libfios2 Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Overview

File I/O Scheduler version 2 (FIOS2) is a C library you can use to schedule and manage I/O requests from multiple game components. Using FIOS2 helps make file access within a game more efficient and reliable by automatically resolving device contention and scheduling requests to optimize I/O performance.

FIOS2 is supported on PlayStation®Vita and Microsoft® Windows®. The Windows libraries are supplied as DLLs, with support for both 64-bit and 32-bit systems. The libraries are located in `host_tools\lib\` and `host_tools\lib.x64\`. In most cases, the FIOS API for the Windows® platform is identical to the API for the PlayStation®Vita platform. For APIs that are not common to both platforms, the *libfios2 Reference* identifies to which platform a specific API applies.

This document provides an overview of FIOS2 features, components, and operation. It describes the APIs and interfaces that games can use to access FIOS2 features and shows the basic steps required to implement a FIOS2-based solution. For detailed information about the FIOS2 APIs, see the *libfios2 Reference*.

FIOS2 Benefits

Like the original FIOS API, FIOS2 optimizes I/O request handling by:

- Intelligently scheduling I/O requests from different game components to prevent device contention from interrupting the flow of critical data.
- Optimizing I/O patterns to reduce seek time.
- Providing transparent access to archived and optionally compressed data. FIOS2 uses the compression libraries to provide direct access to compressed data. Using compression can speed up data transfer during game execution and reduce the storage footprint of saved files.

In addition to the deadline and priority assigned to each request, the scheduler takes into account other requests and how long it will take to complete the request. The completion time is estimated based on previous I/O performance, so it reflects real-world conditions on a specific machine.

FIOS2 introduces a finer-grained approach to processing I/O requests that reduces read latency and improves performance. In the original version of FIOS, large reads weren't broken up into smaller chunks – you had to wait for the entire read to complete. FIOS2 immediately decomposes each request into a collection of small reads or writes that can be scheduled independently. Not only does this eliminate the problem of high-priority requests having to wait for large lower-priority requests to complete, reads that access different media are split into chunks that can be scheduled in parallel, rather than sequentially.

With FIOS2, you don't need to supply a memory allocator. Everything is done with static buffers. Another change is that operations, file handles, and directory handles are all handles instead of pointers – you won't get the same handle twice and FIOS2 can always detect access to a stale handle.

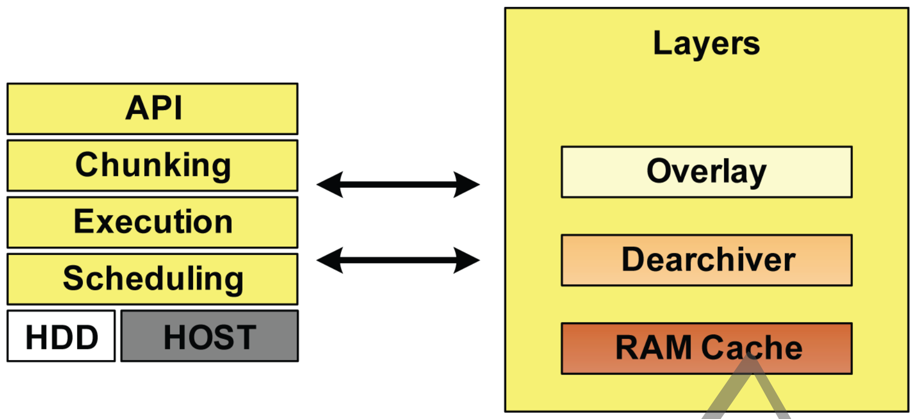
FIOS2 is provided as a C library, but a C++ compatibility layer enables original FIOS code to work without significant modification.

FIOS2 Architecture

FIOS2 has four tiers of operation:

- **API** – provides the client interface for submitting and managing I/O requests.
- **Chunking** – decomposes the incoming requests into a collection of small chunks that can be scheduled individually.
- **Execution** – applies the results of I/O filters and dispatches requests to the appropriate media schedulers.
- **Scheduling** – schedules the actual I/O requests for the underlying devices.

Figure 1 FIOS2 Architecture



In FIOS2, the scheduling process is pushed down to the bottom of the stack. Instead of being closely coupled with the API, the media schedulers sit just above the devices for which they're scheduling I/O.

Unlike the media filters in the original FIOS, overlays and the I/O filters are decoupled from the stack entirely. They are accessed as needed by the Chunking and Execution processes before the individual chunks are passed to the media schedulers.

Request Scheduling

When you submit an I/O request to FIOS2, the request is immediately split into a collection of smaller chunks. FIOS2 builds a dependency graph of these chunks so that they can be dispatched by the media schedulers for the underlying devices. This graph enables chunks that have no dependencies to be scheduled immediately, and chunks that access different media to be scheduled in parallel. However, this scheduling does not reorder existing requests in the queue.

When scheduling individual chunks, the media schedulers take into account three factors: deadline, priority, and data retrieval efficiency. You can use deadline and priority settings to lower the latency for specified operations, and yet still allow for efficient data retrieval patterns.

Deadline

A chunk's deadline is the most important factor in scheduling, so that immediate needs take precedence over less urgent ones. A chunk that cannot meet its deadline is generally scheduled ahead of chunks that will meet their deadlines. If multiple deadlines would be missed, the scheduler then considers their relative priorities.

You can set a deadline to any absolute `SceFiosTime` value (see Working with Time). You can also set a deadline to one of the following special values:

- 0 (the default deadline, which is current time plus 300 seconds)
- `SCE_FIOS_TIME_EARLIEST` (the soonest possible time)
- `SCE_FIOS_TIME_LATEST` (the latest possible time)

Priority

If multiple chunks will miss their deadlines, FIOS2 considers the specified priorities, with higher-priority requests taking precedence. Priority settings provide a way to ensure that important I/O requests are completed before less-important requests. When assigning a high priority to a request, you need to consider the game's overall I/O requirements, not just the needs of a particular thread.

You can set a priority to any value between -128 to +127. Negative values represent low priority, and positive values represent high priority. You can also set a priority to one of the following special values:

- `SCE_FIOS_PRIO_DEFAULT` (equal to 0)
- `SCE_FIOS_PRIO_MIN` (equal to -128)
- `SCE_FIOS_PRIO_MAX` (equal to +127)

Data Retrieval Efficiency

If both the deadline and priority requirements can be met, the FIOS2 scheduler adjusts the read order so that chunks that are in closer proximity to one another on storage media are read in sequence to reduce seek times.

Scheduler Example

Suppose that your game has a video and audio announcement that should play five seconds from now. Each part of the announcement is a separate file on the game disc. Because both the video and audio parts of the announcement should occur simultaneously, both file I/O operations are given the same deadline (which for this example is the current time plus 5 seconds). And because the game player is likely to notice audio glitches more than video ones, the audio file's I/O is given a higher priority to ensure that it plays smoothly. The following code shows the main features of this example:

```
// Set a common deadline to 5 seconds from now
SceFiosTime announceDeadline = sceFiosTimeRelativeSeconds(5);

// initialize the op for the video part
SceFiosOpAttr videoAttr = SCE_FIOS_OPATTR_INITIALIZER;
// set video part's deadline to the common deadline
videoAttr.deadline = announceDeadline;
// set the priority for the video part to default priority
videoAttr.priority = SCE_FIOS_PRIO_DEFAULT;
// open the video file
SceFiosOp videoOp = sceFiosFHOpen(&videoAttr, &video_fh, video_file,
                    &openParams);
// read the video file
videoOp = sceFiosFHRead(&videoAttr, video_fh, video_buffer, video_size);

// initialize the op for the audio part
SceFiosOpAttr audioAttr = SCE_FIOS_OPATTR_INITIALIZER;
// set the audio part's deadline to the common deadline
audioAttr.deadline = announceDeadline;
// set the priority for the audio part to maximum priority
audioAttr.priority = SCE_FIOS_PRIO_MAX;
// open the audio file
SceFiosOp audioOp = sceFiosFHOpen(&audioAttr, &audio_fh, audio_file,
                    &openParams);
// read the audio file
audioOp = sceFiosFHRead(&audioAttr, audio_fh, audio_buffer, audio_size);
```

In this example, FIOS2 decomposes each I/O operation into a number of chunks, and then schedules the chunks:

- All of the chunks have the same deadline.
- The chunks for the audio file have a higher priority than the chunks for the video file.
- If the open and read operations for both files can occur before the common deadline, then FIOS2 ignores their relative priorities, and schedules the requests to optimize data retrieval times.
- If the open and read operations for the audio file are predicted to miss their deadlines, then FIOS2 schedules the I/O for audio chunks before the I/O for video chunks, even if the deadline for the video I/O operations would then be missed.

Overlays and I/O Filters

Both overlays and I/O filters pre-process I/O requests to determine how to translate them into direct media-access. An I/O filter can also post-process the returned data, if needed.

Overlay

An overlay provides a way to remap filesystem accesses from one place to another. For example, downloaded patches can be overlaid over game data, or a folder of localized audio can be overlaid over the default game audio. When a request comes in, the overlay automatically checks its list of redirects and modifies the incoming path appropriately.

Dearchiver

The dearchiver I/O filter transparently handles requests for archived and optionally compressed data. It receives normal read requests and remaps them into a read from an archive as needed. Compressed data read from the storage device is automatically decompressed. Clients do not need to know or care that their data is compressed.

To create a dearchiver I/O filter, you call `sceFiosIOFilterAdd()` and specify `sceFiosIOFilterPsarcDearchiver` as the callback function.

RAM Cache

The RAM cache I/O filter caches data in memory. It can aggregate small reads into larger reads to make the game's disk access more efficient.

When a media request arrives for a piece of data, the RAM cache I/O filter checks to see if the data is stored in RAM. If it is, it returns the data immediately. If not, it reads a minimum of one cache block from the medium. Adjusting the size of the cache block allows a game to adjust the minimum size of its reads. Even though game consoles typically do not have much extra RAM to spare, a small RAM cache of 128KiB or so can greatly improve the performance of certain types of I/O requests.

To create a RAM cache I/O filter, you call `sceFiosIOFilterAdd()` and specify `sceFiosIOFilterCache` as the callback function.

Path Normalization

FIOS normalizes all incoming paths before passing them to overlays and I/O filters. When a path is normalized:

- Drive names such as `host0`: are preserved.
- After the drive name, a slash is prepended for all drives except host drives. For example, `app0:foo/file.txt` becomes `app0:/foo/file.txt`.
- For host access on PlayStation®Vita, paths without a Windows disk designator have any leading slash removed. For example, `host0:/foo/file.txt` becomes `host0:foo/file.txt`, but `host0:C:/foo/file.txt` is unchanged.
- Backslashes are converted to slashes. For example, `app0:\foo\file.txt` becomes `app0:/foo/file.txt`.
- Redundant slashes are removed. For example `app0:/foo//file.txt` becomes `app0:/foo/file.txt`.
- Trailing slashes are removed. For example, `app0:/foo/bar/` becomes `app0:/foo/bar`. This happens whether or not bar is a directory. The only situation where a trailing slash is left in place is the root directory, which is always represented by a single slash (`/`).
- Dot entries are removed. For example, `app0:/foo/./file.txt` becomes `app0:/foo/file.txt`.
- Dot-dot processing deletes one level from the path without checking to make sure that any directory ever existed there. For example, `app0:/foo/bar/../file.txt` is normalized into `app0:/foo/file.txt` whether or not the `/foo/bar` directory exists.

- Normalization fails if the output path or any intermediate path that is generated exceeds `SCE_FIOS_PATH_MAX` characters, including the terminating `NULL`.

So as an extreme example, the path `app0:foo\\/\./bar\\\/..///file.txt` would be normalized into `app0:/foo/file.txt`.

Profiling

To help with profiling and I/O performance analysis, FIOS2 provides both profiling (trace output) and profile callbacks.

Profiling can be helpful when you are trying to debug FIOS2 issues. Profiling has a relatively high overhead cost and can generate large amounts of tty output. Thus, profiling can have a significant impact on game performance. See the sample profile code and the `SceFiosParams` profiling option in the *libfios2 Reference*.

Profile callbacks have a low overhead and can be enabled with little impact on game performance. These callbacks are designed primarily for an in-game profiler, and they allow the profiler to display dynamic I/O information. The callbacks can also be used for debugging and tracing. See the sample code, as well as the `SceFiosParams` *`pProfileCallback`* and the `SceFiosProfileCallback` in the *libfios2 Reference*.

Embedding into a Program

Include `fios2.h` in the source program.

When building your program, link `libSceFios2_stub_weak.a` (the application does not have to load the PRX module for the FIOS2 library because this process is carried out automatically).

Notes for Creating an Application Package

The libfios2 library is a library that is installed in an application package. Therefore, libfios2.suprx included in the SDK must be copied to the app0:sce_module directory of the application package. For details, refer to the *libsysmodule Overview* document.

Sample Programs

The SDK provides the following sample programs that use the FIOS2 library:

sample_code/system/api_fios2/simple

This program is an example that shows the basic usage of the FIOS2 library.

sample_code/system/api_fios2/dearchiver

This program is an example that shows the FIOS2 dearchiver. This sample mounts a PSARC archive file, opens a file inside the archive, reads the file, and then closes the file and unmounts the archive.

sample_code/system/api_fios2/overlay

This program is an example that uses FIOS2 overlays. This sample uses a WRITABLE overlay. The sample creates a base file, then opens the file, reads from it, writes to it through the overlay, re-reads from the file (to verify that the write went to the overlay rather than the base file), and closes the file.

sample_code/system/api_fios2/profiling

This program is an example that shows FIOS2 profiling. This sample enables profiling, opens and reads a file, then closes the file and disables profiling.

sample_code/system/api_fios2/ramcache

This program is an example that shows the FIOS2 RAM cache. This sample shows how to create and use a RAM cache that is enabled below a dearchiver filter.

2 Using FIOS2 to Manage I/O Requests

To use FIOS2 to manage I/O requests, you need to:

(1) Initialize FIOS2.

(2) Submit I/O requests using the filehandle or path I/O APIs.

(3) Manage the operations associated with asynchronous requests. When you submit an asynchronous I/O request, an operation is created to manage the request and the I/O API returns a handle to the operation. You need to:

- Poll the operation to see if it's finished *or* set up a callback to be invoked when it completes.

Note: You can call only asynchronous operations from a callback thread.

- Read any data from the operation you might need, such as an error code.
- Delete the operation when you're finished. (This is mandatory so FIOS2 can reuse the operation for other requests.)

You can request information about current operations and modify operations by changing their priorities and deadlines. These changes cause FIOS2 to reevaluate how the operations are scheduled.

Note: If you use the synchronous APIs, the operations created for the request are managed for you.

(4) Terminate FIOS2 when you no longer need to submit I/O requests. Normally, FIOS2 resources are cleaned up automatically when the client terminates. (Note that you can suspend and reactivate FIOS2 at any time.)

The following sections describe these steps in more detail. For examples, see the sample code included with the FIOS2 release. For detailed API documentation, see the *libfios2 Reference*.

Initializing FIOS2

Before you can use FIOS2, you need to include `fios2.h`. The `fios2.h` header automatically includes the required FIOS2 headers. The key headers are:

- `fios2/fios2_api.h` – core FIOS2 APIs
- `fios2/fios2_debug.h` – debugging APIs
- `fios2/fios2_types.h` – structs, callbacks, enums, and so on
- `fios2/fios2_errors.h` – error codes

To initialize FIOS, you call `sceFiosInitialize()` and pass in a `SceFiosParams` struct that specifies the buffers FIOS2 can use to store operations, file and directory handles, and chunks. The `opStorage`, `fhStorage`, and `dhStorage` buffers control how many simultaneous `SceFiosOp`, `SceFiosFH`, and `SceFiosDH` objects can be created. The `chunkStorage` buffer is used by FIOS2 when breaking I/O requests into smaller pieces for execution. The maximum size allowable for each of these buffers is 512KiB.

Note: If you have existing FIOS code, you can include `fios2_compatibility.h` to use the FIOS2 compatibility layer.

Submitting I/O Requests

You can read and write data through FIOS2 using APIs that are very similar to other systems' I/O APIs. FIOS2 supports both asynchronous and synchronous I/O and provides two ways to access files when performing I/O operations. You can perform individual reads and writes using the path APIs, or use the filehandle APIs to explicitly open files and perform a series of read or write operations.

Using the asynchronous APIs allows the calling thread to continue with other processing while the I/O request is processed. You can either supply a callback that is invoked automatically when the operation finishes, or poll the operation to see if it is complete. The asynchronous APIs return a handle to the operation. You need to explicitly call `sceFiosOpDelete()` to clean up the operation when it completes. If you do not delete completed operations, you will quickly run out of free operations for new data requests.

Using the synchronous I/O APIs is simpler, but limits performance. When you use the synchronous APIs, FIOS2 still creates an I/O operation, but you don't need to manage it directly. The synchronous APIs block until the operation is complete and return a success or error code along with request information. The operation is cleaned up automatically when the API returns. You can specify that open, stat, and close synchronous API calls always run on the caller's thread without being enqueued, even if other ops are enqueued; set the `opFlags` to include `SCE_FIOS_OPFLAG_IMMED` for each API call that you want to run on the caller's thread. Note that for a close call, if any ops are using the file handle passed to the close, the close is enqueued for execution after dependent ops have completed (and thus does not run immediately).

Setting Operation Attributes

When you submit an I/O request with any of the FIOS2 APIs, you can specify a completion deadline and priority to influence I/O scheduling. When you're using the asynchronous I/O APIs, you can also specify a callback for FIOS2 to invoke when the operation completes. If no callback is specified, you'll need to poll the operation that's returned to determine when FIOS2 is done processing your request.

To set the operation attributes, you pass a pointer to a `SceFiosOpAttr` struct when you call an I/O read or write API. You can specify the following attributes with `SceFiosOpAttr`:

- Deadline for the operation to complete
- Callback function and the context for the callback
- Priority for the operation
- Flags to control the operation
- User tag and pointer for client use

Note: If you do not specify any operation attributes, FIOS2 uses a set of default attributes. You can read and change the global defaults with `sceFiosGetGlobalDefaultOpAttr()` and `sceFiosSetGlobalDefaultOpAttr()`.

You can use `sceFiosOpReschedule()` to reschedule an operation and `sceFiosOpRescheduleWithPriority()` to reschedule an operation or reset its priority.

Accessing Files with a Filehandle

The filehandle I/O APIs provide a way to open a file and perform a series of read or write operations. You call `sceFiosFHOpen()` to open the file and are responsible for closing it by calling `sceFiosFHClose()` when you are finished.

When you call `sceFiosFHOpen()`, FIOS2 returns a filehandle that contains the path to the file, the file's status, and the current read/write position within the file. The current position is specified as a byte offset from the beginning of the file. As data is read or written, the filehandle automatically updates the offset to reflect the position when the read or write operation completes.

When reading with the filehandle APIs, you specify the filehandle you want to read, the number of bytes to read, and a pointer to a buffer to receive the data. The `sceFiosFHPread()` API accepts an offset as an argument, and the `sceFiosFHRead()` API uses the filehandle's current offset.

When writing with the filehandle APIs, you specify the filehandle, the number of bytes to be written, and a pointer to a buffer from which to read the data being written. The `sceFiosFHPwrite()` interface accepts an offset as an argument, and the `sceFiosFHWrite()` API uses the filehandle's current offset.

You can explicitly change the current offset between I/O operations by calling `sceFiosFHSeek()`. The seek API enables you to specify a location within the file relative to the beginning of a file, the end of the file, or the current offset. You can get the current offset by calling `sceFiosFHTell()`.

When you're done accessing the file, you need to call `sceFiosFHClose()` to close the filehandle. You can call close as soon as you're done issuing I/O requests against the file – FIOS2 won't close the file and clean up the filehandle until all pending I/O is complete.

The filehandle APIs also support reading and writing using multiple destination and source buffers. For more information, see the *libfios2 Reference*.

Working with the Filesystem

FIOS2 provides filehandle APIs for retrieving a file's size, getting the file path, getting the current offset, and getting complete status information for a file:

- `sceFiosFHGetSize()` returns the size of the file in bytes.
- `sceFiosFHGetPath()` returns the path of the file associated with the filehandle.
- `sceFiosFHTell()` returns the file's current offset.
- `sceFiosFHStat()` returns the full set of status information for the file, including the modification date and size.

There are also directory handle (DH) APIs for working with directories. For more information, see the *libfios2 Reference*.

Accessing Files by Pathname

The path I/O APIs enable you to perform individual reads and writes without explicitly opening and closing a file. These APIs accept a path to a file's location relative to the underlying media. If necessary, FIOS2 opens and closes a filehandle automatically as part of the operation.

The path APIs provide both asynchronous and synchronous interfaces for reading and writing:

- `sceFiosFileRead()`
- `sceFiosFileWrite()`
- `sceFiosFileReadSync()`
- `sceFiosFileWriteSync()`

When reading with the path APIs, you specify the path of the file you want to read, the position where you want to start reading, the number of bytes to read, and a pointer to a buffer to receive the data.

When writing with the path APIs, you specify the file path, the position where you want to start writing, the number of bytes to be written, and a pointer to a buffer from which to read the data being written.

Working with the Filesystem

FIOS2 provides path APIs for retrieving a file's size, deleting a file, checking to see if a file or directory exists, and getting complete status information for a file or directory:

- `sceFiosFileGetSize()` returns the size of the specified file in bytes.
- `sceFiosFileDelete()` deletes the specified file.
- `sceFiosFileExists()` reports whether or not the specified file exists. It returns false if a directory exists instead of a file.
- `sceFiosDirectoryExists()` reports whether or not the specified directory exists. It returns false if a file exists instead of a directory.
- `sceFiosExists()` reports whether or not the specified item exists, regardless of whether it is a file or directory.
- `sceFiosStat()` reports a full set of file status information including modification date and whether the item is a file or directory.

Managing Operations

When you call one of the FIOS2 read or write APIs (either synchronous or asynchronous), FIOS2 splits the operation into chunks and enqueues the chunks according to their deadline and priority requirements. To "execute" each chunk, FIOS2 removes the chunk from the queue and passes it to the system API for I/O processing.

When you use the asynchronous FIOS2 APIs to submit I/O requests, they return a handle to the operation that is created to manage the request. While the request is in the FIOS2 queue, you can change the operation's deadline and priority, cancel the operation, or determine if the request is complete. After FIOS2 removes a particular chunk from the queue, all APIs that modify its state no longer have any effect. When the request completes and you are finished with the operation, you **must** delete the operation to return it to the free operation pool. (Otherwise, you'll run out of operations to submit new requests.)

All of the operation APIs are synchronous.

- `sceFiosOpCancel()` cancels an operation so that it will not execute. It has no effect if the operation has already executed or if the operation is being executed. This API does not delete the operation, so you must still delete the operation after canceling it. (Or use `sceFiosOpDelete()` instead, which both cancels and deletes the operation.)
- `sceFiosOpDelete()` deletes an operation. If the operation is currently executing, the thread will block until it can be deleted. (Note that this behavior is different from original FIOS, where deletion was asynchronous.)
- `sceFiosOpReschedule()` changes the operation's deadline. If the operation is already executing, this API has no effect.
- `sceFiosOpRescheduleWithPriority()` changes the operation's deadline and priority. If the operation is already executing, this API has no effect.
- `sceFiosGetAllOps()` returns an array of all of the outstanding operations. This enables you to iterate over all of the current operations and perform some action. For example, this might be useful when you need to clear out all operations related to one level of a game when game play moves to the next level.

Gathering Statistics for FIOS2

FIOS2 provides the following APIs to allow you to print and reset memory-usage statistics for various parts of the FIOS2 system:

- `sceFiosStatisticsReset()`
- `sceFiosStatisticsPrint()`

The statistics include the highest amount of memory used for each storage pool (the `highWater` value), which can help you specify an optimal size for the storage pools: run your game or test code, print the statistics after the run completes (or other appropriate time), and then adjust storage pool amounts, as needed, to keep the `highWater` values below the maximum allocated values.

Terminating FIOS2

When you are finished issuing FIOS2 I/O requests, you call `sceFiosTerminate()` to clean up and shut down.

3 Using Overlays to Redirect Reads and Writes

An overlay enables you to arbitrarily remap your filesystem. Overlays can be useful during development as well as in shipping titles. During development, you can use overlay redirects to swap in working copies of assets as you make changes or automatically load the latest versions of shared assets from a server. For example, a game may choose to allow files on host0: to override files on the internal storage according to their modification date.

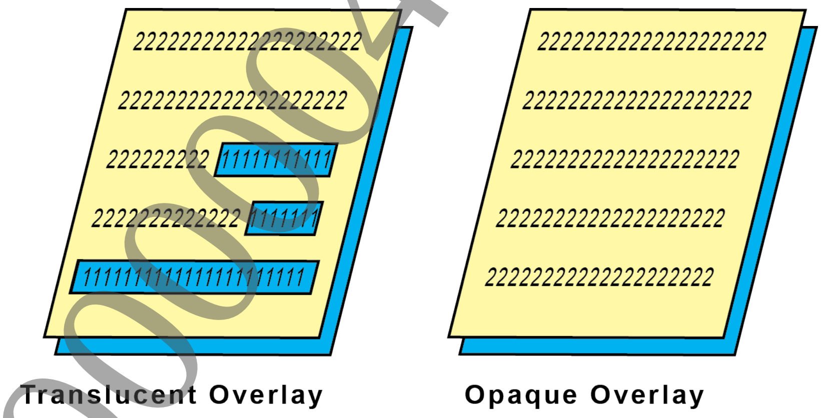
In a shipping title, overlay redirects can be used to manage downloadable content or support user-configurable options such as different UI skins. For example, you could set up a directory for downloadable content and use a path redirect to override the content provided in the game's main directory. The overlay mechanism can be used to support both new content and patches to existing content.

Overlay Types

FIOS2 has four types of overlays:

- **Opaque** – the files in the overlay are always used instead of the original files. The overlay totally replaces the overlaid path.
- **Translucent** – the overlay is checked first and the files in the overlay are used if a match is found. If a file doesn't exist in the overlay, the original path is checked.
- **Newer** – the files in the overlay are used if they have more recent modification dates than the files being overlaid.
- **Writable** – works just like a translucent overlay for read operations, but all writes are performed on the overlay. This is useful when you have a read-only source, or if multiple applications are sharing a directory and you want to redirect write operations to a different location.

Figure 2 Translucent and Opaque Overlays



Reading Files with Overlays

When reading files, a translucent or writable overlay acts like a piece of paper with cutouts. If the file exists in the overlay, it is read from the overlay. If it doesn't exist in the overlay, FIOS2 attempts to read it from the original path. An opaque overlay completely masks the original path – if the file doesn't exist in the overlay, it effectively doesn't exist. A newer overlay always guarantees that the newest version of the file will be read, regardless of whether it resides in the overlay or the underlying path.

Writing Files with Overlays

When writing files, the overlay is ignored unless an opaque or writable overlay is used. An opaque overlay completely masks the original path, so all reads and writes are always directed to the overlay.

A writable overlay captures and redirects all write operations, but reads can pass through to the original path if the file isn't found in the overlay.

Resolving Overlays and I/O Filters

Overlays are resolved as soon as you issue an I/O call with a path, so overlays are always resolved before I/O filters such as the dearchiver. When you call `sceFiosResolve()` or `sceFiosResolveSync()`, the path is resolved through all applied overlays and filters. If you just want to resolve the overlays, you can use `sceFiosOverlayResolveSync()`.

Adding an Overlay

When adding an overlay, you need to create an overlay description (`SceFiosOverlay`) that specifies:

- The type of overlay you want to create.
- The order in which the overlay should be applied.
- The destination path that you are redirecting (the original path).
- The source path for the overlay layer (the overlay path).

FIOS2 supports 128 user overlay layers ordered from 0 to 127. The overlay with an order of 0 is applied first.

Adding multiple redirect layers with the same order is allowed and might be useful in some cases. For example, if you wanted to overlay multiple patch folders on top of one another, you could apply a series of overlays with the same order. The last overlay added will be the first one checked.

Note: The overlay order is independent of the I/O filter order.

To add the overlay, you pass the overlay description to `sceFiosOverlayAdd()`, for example:

```
SceFiosOverlay overlay = SCE_FIOS_OVERLAY_INITIALIZER;
overlay.type = SCE_FIOS_OVERLAY_TYPE_TRANSLUCENT;
overlay.order = 0;
overlay.dst = "dst_path";
overlay.src = "src_path";
err = sceFiosOverlayAdd(&overlay, &id);
```

4 Using a Dearchiver

A dearchiver is an I/O filter that provides transparent access to data in PSARC archives. While FIOS2 supports adding up to 256 I/O filters, the dearchiver and RAM cache are currently the only types of filters supported, and you only need to add one dearchiver to be able to access data stored in PSARC archives. If you do have multiple I/O filters, they are applied in order. The filter in slot 0 is applied first, and the filter in slot 255 is applied last.

Note: Overlays are applied separately from I/O filters. The overlay order is independent of the I/O filter order.

Adding a Dearchiver

To create a dearchiver, you call `sceFiosIOFilterAdd()` and specify `sceFiosIOFilterPsarcDearchiver` as the callback function.

Creating a dearchiver requires a certain amount of memory to be passed in as a work buffer. The work buffer must be at least 192KiB with a 64-byte alignment. (These values are available as `SCE_FIOS_PSARC_DEARCHIVER_WORK_BUFFER_SIZE` and `SCE_FIOS_PSARC_DEARCHIVER_WORK_BUFFER_ALIGNMENT`.)

For example:

```
SceFiosPsarcDearchiverContext context =
    SCE_FIOS_PSARC_DEARCHIVER_CONTEXT_INITIALIZER;
context.pWorkBuffer = malloc(SCE_FIOS_PSARC_DEARCHIVER_WORK_BUFFER_SIZE);
context.workBufferSize = SCE_FIOS_PSARC_DEARCHIVER_WORK_BUFFER_SIZE;
err = sceFiosIOFilterAdd(0, sceFiosIOFilterPsarcDearchiver, &context);
```

The first parameter passed to the add function is the slot number where the filter is to be added. I/O filters are processed in order according to this value. If the add request fails, an error is returned.

Accessing Files

To enable access to the contents of a PSARC archive, you need to *mount* the archive. While the archive is mounted, its contents show up in the file system hierarchy at the mount point directory and you can use the standard FIOS2 APIs to access the content.

Mounting Archives

To mount an archive, you call `sceFiosArchiveMount()` or `sceFiosArchiveMountSync()`. Mounting archives requires a certain amount of memory to be passed in the `mountBuffer` argument. The memory requirement is always the same for a given archive, so it can be precomputed and stored elsewhere. If you are not precomputing the required memory, there are several ways to determine the amount needed:

- Call `sceFiosArchiveGetMountBufferSize()` or `sceFiosArchiveGetMountBufferSizeSync()`. This will open the archive, examine its contents, and then close the archive.
- Call `sceFiosArchiveMount()` with any smaller-than-necessary `mountBuffer` (including a NULL pointer) and let it complete with an error. The mount will fail with `SCE_FIOS_ERROR_BAD_SIZE`. Once the op has completed but before you delete it, you can determine the actual size required by calling `sceFiosOpGetActualCount()`. This will open the archive, examine its contents, and then close the archive.

- Call `sceFiosArchiveMount()` with any larger-than-necessary `mountBuffer`. The mount will succeed. Once the op has completed but before you delete it, you can determine the actual size used by calling `sceFiosOpGetActualCount()`. Any unused memory remaining in `mountBuffer` is available for re-use by the application.

The memory used in `mountBuffer` is not released until you unmount the archive by calling `sceFiosArchiveUnmount()` or `sceFiosArchiveUnmountSync()`.

Accessing the Contents of a Mounted Archive

You use the standard FIOS2 APIs to open, read, and close files in mounted archives, just like normal. FIOS2 automatically decompresses the data (if needed) when you call the read APIs. For more information about accessing files, see Submitting I/O Requests.

PSARC files use a default block size of 64KiB. For optimal performance when reading from compressed files, read using sizes modulo the block size (or larger), and read from offsets that are aligned to the block size boundary.

How Conflicts are Resolved when Multiple Archives are Mounted at the Same Location

You can mount more than one archive at the same location, or within the subdirectory of another mount point. If a particular path refers to a file in more than one archive, the file in the most-recently mounted archive is the one that will be selected.

One common case where you might want to mount multiple archives at the same location is for game patches. You could have a game archive mounted at a particular directory, and then mount one or more patch archives at the same location. The latest patch will always take precedence for any files that exist in more than one of the archives.

How Conflicts are Resolved when Content Already Exists at the Mount Point

By default, when archives are mounted, any previously existing files and directories at the mount point are hidden and inaccessible. In other words, the archive mount is opaque.

If you want to be able to access previously existing files, you can change the dearchiver's default behavior by passing `SCE_FIOS_PSARC_DEARCHIVER_MOUNT_TRANSLUCENT` in the flags field of `SceFiosPsarcDearchiverContext`. When this flag is set, files that already exist at the mount point will be visible and accessible. If a particular path refers to a previously existing file as well as a file in the mounted archive(s), the file in the most-recently mounted archive is the one that will be selected.

Keep in mind that setting the `SCE_FIOS_PSARC_DEARCHIVER_MOUNT_TRANSLUCENT` flag is less efficient than using opaque archive mounts because additional I/O has to be performed to check for previously existing files.

5 Using the RAM Cache to Improve Read I/O Performance

A RAM cache is an I/O filter that caches data from I/O reads in memory. It can aggregate small reads into larger reads to make the game's disk access more efficient (that is, it can reduce the number of system calls for a game that makes many small reads from large files). While FIOS2 supports adding up to 256 I/O filters, the dearchiver and RAM cache are currently the only types of filters supported. See the Using a Dearchiver chapter for information about the dearchiver. If you do have multiple I/O filters, they are applied in order. The filter in slot 0 is applied first, and the filter in slot 255 is applied last.

When a media request arrives for a piece of data, the RAM cache I/O filter checks to see if the data is stored in RAM. If it is, the RAM cache returns the data immediately. If not, the RAM cache reads a minimum of one cache block from the medium. Adjusting the size of the cache block allows a game to adjust the minimum size of its reads. Even though game consoles typically do not have much extra RAM to spare, a small RAM cache of 128KiB or so can greatly improve the performance of certain types of I/O requests.

Note: Overlays are applied separately from I/O filters. The overlay order is independent of the I/O filter order. However, the cache operation is based on paths, which are affected by overlays. If a RAM cache filter is added before overlays, the RAM cache might not perform as expected.

Adding a RAM Cache

To create a RAM cache I/O filter, you call `sceFiosIOFilterAdd()` and specify `sceFiosIOFilterCache` as the callback function.

Creating a RAM cache requires a certain amount of memory to be passed in as a work buffer. The work buffer must have an 8-byte alignment, for which you can use `SCE_FIOS_RAM_CACHE_BUFFER_ALIGNMENT`. However, not all of the work buffer associated with the cache is available for caching data. A small amount of the buffer is reserved for the cache context and block descriptor data. Use `SCE_FIOS_RAM_CACHE_BUFFER_SIZE_PER_BLOCK` to determine the work buffer memory requirements.

For example:

```
SceFiosParams params;
sceFiosIsInitialized(&params);
int blocks = 8;  // define 8 cache blocks
int blocksize = 128*1024;  // define each block as 128 KiB

...

SceFiosRamCacheContext rccontext = SCE_FIOS_RAM_CACHE_CONTEXT_INITIALIZER;
int cacheSize = SCE_FIOS_RAM_CACHE_BUFFER_SIZE_PER_BLOCK(blocks, blocksize,
                params.pathMax);
rccontext.pWorkBuffer = malloc(cacheSize);
rccontext.workBufferSize = cacheSize;
rccontext.blocksize = blocksize;
rccontext.pPath = "some/path";       // optional path associated with this cache
err = sceFiosIOFilterAdd(1, sceFiosIOFilterCache, &rccontext);
```

The first parameter passed to the add API is the slot number where the filter is to be added. I/O filters are processed in order according to this value; see Caching Archived Files. If the add request fails, an error is returned.

Managing Cache Behavior

By default the RAM cache applies to all file reads. Reading a large number of different files can have a detrimental effect on cache contents if the cache is not large enough or if the blocksize is too large. For example, if a 1.5MB cache has twenty-four 64KB blocks, and there is active I/O on more than 24 open files, then there might effectively never be any cached data, and cache fills will likely negatively affect performance. If you need to read from a large number of files, either increase the cache size (which increases the number of available cache blocks) or decrease the block size (so that more blocks fit within the cache). Note that smaller block sizes mean that less read data is cached for each file.

To manage cache behavior:

- Use paths to control what the cache applies to, down to an individual file. With multiple caches, the one with the most-qualified match applies. If multiple caches have equivalent paths (the same or NULL), only the first cache to be registered will be used. If two caches are registered, at least one of them must have a non-NULL path. The drawback to this approach is that a separate cache is then required for each path specification. However, because a cache is a filter, the number of caches is only limited by the number of possible filters (using a dearchiver filter reduces that number by one).
- Use OP flags to prevent individual reads from interacting with a cache. If the flag is set in the file open call, that flag applies to every read operation that uses the returned filehandle. Otherwise, you can control individual read operations with OP flags. Note that the flag is ignored for writes or deletes, which always invalidate cache entries for that file.

RAM Cache Limitations

The current implementation of the RAM cache I/O filter is optimized for sequential read requests and attempts to cache data that is likely to be accessed more than once. When a read request is larger than the cache blocksize, only the block that corresponds to the end of the read is cached. Also, if the end of a read aligns with the end of a cache block, that read’s data is not cached because the next request will most likely read the next block.

Because of the underlying kernel implementation, using a RAM cache below a dearchiver generally provides little to no performance improvements. Using a RAM cache with blocksizes of 64 KiB (or higher) can give performance increases for sequential reads with sizes smaller than 64 KiB, especially when used on top of a dearchiver.

The current implementation does not directly support prefilling the cache or caching the contents of an entire file, but you can emulate this functionality by issuing 1-byte reads at increasing cache-blocksize offsets over the entire file using `sceFiosFHPread()` or `sceFiosFHPreadSync()` requests. Because there is no way to lock data within the cache, the cache buffer must be large enough to accommodate the entire file (and thus avoid flushing cached data).

Caching Archived Files

I/O filters are processed in index-order, so filter order is important. Filter order determines whether you cache files within an archive or cache the archive itself.

To cache a file inside a PSARC archive, the RAM cache filter must be ahead of the dearchiver filter (use lower index). To cache the archive itself, the corresponding cache filter must be after the dearchiver filter (use higher index).

If two caches are registered, one with low index (before the dearchiver) and one with high index (after the dearchiver), and no path is specified for either archive, only one of the caches is actually used. Which cache is used (and therefore, what is cached), depends on which one was registered first (which might not provide the intended behavior). To cache an archive separately from the files that it contains, include a path for one of the caches, most likely the cache below the archive.

6 Working with Time

When requesting I/O from FIOS2, you specify a deadline for data delivery. Deadlines are specified using absolute time: that is, as an integer value such as 2376482367479 that expresses the number of ticks that have elapsed on the system clock since some arbitrary previous time.

Absolute time has no consistent relationship to standard units of time. The current system time is simply a value that increases monotonically at a consistent frequency. The frequency might change, so there's no way to know in advance how absolute time will relate to wall-clock time, such as seconds and milliseconds.

Because the underlying system clock frequency isn't known until runtime, FIOS2 clients need to be able to specify time in wall-clock units that aren't dependent on the system clock frequency. FIOS2 provides APIs to convert between a system's absolute time and wall-clock time. These APIs know the frequency of the underlying platform's clock for accurate conversion.

Getting the System Time

You can call `sceFiosTimeGetCurrent()` to get the current system absolute time. This provides a starting point for calculating relative times.

Converting Between Absolute and Wall-Clock Time

The time APIs support three types of time conversion:

- Absolute-to-wall-clock APIs convert an absolute time interval to a wall-clock time interval.
- Wall-clock-to-absolute APIs convert a wall-clock time interval to an absolute time interval.
- Relative-time APIs accept a wall-clock time interval, read the current absolute time, convert the interval to an absolute time interval, add the interval to the current time, then return the resulting absolute time. In other words, they calculate an absolute clock time relative to the time they are called, using an interval measured in wall-clock time units.

Using these APIs enables you to specify I/O request deadlines using wall-clock time units: 10 milliseconds from now, 5 seconds after another request's deadline, and so on.

Second Conversion APIs

These APIs convert between a value in seconds and a value in absolute system ticks:

- `sceFiosTimeIntervalToSeconds()` converts an interval measured in absolute system ticks to an interval measured in seconds.
- `sceFiosTimeIntervalFromSeconds()` converts an interval measured in seconds to an interval measured in absolute system ticks.
- `sceFiosTimeRelativeSeconds()` accepts an interval measured in seconds and returns an absolute system time that is that many seconds from the time the API is called.

Millisecond Conversion APIs

These APIs convert between a value in milliseconds and a value in absolute system ticks:

- `sceFiosTimeIntervalToMilliseconds()` converts an interval measured in absolute system ticks to an interval measured in milliseconds.
- `sceFiosTimeIntervalFromMilliseconds()` converts an interval measured in milliseconds to an interval measured in absolute system ticks.
- `sceFiosTimeRelativeMilliseconds()` accepts an interval measured in milliseconds and returns an absolute system time that is that many milliseconds from the time the API is called.

Microsecond Conversion APIs

These APIs convert between a value in microseconds and a value in absolute system ticks:

- `sceFiosTimeIntervalToMicroseconds()` converts an interval measured in absolute system ticks to an interval measured in microseconds.
- `sceFiosTimeIntervalFromMicroseconds()` converts an interval measured in microseconds to an interval measured in absolute system ticks.
- `sceFiosTimeRelativeMicroseconds()` accepts an interval measured in microseconds and returns an absolute system time that is that many microseconds from the time the API is called.

Nanosecond Conversion APIs

These APIs convert between a value in nanoseconds and a value in absolute system ticks:

- `sceFiosTimeIntervalToNanoseconds()` converts an interval measured in absolute system ticks to an interval measured in nanoseconds.
- `sceFiosTimeIntervalFromNanoseconds()` converts an interval measured in nanoseconds to an interval measured in absolute system ticks.
- `sceFiosTimeRelativeNanoseconds()` accepts an interval measured in nanoseconds and returns an absolute system time that is that many nanoseconds from the time the API is called.

Working with Dates

FIOS2 also provides several convenience APIs for retrieving and converting dates:

- `sceFiosDateGetCurrent()` returns the current date.
- `sceFiosDateFromComponents()` converts a `struct tm` to a `SceFiosDate`.
- `sceFiosDateToComponents()` converts a `SceFiosDate` to a `struct tm`.
- `sceFiosDateFromFILETIME()` converts a `FILETIME` into a `SceFiosDate`. This API is available on the Windows platform only.
- `sceFiosDateFromSceDateTime()` converts a `SceDateTime` to a `SceFiosDate`. This API is available on the PlayStation®Vita platform only.
- `sceFiosDateToSceDateTime()` converts a `SceFiosDate` to a `SceDateTime`. This API is available on the PlayStation®Vita platform only.