Audio Panning Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

Document serial number: 000004892117

About This Document

Purpose

This document provides an overview of the Audio Panning functionality and implementation. The tutorial provides multiple audio panning solutions, ranging from a basic implementation of a 2D panning algorithm to more complex 3D implementations. The most basic 2D implementation is lightweight and easy to integrate, and the most complex 3D implementations are for advanced users who desire superior audio quality at the expense of more resources. The implementation makes use of NGS as the sound synthesizer and FIOS2 to handle file access. The tutorial provides a reference for use in application design and development.

Audience

This document is intended for PlayStation®Vita Audio application developers implementing a panning solution in their applications.

Related Documentation

Refer to the following related documents, included in the SDK, for further information on the following areas:

- NGS: See the *NGS Overview* and *NGS Reference*.
- FIOS: See the *libfios2 Overview* and *libfios2 Reference*.
- PS Archiver: See the *PSP2PSARC User's Guide* for usage information.
- Razor: See *Performance Analysis and GPU Debugging*.
- Sulpha: See the *libsulpha Overview* and *libsulpha Reference*.

1 Audio Panning Tutorial

Overview

The Audio Panning Tutorial sample shows how to implement an audio panning solution. The various options are provided to meet the needs of various developers, including those concerned with achieving high audio quality at the expense of resources, and other developers limited by resources.

The sample includes performance measurement features to allow the user to evaluate the different options. It is structured using functional modules to facilitate integration in an independent manner.

Main Application and Audio Panning Class

The main application is derived from the sample skeleton and makes use of the common sample utilities to configure the sample graphics and display layout.

The Audio Panning class is the main application control class. It provides features for system setup and allows the user to select the desired panning mode and apply the various panning effects. When the application is launched, all the modules required to run the sample are initialized, as is the default panning algorithm. The user can then interact with the interface buttons to control and configure the sample.

Panning Module

The Panning module provides various methods to generate panning effects on the audio being played. It provides a basic interface to configure the audio source and listener positions and calculate the NGS patch volumes for each channel.

NGS Streamer

This module allows you to instantiate audio streamers independently, regardless of the supported audio format, and provides a basic interface to initialize the streamer to play, seek, and loop sections of an audio file.

The NGS Streamer makes use of the NGS AT9 Streamer and NGS PCM Streamer modules to handle the details related to the various audio formats. The modules can be used independently. The NGS Streamer operates as an abstraction layer, where the user can stream any type of supported audio file, which is configured automatically regardless of type.

The system requires the user to initialize and set up the desired NGS configuration and FIOS. Multiple streamers can then be created to play the desired audio file through NGS voices.

FIOS Handler

The File I/O Scheduler library can be used to schedule and manage I/O requests efficiently. The FIOS handler provides a wrapper to simplify initialization and usage of the library to access audio files. It uses the PS Archiver to read data from archives with multiple files. Note this may be preferable in order to transparently support access to multiple data types; however, there is no benefit to compressing already compressed data (such as AT9 files), and it is recommended that you use a specialized audio compressor.

NGS System Helper

The NGS System Helper is a module used to initialize and set up NGS, and to configure the audio graph and routings.

Sulpha Common

Sulpha Common is a module used to initialize and set up Sulpha, which is used for debugging. Sulpha allows capture and analysis of audio debug information at run time and can be used to easily track errors and for performance analysis.

NGS Common

The NGS Common is a group of helper functions used to initialize and set up the audio system. It contains methods for audio output configuration and debugging.

Graphics

The Mesh and Shader Utils helper classes provide utilities for implementing the graphical interface of the sample. It is not intended as sample code in this tutorial and is not described in this document.

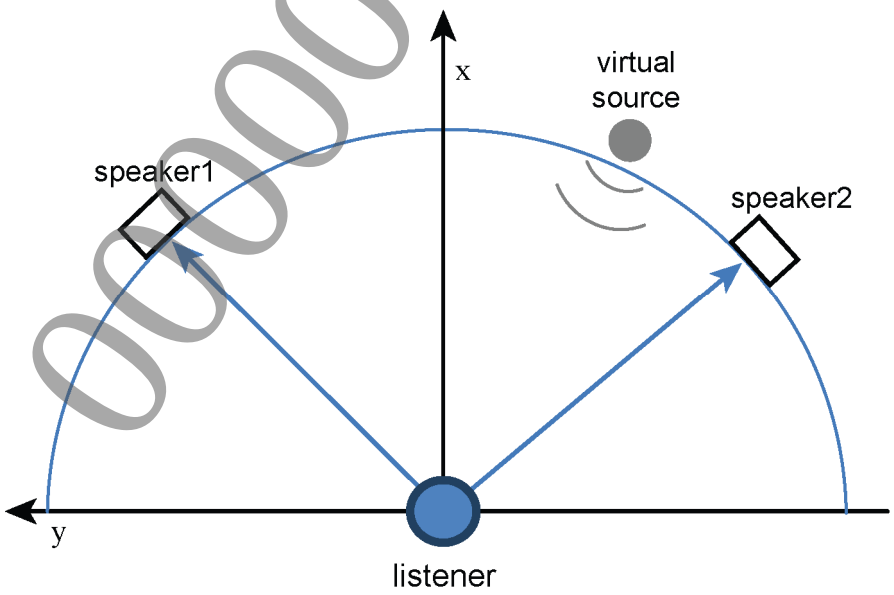
Panning Modes

There are three panning modes currently supported in the sample, and are described in the following sections. The interface switches between them at runtime so they can be compared in terms of performance and quality. The distance between the source and listener can also be adjusted to analyze the effect in each mode.

Simple 2D Panning

This mode presents a two-dimensional amplitude panning calculation, based on the source spherical position relative to the listener. The method is an approximation of real-source localization. Panning is calculated based on the listener's perception when the sound is generated from two speakers with controlled signal levels. The speakers are positioned symmetrically in front of the listener (the default is 45° from the center to simplify calculations). The direction of the virtual source is controlled by varying the amplitude of the two channels, and the volumes are scaled to ensure constant loudness at a certain distance.

Figure 1



The algorithm does not differentiate between sounds coming from the front or the rear, and could potentially be enhanced with a head-related transfer function (HRTF) implementation, which can be computationally expensive.

3D Position-based Panning

In this mode the panning is calculated based on the signal amplitude dependency on the distance to the two virtual speakers, located in the same position as in the previous case (45° from the listener in the XY-plane).

The algorithm receives the spherical positions of the listener and the source, and calculates the left and right channel volumes, making use of the distance, azimuth, and elevation angles.

3D Vector-based Panning

The vector-based panning mode is a 3D panning model. It uses the listener position matrix and source vector to calculate the volumes to feed into each stereo speaker, in order to simulate the panning effect. The algorithm calculates the source vector position in the listener space and the pan from that vector. The volumes are then scaled to reflect the amplitude decrease (also known as roll off) with the distance.

Application

The application provides a basic display to aid visualization of the source position relative to the listener, along with usage and debugging information.

The menu at the top left of the screen allows you to change the panning mode and modify the distance from the source to the listener at runtime. Use the directional buttons to browse through the menu (up and down) and change the settings (left and right). The left stick can be used to change the position of the source within the sphere equidistant from the listener.

Figure 2

Interface

- MENU
  - UP button: Selects the previous menu item
  - DOWN button: Selects the next menu item
  - RIGHT button: Increments the value of selected menu item or changes it to a different string (panning mode)
  - LEFT button: Decrements the value of selected menu item or changes it to a different string (panning mode)
- Left analog stick: Rotates the audio source
- TRIANGLE button: Show/Hide Debug Information

Usage

(1) Build the sample in `\target\samples\sample_code\audio_video\tutorial_audio_panning`.
(2) Launch `tutorial_audio_panning.self`.
(3) Configure the sample from the menu, using the buttons to control the application.

Configuration

The sample is configurable so it can be customized to suit developer requirements. The following sections contain configuration details for each module:

(1) Audio Panning Application
(2) Panning module
(3) Streamers
(4) NGS system
(5) FIOS/PS Archiver

The sample also provides the possibility to use Sulpha or Razor for the purpose of debugging and performance analysis.

Sample Configuration

Number of Voices

The number of NGS voices that can be used in the sample can be configured in `audio_panning.h`. By default, only one AT9 voice is used, but this can be modified to customize the number and type of sources and create different panning effects for each one.

```
#define NGS_AT9_VOICES          (1)
#define NGS_PCM_VOICES          (0)
#define NGS_ADPCM_VOICES        (0)
```

Audio Files to Stream

The name of the files to be played is configured in the `AudioPanning` constructor (defined in `audio_panning.cpp`).

```
m_pFiles[0] = MOUNTPOINT "/music.at9";
```

Note that the number of entries in `m_pFiles` is designed to match the number of voices (`NGS_SOURCE_VOICES`). However, the sample may be modified so the same streamer is reusable for streaming multiple files of the same type.

Audio Output Mode

The sample can be configured in `audio_panning.cpp` to play the resulting audio or write the output to an audio file. Define `OUTPUT_MODE` as `NGS_SEND_TO_DEVICE` or `NGS_WRITE_TO_FILE` for those purposes. By default `OUTPUT_MODE` is `NGS_SEND_TO_DEVICE`.

```
#define OUTPUT_MODE          (NGS_SEND_TO_DEVICE)
#define OUTPUT_FILE_NAME     ("app0:out.raw")
```

Actions

The panning effect can be configured at run time using the sample's menu. See the "Interface" section for more details.

Figure 3

By default, the simplest 2D panning mode is in use and the source is relatively close to the listener at 45° elevation and azimuth; the user can change this configuration at runtime to compare the output and analyze the performance impact.

Panning Configuration

Source Distance

The distance from the source to the listener can be configured in `panning.h`. The distance varies from `PANNING_MIN_SOURCE_DISTANCE` to `PANNING_MAX_SOURCE_DISTANCE` in the sample, and the user can modify it at runtime in `PANNING_DISTANCE_STEP` increment steps.
`PANNING_DEFAULT_SOURCE_DISTANCE` is the default distance for the source location when the sample starts.

```
#define PANNING_DEFAULT_SOURCE_DISTANCE      (10.0f)
#define PANNING_MIN_SOURCE_DISTANCE          (1.0f)
#define PANNING_MAX_SOURCE_DISTANCE          (150.0f)
#define PANNING_DISTANCE_STEP                (5.0f)
```

Virtual Speakers Distance

The distance at which the virtual speakers are located from the listener, used in the 3D Position panning mode, can be configured in `panning.h`. By default, they are positioned close to the listener, but this can be scaled to achieve different characteristics or improved separation, based on the source configuration. This may affect the audio quality at different distance ranges.

```
#define PANNING_SPEAKERS_DISTANCE            (1.0f)
```

Streamers Configuration

The streamers can be configured by modifying the definitions in `ngs_streamer_config.h`.

Note that some restrictions apply to ensure efficient operation.

Number of Streaming Buffers

`NUM_STREAM_BUFFERS` is the number of streaming buffers used by each of the streamers. Default: 2.

```
#define NUM_STREAM_BUFFERS          (2)
```

Streaming Buffer Sizes

`DEFAULT_STREAM_BUFFER_SIZE` and `DEFAULT_AT9_STREAM_BUFFER_SIZE` are the sizes of the PCM and AT9 streaming buffers, respectively, in KB. Default: 8 KB.

```
#define DEFAULT_STREAM_BUFFER_SIZE          (8 * 1024)
#define DEFAULT_AT9_STREAM_BUFFER_SIZE      (8 * 1024)
```

In order to configure this parameter to ensure the streamers' correct operation and performance, note the following restrictions. Some of the restrictions, such as minimum buffer size and buffer alignment, are specific to the audio type and are described in separate below.

The buffer sizes affect the performance of the FIOS system. Using small buffers has a negative impact on FIOS performance, especially if the PS archiver is in use (the default in this sample).

Optimal buffer sizes balance memory usage with data access performance.

AT9 Streaming Buffer Sizes

(1) Minimum Buffer Size

In order to avoid data starvation in the system, the minimum recommended buffer size (in bytes) is:

```
(((nGranularity * numChannels * MAX_PITCH_RATIO / AT9_SAMPLES_PER_PACKET)
* packetEncodedSize) + packetEncodedSize
```

where:

- `nGranularity` – system granularity
- `numChannels` – number of audio channels in this voice
- `MAX_PITCH_RATIO` – the maximum value is 4 (output sample rate = 48kHz, input sample rate < 192kHz). It can be changed to a smaller value (for example, if you are only playing music at 48kHz, `MAX_PITCH_RATIO` can be set to 1).
- `AT9_SAMPLES_PER_PACKET` – samples per packet
- `packetEncodedSize` – AT9 packet encoded size (`nBlockAlign` in '`fmt`' chunk)
- (`+ packetEncodedSize`) is added to ensure there is always enough data for the look ahead

Note this is the recommended minimum size as it ensures every buffer contains enough data to be processed per update. It is possible to use a higher number of buffers of smaller size; however, only one callback is generated by the system per update, and it is the developer's responsibility to ensure all the processed buffers are filled with more data in time to avoid data starvation.

(2) Buffer Alignment

Ensure the streaming buffers are always aligned to the beginning of a packet or superpacket, depending on whether the superframe mode is OFF or ON. This simplifies the operation when using `nSamplesDiscardStart`/`nSamplesDiscardEnd`.

In order to avoid decode errors:

(a) When using `nSamplesDiscardStart`, ensure the buffer is always aligned to the beginning of a superpacket.

(b) When using `nSamplesDiscardEnd`, ensure the buffer is aligned to both the beginning and end of a superpacket.

PCM/ADPCM Streaming Buffer Sizes

(1) Minimum Buffer Size

For PCM and ADPCM formats, ensure there is enough data in the buffers to avoid data starvation in the system at all times.

(2) Buffer Alignment

Align ADPCM buffers to the encoded block size (`VAG_FILE_DATA_BLOCK_SIZE`) to ensure that full blocks are always available for decoding.

Looping Configuration

Use the `NGS_STREAMER_USE_HEADER_LOOPING_INFO` flag to determine whether the looping information is read from the file header or if the user prefers to manually configure it when calling `sceNgsStreamerSetLoop()`. This is only supported for AT9 and PCM types. If the flag is passed in `nUseHeaderInfo` when calling this function, the loop start and loop end parameters are

configured using the header information. The function defaults to the user-defined parameters if the header does not contain this information.

NGS Configuration

Configure NGS settings in `ngs_config.h`.

Number of Modules

`NUM_MODULES` defines how many unique module types NGS allows to be loaded. This value refers to the number of module types, regardless of the number of instances. It is set to 15 by default, as this is the current maximum number of available modules.

```
#define NUM_MODULES                (15)
```

System Granularity

`SYS_GRANULARITY` defines the PCM sample granularity. NGS processes and outputs PCM sample packets of this size, for every channel, with each update.

```
#define SYS_GRANULARITY           (512)
```

Sample Rate

`SYS_SAMPLE_RATE` defines the sample rate for NGS.

```
#define SYS_SAMPLE_RATE         (48000)
```

Voices Number of Channels

`NGS_VOICES_NUM_CHANNELS` is the default number of channels defined for the NGS voices.

```
#define NGS_VOICES_NUM_CHANNELS    (2)
```

Number of Racks

`NGS_NUM_RACKS` is the default number of racks in the system. By default there are 3: the master rack, the PCM player rack, and the AT9 player rack.

```
#define NGS_NUM_RACKS              (3)
```

Number of Voices

`NGS_NUM_VOICES` is the default number of voices in the system. By default there are 3: the master voice, the PCM player voice, and the AT9 player voice.

```
#define NGS_NUM_VOICES             (3)
```

Default Volume

`NGS_DEFAULT_VOLUME` is the default volume set for the patches. The default is 1.0f.

```
#define NGS_DEFAULT_VOLUME        (1.0f)
```

Maximum Pitch Ratio

`MAX_PITCH_RATIO` is the maximum Pitch Ratio supported in the system. The default is 4 (output sample rate = 48kHz, input sample rate < 192kHz). It can be changed to a smaller value (for example, if you are only playing music at 48kHz, set `MAX_PITCH_RATIO` to 1).

```
#define MAX_PITCH_RATIO            (4)
```

FIOS Configuration

The FIOS module is initialized in the `fiosHandlerInit()` function, where memory allocation and various other system parameters are configured. See the *libfios2 Overview* and *libfios2 Reference* for more information.

The FIOS handler module uses the PS archiver by default, but this feature can be switched off by disabling the flag `FIOS_PSARCHIVER_ENABLED` in `fios_handler.h`, which results in improved data reading performance, especially if archiver compression is in use.

```
#define FIOS_PSARCHIVER_ENABLED     (1)
```

PS Archiver Usage

The sample reads data from the `app0:/archive_data.psarc` archive file by default (`FIOS_PSARCHIVER_ENABLED` is on), which is copied from the `/data` directory to `app0` when building the sample.

The archive included in the sample (in the `/data` folder) has been created with the PSP2PSARC tool with no compression. For more information on the format of the file and how to create your own, see *PSP2PSARC User's Guide*.

Sulpha Usage

To use Sulpha, enable the flags described below, either in the `sulpha_common.h` header file or as additional preprocessor definitions in the sample project.

The data captured can be interpreted by the Sulpha tool, which is included in the SDK. For more information, see the *libsulpha Overview* and *libsulpha Reference*.

Sulpha Enabling

The Sulpha functionality can be enabled with the `SULPHA_ENABLED` flag. The global Sulpha Capture enabling flag enables debugging data capture to be interpreted by the Sulpha tool. Output data is either sent to Sulpha in real time or to an output file, depending on the value of `SULPHA_LIVE_ENABLED`. The default is 0 (disabled).

Sulpha Mode

The `SULPHA_LIVE_ENABLED` flag enables or disables real-time capture in the Sulpha Tool. If it is enabled (`SULPHA_LIVE_ENABLED=1`), the debug information is visible in the Sulpha Tool at runtime. If it is disabled (`SULPHA_LIVE_ENABLED=0`), the debug data is saved in an output file for later use. The default is 0 (disabled).

Sulpha Capture File Name

If `SULPHA_LIVE_ENABLED` is 0, the module saves the Sulpha debug information to an output file defined as `CAPTURE_FILE_NAME`. By default, the file is saved to `app0:` and is named `out.sul`.

```
#define CAPTURE_FILE_NAME           ("app0:out.sul")
```

Razor Usage

Razor enables performance data capture of the application at run-time. To enable Razor support in the sample, add `ENABLE_RAZOR_CAPTURE` to the list of preprocessor definitions in the sample project and rebuild. If the application is then loaded from the Razor plugin, the user will be able to capture performance data. A default marker has been configured to measure performance of the different panning solutions.

2 Audio Panning Source Code Overview

The Audio Panning sample source code is split into the modules described in this section.

Audio Panning Main Code

The application entry point is in `main.cpp`. The `AudioPanning` class (derived from `SampleSkeleton`) contains the high-level application code (`audio_panning.h` and `audio_panning.cpp`). Key functions are described in the following sections.

main()

The application entry point. This function initializes the application using the `init()` function, calls the `update()` and `render()` functions, which contain the main application code, and deallocates the system using `shutdown()`.

If Razor is enabled in the sample, the module is also loaded from here and used by the various modules in the sample.

init()

This function sets up the application by loading the necessary modules and initializing the system.

The first initializations begin in the base class (`SampleSkeleton::init()`), from which the `AudioPanning` class derives, and the sample utilities and graphical interface used by the sample.

NGS is then initialized (`initNGS()`), as well as the audio graph, making use of the NGS system helper functions (including `createRack()` and `connectRacks()`), to create the desired components and routings. The FIOS handler initializer (`fiosHandlerInit()`) is also invoked at this stage to set up the FIOS module for controlling file I/O.

The function then configures one streamer per audio file required by the sample, and prepares the audio output to either play the resulting audio or write it to an audio file.

The panning module is also initialized at this stage, providing the start-up volumes needed to configure the NGS patches.

Finally, the audio update and audio data threads are started. The audio update thread handles NGS updates and audio output. The audio data thread is used to handle NGS Player callbacks and read audio data into the relevant buffers.

update()

The main processing consists of an application loop that is terminated by pressing the PS button. The update function performs the following tasks:

(1) Wakes up the Audio Data processing thread if there is any pending player callback to be processed.

(2) Handles user configuration input, updating the position of the source, as well as the distance to the listener and the panning mode as required.

(3) Recalculates the panning if there has been any change that needs to be reflected.

(4) Updates the volumes on the NGS patches accordingly.

render()

Calls in the main processing loop responsible for the graphics rendering.

shutdown()

Shuts down the application by releasing the resources allocated in the `init()` function and unloading the relevant system modules. `SampleSkeleton::shutdown()` deallocates the base class components and must be called at the end of this function.

Panning Module

The Panning module is defined in `panning.h` and `panning.c`. The module provides a simple interface to calculate the volumes that can be set to the stereo channels in the patch. The current solutions do not require any additional initialization, but the interface may be expanded in the future with extra modes.

The current interface is described below:

calculatePanning()

The function receives information about the source and listener positions in various formats, and returns the left and right channel volumes used to configure the NGS patches to generate the appropriate panning effect. Not all the input parameters are needed for each solution.

The Simple 2D Panning mode uses the source position in spherical coordinates only. This position is relative to the listener.

The 3D Position mode receives the source and listener position in spherical coordinates, and calculates the panning based on the distance between them.

The 3D Vector mode receives the listener position matrix and source vector and reuses this information, typically available in the application for graphics calculations, to calculate the position vector relative to the listener and provide the panning characterization.

NGS Streamer

The Streamer module is defined in `ngs_streamer.h` and `ngs_streamer.c`. The module provides a simple interface to stream any supported audio file (AT9, PCM or VAG), given the name of the file and the NGS voice handle to stream it. It also provides an interface to set up the streamer to loop and seek within the audio file.

The main functions are described below:

sceNgsStreamerInit()

Streamer initialization function. It initializes the voice and resources needed.

sceNgsStreamerSetLoop()

Streamer function to set up an audio loop and initialize the streamer to play the different sections in the file a specified number of times. Only one loop is supported at any given time, and calling this function multiple times will overwrite the looping information.

Looping can be configured based either on the parameters (`nLoopStart` and `nLoopEnd`) if `nUseHeaderInfo` is 0, or the file's header looping information (only supported for AT9 and PCM) if the flag `NGS_STREAMER_USE_HEADER_LOOPING_INFO` is passed in nUseHeaderInfo.

sceNgsStreamerSeek()

Streamer function to seek a specified section to play when the user calls `sceNgsStreamerPlay()`.The function receives an offset and the number of samples, and configures the streamer to play only this section.

sceNgsStreamerPlay()

Streamer playing start function. It fills the streaming buffers with data and sets up the voice to start playing. Note that the user must configure the streamer to play the desired section, even if it is the complete file, by calling `sceNgsStreamerSeek()` or `sceNgsStreamerSetLoop()` prior to playing.

sceNgsStreamerStop()

Streamer playing stop function. It stops the playback and kills the voice.

sceNgsStreamerRelease()

Streamer release function. It releases the resources allocated in `sceNgsStreamerInit()`.

sceNgsStreamerHandlePlayerCallback()

Streamer function called to handle the NGS player callbacks. The function updates the streamer status and fills in the streaming buffers with more data if necessary.

This function is provided to allow the user to move the data reading to a separate thread and avoid blocking the callback generator thread. The data reading can be slow depending on the configuration, so this allows for improved performance optimization.

AT9 and PCM NGS Streamers

The NGS Streamer module uses the AT9 NGS Streamer (see `ngs_at9_streamer.h` and `ngs_at9_streamer.c`) and the PCM NGS Streamer (see `ngs_pcm_streamer.h` and `ngs_pcm_streamer.c`) to handle the operation of AT9 streaming and PCM/ADPCM streaming, respectively. The interface of these two modules is similar to the NGS Streamer itself, as this is just a wrapper to provide transparent handling of any audio type.

NGS System Helper

The group of helper functions in `ngs_system_helper.h` and `ngs_system_helper.c` are designed to facilitate the NGS system initialization and setup. It provides functions to initialize NGS (`initNGS()`), create and connect racks (`createRack()` and `connectRacks()`) and set volumes in a specified patch (`setPatchVolume()`).

NGS Common

The helper functions in `ngs_common.h` and `ngs_common.c` include audio output handling functionality (`prepareAudioOut()`, `writeAudioOut()` and `shutdownAudioOut()`) and other debugging utilities, such as Razor performance counter initialization (`threadPerInit()`) and NGS debug information (`printParamError()`).

FIOS Handler

The FIOS handling module is defined in `fios_handler.h` and `fios_handler.c`. The module provides a wrapper around the FIOS library, with a simple interface to initialize and control the system.

The main functions are described below:

fiosHandlerInit()

FIOS initialization function. It initializes the FIOS system and resources needed for its correct operation.

fiosHandlerTerminate()

FIOS deallocation function. It deallocates the FIOS system and resources allocated in `fiosHandlerInit()`.

fiosHandlerOpen()

This function opens a file in the FIOS system. It receives the name of the file to be opened and returns a file handle.

fiosHandlerClose()

This function closes a file in the FIOS system, using the associated file handle.

fiosHandlerReadSync()

This function reads synchronously from the specified file.

By default the FIOS system is configured to read at the maximum priority and to deliver the data at the earliest time. However, this can be modified to suit user's requirements for the different I/O accesses.

fiosHandlerReadAsync()

This function starts an asynchronous read from the specified file. Note that the data will not be available in the relevant buffer until the callback `fiosHandlerReadCallback()` is received.

By default, the FIOS system is configured to read at the maximum priority and to deliver the data at the earliest time. However, this can be modified to suit user requirements for I/O access.

Sulpha Common

The Sulpha handling functions in `sulpha_common.h` and `sulpha_common.c` provide an interface to initialize (`sulphaTracingStart()`) and un-initialize (`sulphaTracingStop()`) Sulpha,includes an update function (`sulphaTracingUpdate()`) to regularly update the system, and provides message tracing functionality (`sulphaTracingMessage()`) to facilitate debugging.