

Creates a new main partition or opens an existing main partition. The partition identifier string consists of the device, the partition name, and the partition size if a new partition is being created.

Examples for the partition identifier string (source):

1. “hdd0:Media” to open the existing partition called “Media”
2. “hdd0:Games,128M” to create a partition called “Games”, which is 128MB large.

The possible choices for the size part of the partition identifier are:

128M, 256M, 512M, 1G, 2G, 4G, 8G, 16G, 32G

Returns the file descriptor for the partition (> 0) on success, -1 times errno if an error occurred.

```
int fileXioClose(int fd);
```

'fd'	The file descriptor returned by fileXioOpen

Closes an opened partition and frees the file descriptor.

Returns 0 on success, -1 times errno on failure.

```
int fileXioDopen(const char *name);
```

'name'	Device name (currently fixed at "hdd0:")

Opens a partition table for reading. Once a partition table is open, you may retrieve a list of all partitions on the device through calls to fileXioDread.

Returns a file descriptor on success (> 0), -1 times errno if an error occurred.

```
int fileXioDread(int fd, iox_dirent_t *dirent);
```

'fd'	File descriptor returned by fileXioDopen
'dirent'	Pointer to a iox_dirent_t structure which will be filled according to the next entry in partition entry list.

In order to retrieve a list of all partitions on the device opened by fileXioDopen, you can iterate over this function. Each time the function is called, the next entry in the partition list is read and the structed at dirent is filled.

Returns the length of the partition name (length of string in name member of iox_dirent_t), or 0 if the end of the partition list is reached (ie: when this returns 0, its time to stop the iteration). If an error occurred, returns -1 times errno.

```
int fileXioDclose(int fd);
```

'fd'	File descriptor returned by fileXioDopen

Closes a partition table opened by fileXioDopen. Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioGetStat(const char *name, iox_stat_t *stat);
```

‘name’	Partition identifier string (as described above)
‘stat’	Pointer to buffer where partition information will be stored

Fills a iox_stat_t structure with information corresponding to the partition specified by the partition identifier string.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioRemove(const char* name);
```

‘name’	Partition identifier string (as described above)

Removes the specified main partition. All attached sub partitions are also deleted.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioRead(int fd, unsigned char *buf, int size);
```

‘fd’	File descriptor for partition returned by fileXioOpen
‘buf’	Buffer on EE where data will be stored when read
‘size’	Amount of data to read. Must be a multiple of 512 bytes.

This will read data from the extended attribute area of the main partition specified by fd (the extended attribute area is explained in the introduction to APA).

Returns the number of bytes read on success, or -1 times errno if an error occurred.

```
int fileXioWrite(int fd, unsigned char *buf, int size);
```

Analogous to fileXioRead except for this function data is written from EE to the hdd rather than data read from the hdd to EE.

```
int fileXioLseek(int fd, long offset, int whence);
```

‘fd’	File descriptor for partition returned by fileXioOpen
‘offset’	Distance to move read/write pointer (must be a multiple of 512 bytes)
‘whence’	One of SEEK_SET, SEEK_CUR, SEEK_END

This moves the read/write position of reading/writing to the extended attribute area of the partition specified by fd.

Returns the updated position on success, -1 times errno if an error occurred.

```
int fileXioDevctl(const char *name, int cmd, void *arg, unsigned int arglen,
void *buf,unsigned int buflen);
```

‘name’	Device name (currently fixed at “hdd0:”)
‘cmd’	For the APA driver, one of:

1. HDDCTL_MAX_SECTORS
2. HDDCTL_TOTAL_SECTORS
3. HDDCTL_STATUS
4. HDDCTL_FORMAT
5. HDDCTL_FREE_SECTORS

‘arg’	Command arguments. Depends on cmd.
‘arglen’	Size of arg, in bytes.
‘buf’	Buffer for data received from command. Depends on cmd.
‘buflen’	Size of buf, in bytes.

This function performs special operations for a device (in this case, the hdd). See a subsequent section for details on each command.

Returns a command dependent value on success, or -1 times errno if an error occurred.

```
int fileXioIoctl2(int fd, int command, void *arg, unsigned int arglen, void *buf,
unsigned int buflen);
```

‘fd’	The fd assigned to the partition, returned by fileXioOpen
‘cmd’	For the APA driver, one of:

1. HDDIO_ADD_SUB
2. HDDIO_DELETE_END_SUB
3. HDDIO_NUMBER_OF_SUBS
4. HDDIO_GETSIZE

‘arg’	Command arguments. Depends on cmd.
‘arglen’	Size of arg, in bytes.
‘buf’	Buffer for data received from command. Depends on cmd.
‘buflen’	Size of buf, in bytes.

This function performs special operations on a file descriptor (in this case, an open partition). See a subsequent section for details on each command.

Returns a command dependent value on success, or -1 times errno if an error occurred.

Devctl commands

HDDCTL_MAX_SECTORS

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the maximum size for a partition that can be created, in units of sectors (512 byte units).

Returns the maximum size.

HDDCTL_TOTAL_SECTORS

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the total capacity of the hdd, in sectors (512 byte units).

Returns the number of sectors.

HDDCTL_DEV9_SHUTDOWN

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Powers down the dev9 device which the hdd is connected to. This is done by the default power-off processing code supplied by libHdd.

Returns 0.

HDDCTL_STATUS

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the hdd status.

Returns a value which corresponds to the current drive status. Possible return values are as follows:

0 – Normal
1 – hdd is not formatted
2 – hdd is locked
3 – hdd is not connected to the DEV9 device

HDDCTL_FORMAT

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the version of the APA system on the hdd.

Returns the version number.

HDDCTL_FREE_SECTORS

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Pointer to an ‘int’ for storing the free sector amount
‘buflen’	4

Gets the free space, in sectors, as indicated by the APA driver. The free space value is stored in buf.

Returns 0 on success.

Ioctl2 commands

HDDIO_ADD_SUB

'arg'	Pointer to the partition size string
'arglen'	Size of arg
'buf'	Must be NULL
'buflen'	Must be 0

Adds a sub partition of the size specified by the string pointed to by arg to the main partition specified by the file descriptor used with the fileXioIoctl2 call.

Example:

```
char subSize[] = "512M";
fileXioIoctl2(partFd, hddIO_ADD_SUB, subSize, strlen(subSize) + 1, NULL, 0);
```

Returns 0 on success, -1 times errno if an error occurred.

HDDIO_DELETE_END_SUB

'arg'	Must be NULL
'arglen'	Must be 0
'buf'	Must be NULL
'buflen'	Must be 0

Removes the last sub partition that was created from a main partition. Be warned, this will destroy a PFS filesystem which is occupying the main & subs.

Returns 0 on success, -1 times errno if an error occurred.

HDDIO_NUMBER_OF_SUBS

'arg'	Must be NULL
'arglen'	Must be 0
'buf'	Must be NULL
'buflen'	Must be 0

Gets the number of sub partitions attached to a main partition.

Returns the number of sub partitions.

HDDIO_GETSIZE

‘arg’	Pointer to an “int” which contains the “partition number”
‘arglen’	4
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the size of a main partition, or one of the sub partitions linked to it. If the partition number is 0, then the size of the main partition is retrieved. If the partition number is >= 1, then the size of sub partition which corresponds to the partition number – 1 is retrieved.

Returns the size of the specified partition, in units of sectors (512 byte units).

The PFS driver

The PFS driver provides support for using the Playstation Filesystem. The driver sits on top of the APA driver, so you must have loaded ps2dev9.irx, ps2atad.irx and hdd.irx before loading the PFS driver. As with the APA driver, all access is done through the fileXio file IO manager.

On loading the PFS driver you can pass arguments to control the memory consumption, efficiency and other parameters of the driver. The possible arguments are as follows:

-m <#>	- Maximum number of simultaneous mounts. In the current implementation of the driver, each mount consumes 324 bytes of memory.
-n <#>	- Number of buffer/caches to use. Increasing the number of buffers will generally increase the performance of the driver, at the expense of increased memory consumption. In the current driver, each buffer consumes 1052 bytes of memory. By default 8 buffers are used. You can specify up to 127 buffers to be used.
-o <#>	- Number of files which may be opened simultaneously. Each open consumes 1 or 2 buffers, and 560 bytes of memory.
-debug	- Enables debug output for the driver. Only really useful if you experience a problem with the driver and need to get an idea of what is going on.

Throughout the remainder of this chapter, usage of the PFS driver via fileXio is documented.

Structures

```
typedef struct {
        unsigned int    mode;
        unsigned int    attr;
        unsigned int    size;
        unsigned char   ctime[8];
        unsigned char   atime[8];
        unsigned char   mtime[8];
        unsigned int    hisize;
        unsigned int    private_0;
        unsigned int    private_1;
        unsigned int    private_2;
        unsigned int    private_3;
        unsigned int    private_4;
        unsigned int    private_5;
} iox_stat_t;
```

‘mode’	File mode (permissions, file type etc). See the macros in stat.h from ps2drv for more information.
‘size’	Lower 32 bits of the 64-bit file size

‘ctime’	File creation time.
	
	ctime[1] = seconds
	ctime[2] = minutes
	ctime[3] = hours
	ctime[4] = day
	ctime[5] = month
	ctime[6 & 7] = year (4 digits)

‘atime’	Last access time.
‘mtime’	Last modification time.
‘hisize’	Upper 32 bits of the 64-bit file size
‘private_0’	User ID (currently unused)
‘private_1’	Group ID (currently unused)
‘private_2’	The number of filesystem zones are reserved for the file.

```
typedef struct {
        iox_stat_t      stat;
        char            name[256];
        unsigned int    unknown;
} iox_dirent_t;
```

‘stat’	File status, as described above.
‘name’	The name of the file.

Functions

```
int fileXioFormat(const char *dev, const char *blockdev, const char *args, int arglen);
```

‘dev’	Must be “pfs:”
‘blockdev’	Partition identifier string of partition to be formatted. Must have been created in advance.
‘args’	Format parameters.
‘arglen’	Size of args.

Builds a new filesystem by formatting the specified partition with PFS. The zone size for the new filesystem must be specified in the format parameters. The zone size must be a power of 2 and in the range of 2kb to 128kb. You may also specify a fragment pattern, which will be applied to the zone bitmaps during formatting. This is really only useful for debugging, or if you would like to know how your program will perform with certain fragmentation.

Example only setting zone size:

```
int zoneSize = 8192;
fileXioFormat(“pfs:”, “hdd0:XXX”, &zoneSize, sizeof(int));
```

Example of setting fragment pattern:

```
int formatArg[3];
formatArg[0] = 8192;          // Zone size
formatArg[1] = 0x2dff;        // “-f”, enabled fragment pattern
formatArg[2] = 0xf0f0f0f0;    // Fragment bit pattern
fileXioFormat(“pfs:”, “hdd0:XXX”, formatArg, sizeof(formatArg));
```

The fragment pattern is repeated over the zone bitmaps during formatting. A binary 1 corresponds to a used zone and a binary 0 corresponds to a free zone. In the example above where 0xf0f0f0f0 is used, the bitmaps will be initialized such that four zones are used, four zones are unused, and so on.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioMount(const char* mountpoint, const char* blockdev, int flag);
```

‘mountpoint’	PFS logical mount device, ie “pfs0:”, “pfs1:”, “pfs2:” etc
‘blockdev’	Partition identifier string of partition to be mounted. Must have been created and formatted in advance.
‘flag’	Specifies the access mode for the mount (read-only or read/write). One of FIO_MT_RDWR or O_FIO_MT_RDONLY. You may also logical OR with PFS_MT_ROBUST to enable **robust mode.**

In order to access a filesystem it must first be mounted to a file IO device. Once it has been mounted you can operate on the filesystem using standard fileXio calls. The PFS driver supports mounting filesystems to file IO devices pfs0: through to pfs9:

The fileXioMount function mounts the “block device” specified by the partition identifier string blockdev to the file IO device specified by mountpoint. The filesystem is mounted either as read-only or read/write. If the filesystem is mounted as read-only then any attempt to modify the filesystem will fail.

Examples:

```
fileXioMount(“pfs0:”, “hdd0:Media”, O_RDONLY);
fileXioMount(“pfs1:”, “hdd0:Boot”, O_RDWR);
```

If **robust mode** is enabled for a mount, then the PFS driver will not keep any data cached and any changes to metadata will be written back to disk as soon as they occur. This will help minimize any risk of filesystem corruption occurring in the case of something like a power failure. However, the use of robust mode has a SEVERE impart on performance.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioOpen(const char *name, int flags, int modes);
```

‘name’	Filename of file
‘flags’	File access mode. One or more (via logical OR) of O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_EXCL
‘modes’	Initial file permissions (see macros in stat.h of ps2drv)

Creates a new file or opens an existing file. The filename is the name of the file plus the file IO device, for example: “pfs0:/somedir/somefile”

Returns the file descriptor (>0) on success, -1 times errno if an error occurred.

```
int fileXioClose(int fd);
```

‘fd’	File descriptor returned by fileXioOpen

Closes an opened file and frees the file descriptor.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioRead(int fd, unsigned char *buf, int size);
```

‘fd’	File descriptor returned by fileXioOpen
‘buf’	Buffer in EE memory where data will be read to.
‘size’	Requested amount of data to be read.

Reads at maximum size bytes from a previously opened file into the buffer pointed to by buf. Performance can be increased if you ensure that the buffer is aligned to a 64 byte boundary, and even more so if data is read in 512 byte units.

Returns the number of bytes read on success (0 is returned once the end of file has been reached). -1 times errno is returned if an error occurred.

```
int fileXioWrite(int fd, unsigned char *buf, int size);
```

Analogous to fileXioRead except for this function data is written from EE to the hdd rather than data read from the hdd to EE.

```
int fileXioLseek(int fd, long offset, int whence);
```

‘fd’	File descriptor returned by fileXioOpen
‘offset’	Distance to move file pointer.
‘whence’	One of SEEK_SET, SEEK_CUR, SEEK_END

The current file position is changed according to the offset and whence.

Returns the new file position on success, -1 times errno if an error occurred.

```
int fileXioRemove(const char *name);
```

‘name’	The filename (must include file IO device) of the file to be removed.

Deletes the specified file.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioRename(const char* source, const char* dest);
```

'source'	Source file/directory
'dest'	Destination file/directory

Renames a file or directory. If required, performs movement between directories.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioMkdir(const char *name, int mode);
```

'name'	Name of directory to be created (including file IO device)
'mode'	Directory permissions (same as for fileXioOpen)

Creates a new directory.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioRmdir(const char *name);
```

'name'	Name of directory to be removed (including file IO device)

Removes a directory. The directory must be empty, otherwise an error is returned.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioChdir(const char *name);
```

'name'	Name of directory to be used as the new "working directory" (including file IO device).

Changes the current working directory. With PFS, when a file is referenced it can either be referenced as an absolute path, or relative to the current directory. For example, if the current directory was "/foo/bar", then the following two open commands would open the same file:

```
fileXioOpen("pfs0:/foo/bar/document.dat"….);
fileXioOpen("pfs0:document.dat"…);
```

Returns 0 on success, -1 if an error occurred.

```
int fileXioDopen(const char *name);
```

‘name’	Name of directory to open (including file IO device)

Opens a directory. Once the directory has been opened you may retrieve the contents of the directory with calls to fileXioDread.

Returns the file descriptor (>0) on success, -1 times errno if an error occurred.

```
int fileXioDclose(int fd);
```

‘fd’	File descriptor returned by fileXioDopen

Closes a directory opened by fileXioDopen.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioDread(int fd, iox_dirent_t *dirent);
```

‘fd’	File descriptor returned by fileXioDopen
‘dirent’	Pointer to a iox_dirent_t structure which will be filled according to the next file in the directory.

In order to retrieve a list of all files/directories in a directory, you can iterate over this function. Each time this function is called, the next entry in the directory is read and the structure at dirent is filled.

Returns the length of the file/directory name (length of string in name member of iox_dirent_t), or 0 if the end of the partition list is reached (ie: when this returns 0, its time to stop the iteration). If an error occurred, returns -1 times errno.

```
int fileXioGetStat(const char *name, iox_stat_t *stat);
```

‘name’	File name (including file IO device)
‘stat’	Pointer to a iox_stat_t structure which will be filled according to the properties of the file.

Retrieves the properties/information for the file/directory specified by name and stores the information in the buffer pointed to by stat.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioChStat(const char *name, iox_stat_t *stat, int mask);
```

‘name’	File name (including file IO device)
‘stat’	Pointer to a iox_stat_t structure which will be used to alter the properties of the file.
‘mask’	Specifies which fields in the stat structure should be used to update the properties of the file. One or more (via logical OR) or the following may be specified: FIO_CST_MODE, FIO_CST_ATTR, FIO_CST_SIZE, FIO_CST_CT, FIO_CST_AT, FIO_CST_MT, FIO_CST_PRVT

Changes the properties of the file/directory specified by name. The properties specified by mask are updated using the contents of the stat structure.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioSync(const char *devname, int flag);
```

‘devname’	Mountpoint of filesystem to flush, ie “pfs0:”
‘flag’	Must be 0

The PFS driver buffers data in memory to avoid un-necessary writing to the hdd. This function flushes all the buffers back to the hdd. It also flushes the hdd’s internal cache.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioUmount(const char* mountpoint);
```

‘mountpoint’	Mountpoint of filesystem to be un-mounted.

Unmounts a filesystem. All data for the filesystem still in the cache is written back.

Returns 0 on success, -1 times errno if an error occurred.

```
int fileXioDevctl(const char *name, int cmd, void *arg, unsigned int arglen,
void *buf,unsigned int buflen);
```

'name'	Mount point (ie: "pfs0:", "pfs1:" etc)
'cmd'	For the PFS driver, one of: 1. PFSCTL_GET_ZONE_SIZE 2. PFSCTL_GET_ZONE_FREE 3. PFSCTL_CLOSE_ALL
'arg'	Command arguments. Depends on cmd.
'arglen'	Size of arg, in bytes.
'buf'	Buffer for data received from command. Depends on cmd.
'buflen'	Size of buf, in bytes.

This function performs special operations for a device (in this case, a mounted filesystem). See a subsequent section for details on each command.

Returns a command dependent value on success, or -1 times errno if an error occurred.

```
int fileXioIoctl2(int fd, int command, void *arg, unsigned int arglen, void *buf,
unsigned int buflen);
```

'fd'	The fd assigned to the file, returned by fileXioOpen
'cmd'	For the PFS driver, one of: 1. PFSIO_ALLOC 2. PFSIO_FREE
'arg'	Command arguments. Depends on cmd.
'arglen'	Size of arg, in bytes.
'buf'	Buffer for data received from command. Depends on cmd.
'buflen'	Size of buf, in bytes.

This function performs special operations on a file. See a subsequent section for details on each command.

Returns a command dependent value on success, or -1 times errno if an error occurred.

Devctl commands

PFSCTL_GET_ZONE_SIZE

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Gets the zone size in bytes.

PFSCTL_GET_ZONE_FREE

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Returns the number of free zones.

PFSCTL_CLOSE_ALL

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Closes all files on all mounted filesystems. However, file descriptors do not get freed so only use this when powering down the system.

Ioctl2 commands

PFSIO_ALLOC

‘arg’	Pointer to an “int” which holds the number of zones to allocate
‘arglen’	4
‘buf’	Must be NULL
‘buflen’	Must be 0

Attempts to allocate the number of zones specified by the argument for the file. This can help speed up writing, as discussed in the PFS overview.

Returns 0 on success, -1 times errno if an error occurred.

PFSIO_FREE

‘arg’	Must be NULL
‘arglen’	Must be 0
‘buf’	Must be NULL
‘buflen’	Must be 0

Frees any zones not currently being used by the file (in the event that it was truncated, etc).

Returns 0 on success, -1 times errno if an error occurred.