



# *ASIO 2.3*

*Audio Streaming Input Output  
Development Kit*

*Documentation Release #3*

**Steinberg Audio Stream I/O API**

(c) 1997-2013, Steinberg Media Technologies GmbH

**ASIO Interface Specification v 2.3****Contents****I. Overview**

1. Introduction
2. The Goal
3. The Design

**II. Implementation Guide**

1. Instantiation
2. Operation
3. Driver Query by the Host Application
4. Host Query by the Driver
5. Audio Streaming
6. Media Synchronisation (Sampleposition and System Time)
7. Driver Notifications to the Host Application

**III. Function Reference**

1. Initialization/Termination
2. Start/Stop
3. Inquiry methods and sample rate
4. Buffer preparation
5. Miscellaneous
6. Callbacks
7. Type definitions

**IV. Host Utility API Reference**

1. AsioDrivers Class

**V. Appendix**

- A. Using the bufferSwitchTimeInfo() callback
- B. Latency vs. Sample Placement
- C. Test methods
- D. Platform/OS Differences
- E. ASIODriver class for the driver implementation
- F. Sony DSD Support
- G. Microsoft Windows 64 bit

# I. Overview

## 1. Introduction

This document presents an overview of the design goals and implementation tradeoffs for an efficient personal computer based audio processing system. Due to technological advances in the computer industry in the recent years, and the ever-growing use of the Steinberg Virtual Studio Technology (VST) as a standard for plug-in effects, it is clear that the personal computer will be a major target platform for a vast number of (previously impractical) audio-processing tasks.

## 2. The Goal

The personal computer platform really misses a relatively simple way of accessing multiple audio inputs and outputs. Today's operating system's audio support is designed for stereo input and stereo output only. There is no provision to extend this without creating major problems, i.e. synchronization issues between the different input and output channels.

With the Steinberg Audio Stream I/O (ASIO) Steinberg wants to help hardware and software manufacturers to create hardware and driver software which extends the personal computer's audio connectivity and meets the expectations of the customer (musician and audio engineer).

The Audio Stream I/O API addresses the areas of efficient audio processing, high data throughput, synchronization, low latency and extensibility on the audio hardware side. The interface is not bound to any fixed number of input and output channels (of course this numbers is limited by the processing power and data throughput of the computer system itself). It puts no limitation on the sample rate (32 kHz to 96 kHz or higher), sample format (16, 24, 32 bit or 32/64 bit floating point formats). It takes advantage of today's computer architectures for high data throughput (PCI, FireWire). It supports sophisticated hardware solutions for additional audio processing but it remains simple in comparison to other approaches.

## 3. The Design

The audio subsystem/hardware is treated as a software component (called audio driver). That is ASIO requires that the hardware manufacturers provide a driver, which abstracts the audio hardware in the way ASIO can deal with.

For efficient processing and great flexibility ASIO implements channels (input and output) as circular buffers with blocks of data. Actually a double buffer mechanism is used, which can be easily adapted to a great number of different buffer implementations. One buffer contains always data for one single channel only. This approach allows the host application to implement additional processing algorithms very effectively (opposed to an interleaved buffer model). The size of the data blocks is ascertained from the audio driver, to allow the best support for the audio hardware.

The audio driver allocates the memory for the actual data blocks. This allows the hardware manufacturer to select the best memory access method for their audio I/O solution. DMA or memory mapped I/O can be supported with equal efficiency. This is one of the keys for achieving

low latency in audio processing. Since current operating systems are in transition towards multi-tasking, with all the advantages and tradeoffs, ASIO supports all features (timestamped event notification and adaptive data pre-fetch/processing) necessary for asynchronous operation, usually occurring in preemptive multitasking operating systems such as MacOS X, Windows 95/NT, IRIX, BeOS. This will become also important for multi processor machines (SMP) or network distributed processing. (A direction in which personal computer systems will be heading to in the near future.)

## II. Implementation Guide

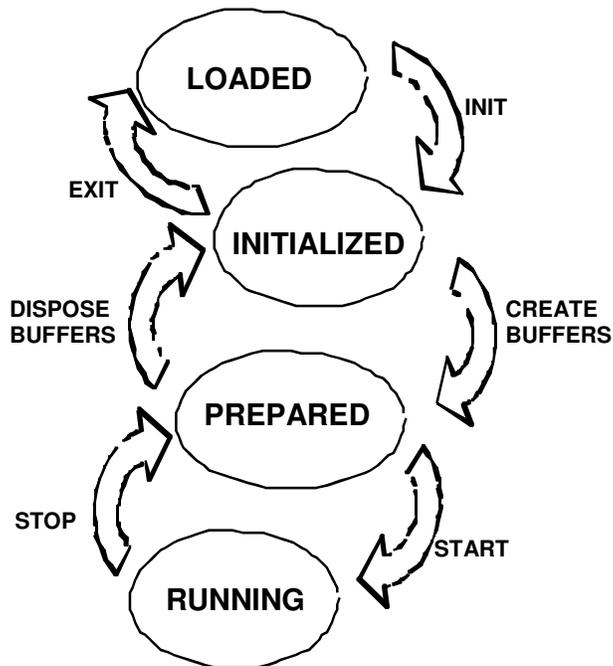
### 1. Instantiation

The host application loads the driver and "links" it to itself. Depending on the Operating System, different methods are necessary. Please refer to the Chapter 4 "Host Utility API Reference" and the provided code.

### 2. Operation

The Finite State Machine diagram illustrates the states of the operation.

**Loaded:** Driver code is accessible by the application



**Initialized:** Driver is allocated by the application and inquiries from the application can be accepted. It is not necessary that the hardware be already allocated.

**Prepared:** Audio buffers are allocated and the driver is prepared for running

**Running:** The hardware is running and the audio streaming takes place

The commands altering the states are:

*Init* Initialize the driver for use by the application. Optionally acquire hardware or load additional driver components. **ASIOInit()**

*CreateBuffers* Allocate memory for the audio buffers and allocate hardware resources for the audio channels. **ASIOCreateBuffers()**

*Start* Starts the audio streaming process. **ASIOStart()**

<i>Stop</i>	Stops the streaming process. <b>ASIOStop()</b>
<i>DisposeBuffers</i>	Deallocates hardware resources for the used channels and disposes the memory for the audio buffers. <b>ASIODisposeBuffers()</b>
<i>Exit</i>	Deallocates any remaining resources and puts the driver back into the uninitialized state. <b>ASIOExit()</b>

### 3. Driver Query by the Host Application

The ASIO API provides several inquiry functions for the host application and the driver.

After the driver is initialized, the application will query for all or some of the following capabilities:

<b>ASIOGetChannels()</b>	query for the number of available audio channels
<b>ASIOGetBufferSize()</b>	get the supported audio buffer sizes
<b>ASIOCanSampleRate()</b>	ask whether a specific sample rate is supported by the driver/hardware
<b>ASIOGetSampleRate()</b>	get the current sample rate
<b>ASIOGetClockSources()</b>	get the possible clock source options of the audio hardware
<b>ASIOGetChannelInfo()</b>	get information about a specific channel (sample type, name, word clock group)
<b>ASIOSetSampleRate()</b>	Set a sample rate of the internal clock, should be used as base sample rate if an external clock source is selected.
<b>ASIOSetClockSource()</b>	set the clock source of the card.
<b>ASIOGetLatencies()</b>	query for the constant audio latencies relative to the bufferSwitch() callback.

Note: As **ASIOGetLatencies()** will also have to include the audio buffer size of the **ASIOCreateBuffers()** call, the application has to call this function after the buffer creation. In the case that the call occurs beforehand the driver should assume preferred buffer size.

Additional queries introduced with ASIO 2.0. These queries will occur via the **ASIOFuture()** call.

<b>kAsioCanTimeInfo</b>	query for support of the new bufferSwitchTimeInfo() callback
<b>kAsioCanTimeCode</b>	query for time code support of the audio hardware
<b>kAsioCanInputMonitor</b>	query for direct input monitoring support

Additional queries introduced with ASIO 2.3. They also occur via the **ASIOFuture()** call.

<b>kAsioCanReportOverload</b>	query if the driver can detect overload conditions. If the driver returns ASE_SUCCESS it has the sole responsibility to detect any drop-out in the audio stream and report them to the host. This is done via the asioMessage() callback and the kAsioOverload message selector.
<b>kAsio GetInternalBufferSamples</b>	query for the internal buffering of the driver. The authors of ASIO once assumed that drivers work directly with the by ASIO demanded double buffer, but driver designers might choose to add additional internal buffering due to technical constraints

(e.g. USB audio interfaces). Knowing about such internal buffering, a host may optimize its own performance accordingly or derive e.g. more precise headroom information.

#### **4. Host Query by the Driver**

After the *Prepared* state the driver can query the application for the following information via the `asioMessage()` callback:

**kAsioSelectorSupported** query whether the `asioMessage` selector is supported by the application  
**kAsioEngineVersion** query the host application for the ASIO version it implements. The application will return the highest implemented version. The host application is expected to always support lower version implementations.

## 5. Audio Streaming

The audio streaming starts after the ASIOStart() call. Prior to starting the hardware streaming the driver will issue one or more bufferSwitch() or bufferSwitchTimeInfo() callbacks to fill its first output buffer(s). Since the hardware did not provide any input data yet the input channels' buffers should be filled with silence by the driver.

After the driver started the hardware streaming the bufferSwitch() or bufferSwitchTimeInfo() callbacks will always be called when the hardware requires more data. The driver passes the index of the double buffer half, which has to be processed, to the application. Upon return of the callback the application has read all input data and provided all output data.

Note: In order to reduce the amount of temporary buffers for a driver/hardware solution which has to further process the audio data, e.g. copy the data to the hardware, the ASIO API provides the ASIOOutputReady() method. ASIOOutputReady() is called immediately when the application has finished the processing of the buffers. This is a provision for drivers which require an immediate return of the bufferSwitch() or bufferSwitchTimeInfo() callback on interrupt driven systems like the Apple Macintosh, which cannot spend too long in the callback at a low level interrupt time. Instead the application will process the data at an interruptible execution level and inform the driver once it finished the processing. On a thread based system like Windows 95/98/NT or 2000 the callback can always process the data.

## 6. Media Synchronization (Sampleposition and System Time)

In order to provide proper media synchronization information to the host application a driver should fetch, at the occurrence of the bufferSwitch() or bufferSwitchTimeInfo() callback invocation event (interrupt or timed event), the current system time and sample position of the first sample of the audio buffer, which will be past to the callback. The host application retrieves this information during the bufferSwitch() callback with ASIOGetSamplePosition() or in the case of the bufferSwitchTimeInfo() callback this information is part of the parameters to the callback.

The following table shows the callback sequence and its values for the bufferSwitchTimeInfo() callback (1024 samples audio buffer size at 44100 Hz, System Time at Start is 2000 ms).

Callback No:	0	1	2	3	4	5	6
BufferIndex:	0	1	0	1	0	1	0
SystemTime(ms):	2000	2000	2023	2046	2069	2092	2116
SamplePosition:	0	1024	2048	3072	4096	5120	6144

Please note in the above table the same system time is used for the first two callbacks, as the hardware did not start yet. Since ASIO implements a constant streaming model, the time information of the first few bufferSwitch callbacks has to be ignored. The media synchronization process will start once the host application detects the constant streaming process. (It can be assumed that the streaming will be constant after the third bufferSwitch callback, more

sophisticated detection methods however are preferred as some driver/hardware combinations might retrieve more audio data for the hardware output processing.)

Note: The system time timestamp is a reference time of the operating system. Please see the Appendix D. "Platform / OS Differences" for the used system time information on the operating system.

## 7. Driver Notifications to the Host Application

The driver can notify the host application of the occurrence of some events which need special treatment by the host application. The following notification messages will be send via the `asioMessage()` callback.

<b>kAsioResetRequest</b>	the driver needs a reset, in case of an unexpected event or a reconfiguration
<b>kAsioBufferSizeChange</b>	the buffer sizes will change, issued by the user (however few applications support this yet, hence it is recommended to issue the <b>kAsioResetRequest</b> instead)
<b>kAsioResyncRequest</b>	the driver detected underruns and requires a resynchronization
<b>kAsioLatenciesChanged</b>	the driver detected a latency change

Note: A host application has to defer processing of these notifications to a later "secure" time as the driver has to finish its processing of the notification. Especially on the **kAsioResetRequest** it is a bad idea to unload the driver during the `asioMessage` callback since the callback has to return back into the driver, which would then be no longer present.

**sampleRateDidChange()** informs the host application that the driver detected a sample rate change. Usually only used for an external clock source which changes its sample rate.

**kAsioOverload** informs the host application that the driver detected an overload condition, i.e. a drop-out in the audio stream occurred. Though once an overload occurred, it can't be healed anymore, but at least the host can warn the user by some appropriate notification (e.g. flashing GUI element)

## III. Function Reference

### Initialization/Termination

```
ASIOError ASIOInit(ASIODriverInfo *info);
ASIOError ASIOExit(void);
```

### Start/Stop

```
ASIOError ASIOStart(void);
ASIOError ASIOStop(void);
```

### Inquiry methods and sample rate

```
ASIOError ASIOGetChannels(long *numInputChannels, long *numOutputChannels);
ASIOError ASIOGetLatencies(long *inputLatency, long *outputLatency);
ASIOError ASIOGetBufferSize(long *minSize, long *maxSize, long *preferredSize, long
    *granularity);
ASIOError ASIOCanSampleRate(ASIOSampleRate sampleRate);
ASIOError ASIOGetSampleRate(ASIOSampleRate *currentRate);
ASIOError ASIOSetSampleRate(ASIOSampleRate sampleRate);
ASIOError ASIOGetClockSources(ASIOClockSource *clocks, long *numSources);
ASIOError ASIOSetClockSource(long reference);
ASIOError ASIOGetSamplePosition (ASIOSamples *sPos, ASIOTimeStamp *tStamp);
ASIOError ASIOGetChannelInfo(ASIOChannelInfo *info);
```

### Buffer preparation

```
ASIOError ASIOCreateBuffers(ASIOBufferInfo *bufferInfos, long numChannels, long bufferSize,
    ASIOCallbacks *callbacks);
ASIOError ASIODisposeBuffers(void);
ASIOError ASIOOutputReady(void);
```

### Miscellaneous

```
ASIOError ASIOControlPanel(void);
ASIOError ASIOFuture(long selector, void *params);
```

### Callbacks

```
void (*bufferSwitch) (long doubleBufferIndex, ASIOBool directProcess);
ASIOTime* (*bufferSwitchTimeInfo) (ASIOTime* params, long doubleBufferIndex, ASIOBool
directProcess);
void (*sampleRateDidChange) (ASIOSampleRate sRate);
long (*asioMessage) (long selector, long value, void* message, double* opt);
```

# 1. Initialization/Termination

---

**ASIOError ASIOInit(ASIODriverInfo \*info);**

**Purpose:**

Initialize the AudioStreamIO.

**Parameter:**

*info* pointer to an **ASIODriverInfo** structure.

**Returns:**

If neither input nor output is present ASE\_NotPresent will be returned. ASE\_NoMemory, ASE\_HWMalfunction are other possible error conditions

struct **ASIODriverInfo**

```
{
    long      asioVersion;          'on input' - the host version. This is 0 for earlier ASIO
                                   implementations. The asioMessage callback is implemented
                                   only if asioVersion is 2 or greater. (Sorry but due to a design
                                   oversight the driver doesn't have access to the host version in
                                   ASIOInit, see also Appendix E.)
                                   Added selector for host (engine) version in the asioMessage
                                   callback so we're ok from now on.

                                   'on return', - ASIO implementation version. Older versions
                                   are 1. If you support this version (namely,
                                   ASIOOutputReady() ) this should be 2 or higher. also see the
                                   note in ASIOGetSamplePosition() !

    long      driverVersion;        on return, the driver version (format is driver specific)

    char      name[32];             on return, a null-terminated string containing the driver's
                                   name

    char      errorMessage[124];    on return, should contain a user message describing the type
                                   of error that occurred during ASIOInit(), if any.

    void      *sysRef;             on input: system reference (Windows: application main
                                   window handle)
};
```

-----  
**ASIOError ASIOExit(void);**

**Purpose:**

Terminates the AudioStreamIO.

**Parameter:**

None.

**Returns:**

If neither input nor output is present ASE\_NotPresent will be returned.

**Notes:**

This implies ASIOStop() and ASIODisposeBuffers(), meaning that no host callbacks must be accessed after ASIOExit().

## 2. Start/Stop

---

### ASIOError ASIOStart(void);

**Purpose:**

Start input and output processing synchronously.

This will

- reset the sample counter to zero
- start the hardware (both input and output)

The first call to the hosts' bufferSwitch(index == 0) then tells the host to read from input buffer A (index 0), and start processing to output buffer A while output buffer B (which has been filled by the host prior to calling ASIOStart()) is possibly sounding (see also ASIOGetLatencies())

**Parameter:**

None.

**Returns:**

If neither input nor output is present, ASE\_NotPresent will be returned. If the hardware fails to start, ASE\_HWMalfunction will be returned.

**Notes:**

There is no restriction on the time that ASIOStart() takes to perform (that is, it is not considered a real-time trigger).

---

### ASIOError ASIOStop(void);

**Purpose:**

Stops input and output processing altogether.

**Parameter:**

None.

**Returns:**

If neither input nor output is present ASE\_NotPresent will be returned.

**Notes:**

On return from ASIOStop(), the driver must not call the hosts bufferSwitch() routine. On a pre-emptive multitasking OS you have to make sure that no pending events will call bufferSwitch() after the driver returned from this function.

### 3. Inquiry methods and sample rate

---

**ASIOError ASIOGetChannels(long \*numInputChannels, long \*numOutputChannels);**

**Purpose:**

Returns number of individual input/output channels.

**Parameter:**

*numInputChannels* will hold the number of available input channels.

*numOutputChannels* will hold the number of available output channels.

**Returns:**

If no input/output is present ASE\_NotPresent will be returned. If only inputs, or only outputs are available, the according other parameter will be zero, and ASE\_OK is returned.

---

**ASIOError ASIOGetLatencies(long \*inputLatency, long \*outputLatency);**

**Purpose:**

Returns the input and output latencies. This includes device specific delays, like FIFOs etc.

**Parameter:**

*inputLatency* on return will hold the 'age' of the first sample frame in the input buffer when the hosts reads it in bufferSwitch() (this is theoretical, meaning it does not include the overhead and delay between the actual physical switch, and the time when bufferSwitch() enters). This will usually be the size of one block in sample frames, plus device specific latencies.

*outputLatency* on return will specify the time between the buffer switch, and the time when the next play buffer will start to sound. The next play buffer is defined as the one the host starts processing after (or at) bufferSwitch(), indicated by the index parameter (0 for buffer A, 1 for buffer B).

It will usually be either one block, if the host writes directly to a DMA buffer or two or more blocks if the buffer is 'latched' by the driver. As an example, on ASIOStart(), the host will have filled the play buffer at index 1 already; when it gets the callback (with the parameter index == 0), this tells it to read from the input buffer 0, and start to fill the play buffer 0 (assuming that now play buffer 1 is already sounding). In this case, the output latency is one block. If the driver decides to copy buffer 1 at that time, and pass it to the hardware at the next slot (which is most commonly done, but should be avoided), the output latency becomes two blocks instead, resulting in a total i/o latency of at least 3 blocks. As memory access is the main bottleneck in native DSP processing, and to achieve lower latency, it is highly recommended to try to avoid copying (this is also why the driver is the owner of the buffers). To summarize, the minimum i/o latency can be achieved if the

input buffer is processed by the host into the output buffer which will physically start to sound on the next time slice. Also note that the host expects the `bufferSwitch()` callback to be accessed for each time slice in order to retain sync, possibly recursively; if it fails to process a block in time, it will suspend its operation for some time in order to recover.

**Returns:**

If no input/output is present `ASE_NotPresent` will be returned.

-----  
**ASIOError ASIOGetBufferSize(long \*minSize, long \*maxSize, long \*preferredSize, long \*granularity);**

**Purpose:**

Returns min, max, and preferred buffer sizes for input/output

**Parameter:**

<i>minSize</i>	on return will hold the minimum buffer size
<i>maxSize</i>	on return will hold the maximum possible buffer size
<i>preferredSize</i>	on return will hold the preferred buffer size (a size which best fits performance and hardware requirements)
<i>granularity</i>	on return will hold the granularity at which buffer sizes may differ. Usually, the buffer size will be a power of 2; in this case, granularity will hold -1 on return, signaling possible buffer sizes starting from <code>minSize</code> , increased in powers of 2 up to <code>maxSize</code> .

**Returns:**

If no input/output is present `ASE_NotPresent` will be returned.

**Notes:**

When minimum and maximum buffer size are equal, the preferred buffer size has to be the same value as well; granularity should be 0 in this case.

-----  
**ASIOError ASIOCanSampleRate(ASIOSampleRate sampleRate);**

**Purpose:**

Inquires of the hardware if a specific available sample rate is available.

**Parameter:**

*sampleRate* is the rate in question.

**Returns:**

If the stated sample rate is not supported, ASE\_NoClock will be returned. If no input/output is present ASE\_NotPresent will be returned.

-----  
**ASIOError ASIOGetSampleRate(ASIOSampleRate \*currentRate);**

**Purpose:**

Get the current sample Rate.

**Parameter:**

*currentRate* on return will hold the current sample rate on return.

**Returns:**

If sample rate is unknown, sampleRate will be 0 and ASE\_NoClock will be returned.  
If no input/output is present ASE\_NotPresent will be returned.

-----  
**ASIOError ASIOSetSampleRate(ASIOSampleRate sampleRate);**

**Purpose:**

Set the hardware to the requested sample Rate. If sampleRate == 0, enable external sync.

**Parameter:**

*sampleRate* the requested rate

**Returns:**

If sampleRate is unknown ASE\_NoClock will be returned. If the current clock is external, and sampleRate is != 0, ASE\_InvalidMode will be returned. If no input/output is present ASE\_NotPresent will be returned.

-----  
**ASIOError ASIOGetClockSources(ASIOClockSource \*clocks, long \*numSources);**

**Purpose:**

Get the available external audio clock sources

**Parameter:**

*clocks* points to an array of **ASIOClockSource** structures.

*numSources* on input: the number of allocated array members  
on output: the number of available clock sources, at least 1 (internal clock generator).

**Returns:**

If no input/output is present ASE\_NotPresent will be returned.

```
struct ASIOClockSource {  
    long      index;          this is used to identify the clock source when  
                                ASIOSetClockSource() is accessed, should be an index  
                                counting from zero  
  
    long      associatedChannel; the first channel of an associated input group, if any.  
    long      associatedGroup;  the group index of that channel. Groups of channels are  
                                defined to separate, for instance analog, S/PDIF, AES/EBU,  
                                ADAT connectors etc, when present simultaneously. Note that  
                                associated channel is enumerated according to  
                                numInputs/numOutputs, meaning it is independent from a  
                                group (see also ASIOGetChannelInfo()) inputs are associated  
                                to a clock if the physical connection transfers both data and  
                                clock (like S/PDIF, AES/EBU, or ADAT inputs). If there is no  
                                input channel associated with the clock source (like Word  
                                Clock, or internal oscillator), both associatedChannel and  
                                associatedGroup should be set to -1.  
  
    ASIOBool isCurrentSource; on exit, ASIOTrue if this is the current clock source,  
                                ASIOFalse else  
  
    char      name[32];      a null-terminated string for user selection of the available  
                                sources.  
};
```

---

**ASIOError ASIOSetClockSources(long index);****Purpose:**

Set the audio clock source

**Parameter:**

*index* as obtained from an inquiry to ASIOGetClockSources()

**Returns:**

If no input/output is present ASE\_NotPresent will be returned. If the clock can not be selected because an input channel which carries the current clock source is active, ASE\_InvalidMode \*may\* be returned (this depends on the properties of the driver and/or hardware).

**Notes:**

Should \*not\* return ASE\_NoClock if there is no clock signal present at the selected source; this will be inquired via ASIOGetSampleRate(). It should call the host callback procedure sampleRateHasChanged(), if the switch causes a sample rate change, or if no external clock is present at the selected source.

---

**ASIOError ASIOGetSamplePosition (ASIOSamples \*sPos, ASIOTimeStamp \*tStamp);****Purpose:**

Inquires the sample position/time stamp pair.

**Parameter:**

*sPos* on return will hold the sample position on return. The sample position is reset to zero when ASIOStart() gets called.

*tStamp* on return will hold the system time when the sample position was latched.

**Returns:**

If no input/output is present, ASE\_NotPresent will be returned. If there is no clock, ASE\_SPNotAdvancing will be returned.

**Notes:**

In order to be able to synchronize properly, the sample position / time stamp pair must refer to the \*current block\*, that is, the engine will call ASIOGetSamplePosition() in its bufferSwitch() callback and expect the time for the first sample of the current block. Thus, when requested in the very first bufferSwitch after ASIOStart(), the sample position should be zero, and the time stamp should refer to the very time where the stream was started. It also means that the sample position must be block aligned. The driver must ensure proper interpolation if the system time can not be determined for the block position. The driver is responsible for precise time stamps as it usually has most direct access to lower level resources. Proper behavior of ASIOGetSamplePosition() and ASIOGetLatencies() are essential for precise media synchronization!

-----  
**ASIOError ASIOGetChannelInfo (ASIOChannelInfo \*info);**

**Purpose:**

Retrieve information about the nature of a channel.

**Parameter:**

*info* is a pointer to one **ASIOChannelInfo** structure.

**Returns:**

If no input/output is present ASE\_NotPresent will be returned.

**Notes:**

If possible, the string should be organized such that the first characters are most significantly describing the nature of the port, to allow for identification even if the view showing the port name is too small to display more than 8 characters, for instance.

struct **ASIOChannelInfo**

```
{
    long          channel;          on input: the channel index of the channel in question.
    ASIOBool     isInput;          on input: ASIOTrue if info for an input channel is
                                   requested, else an output channel is addressed
    ASIOBool     isActive;        on output: ASIOTrue if channel is active as it was
                                   installed by ASIOCreateBuffers(), otherwise it is set to
                                   ASIOFalse
    long         channelGroup;    on output: the channel group that the channel belongs to.
                                   For drivers which support different types of channels, like
                                   analog, S/PDIF, AES/EBU, ADAT etc interfaces, there
                                   should be a reasonable grouping of these types. Groups
                                   are always independent from a channel index, that is, a
                                   channel index always counts from 0 to
                                   numInputs/numOutputs regardless of the group it may
                                   belong to. There will always be at least one group (group
                                   0). Please also note that by default, the host may decide to
                                   activate channels 0 and 1; thus, these should belong to the
                                   most useful type (analog i/o, if present).
    ASIOSampleType type;          on output: contains the sample type of the channel
    char         name[32];        on output: describing the type of channel in question.
                                   Used to allow for user selection, and enabling of specific
                                   channels. Examples: "Analog In", "SPDIF Out" etc.
};
```

## 4. Buffer preparation

-----  
**ASIOError ASIOCreateBuffers(ASIOBufferInfo \*bufferInfos, long numChannels, long bufferSize, ASIOCallbacks \*callbacks);**

### Purpose:

Allocates input/output buffers for all input and output channels to be activated.

### Parameter:

*bufferInfos* is a pointer to an array of **ASIOBufferInfo** structures.

*numChannels* is the sum of all input and output channels to be created; thus *bufferInfos* is a pointer to an array of *numChannels* **ASIOBufferInfo** structures.

*bufferSize* selects one of the possible buffer sizes as obtained from **ASIOGetBufferSizes()**.

*callbacks* is a pointer to an **ASIOCallbacks** structure.

### Returns:

If not enough memory is available **ASE\_NoMemory** will be returned. If no input/output is present **ASE\_NotPresent** will be returned. If *bufferSize* is not supported, or one or more of the *bufferInfos* elements contain invalid settings, **ASE\_InvalidMode** will be returned.

### Notes:

If individual channel selection is not possible but requested, the driver has to handle this. Namely, **bufferSwitch()** will only fill buffers of enabled outputs. If possible, processing and buss activities overhead should be avoided for channels that were not enabled here.

struct **ASIOBufferInfo**

```
{
    ASIOBool    isInput;        on input: set to ASIOTrue if it describes an input buffer.
                                Otherwise it is an output buffer

    long        channelNum;      on input: the index of the channel in question (counting from
                                0)

    void        *buffers[2];     on output: 2 pointers to the two halves of the channels'
                                double-buffer. The sizes of the buffer(s) of course depend on
                                both the bufferSize and the ASIOSampleType of the device
                                for the channel. See also ASIOGetChannelInfo().
};
```

---

**ASIOError ASIODisposeBuffers(void);****Purpose:**

Releases all buffers for the device.

**Parameter:**

None.

**Returns:**

If no buffer were ever prepared, ASE\_InvalidMode will be returned. If no input/output is present ASE\_NotPresent will be returned.

---

**ASIOError ASIOOutputReady(void);****Purpose:**

This tells the driver that the host has completed processing the output buffers. If sample data format required by the hardware differs from the supported ASIO sample formats, but the hardware buffers are DMA buffers, the driver will have to convert the audio stream data. As the bufferSwitch callback is usually issued at DMA block switch time, the driver will have to convert the \*previous\* host buffer, which increases the output latency by one block.

When the host finds out that ASIOOutputReady() returns true, it will issue this call whenever it completed output processing. Then the driver can convert the host data directly to the DMA buffer to be played next, reducing output latency by one block.

Another way to look at it is, that the buffer switch is called in order to pass the \*input\* stream to the host, so that it can process the input into the output, and the output stream is passed to the driver when the host has completed its process.

**Parameter:**

None

**Returns:**

Only if the above mentioned scenario is given, and a reduction of output latency can be achieved by this mechanism, should ASE\_OK be returned. Otherwise (and usually) ASE\_NotPresent should be returned in order to prevent further calls to this function. Note that the host may want to determine if it can use this when the system is not yet fully initialized, so ASE\_OK should always be returned if the mechanism makes sense.

**Notes:** Please remember to adjust ASIOGetLatencies() according to whether ASIOOutputReady() was ever called or not, if your driver supports this scenario. Also note that the engine may fail to call ASIOOutputReady() in time in overload cases. As already mentioned, bufferSwitch should be called for every block regardless of whether a block could be processed in time.

## 5. Miscellaneous

---

### ASIOError ASIOControlPanel(void);

**Purpose:**

Request the driver to start a control panel component for device specific user settings. This might not be accessible on all platforms.

**Parameter:**

None.

**Returns:**

If no panel is available ASE\_NotPresent will be returned. Actually, the return code is ignored.

**Notes:**

If the user applied settings which require a re-configuration of parts or all of the engine and/or driver (such as a change of the block size), the asioMessage callback can be used (see ASIOCallbacks).

---

### ASIOError ASIOFuture(long selector, void \*params);

**Purpose:**

various

**Parameter:**

*selector* operation Code as to be defined. Zero is reserved for testing purposes.

*params* depends on the selector; usually pointer to a structure for passing and retrieving any type and amount of parameters.

**Returns:**

The return value is also selector dependent. If the selector is unknown, ASE\_InvalidParameter should be returned to prevent further calls with this selector. On success, ASE\_SUCCESS must be returned

**Notes:**

See selectors defined below.

ASIOFuture() *selectors:*

**kAsioEnableTimeCodeRead**

**kAsioDisableTimeCodeRead**

**Purpose:**

Enable or Disable the time code reader if the hardware device supports time code.

**Parameter:**

None.

**Returns:**

*ASE\_SUCCESS* if request is accepted or *ASE\_NotPresent* otherwise

**Note:**

If the hardware/driver does not support time code reader facilities *ASE\_NotPresent* should be returned.

**kAsioSetInputMonitor**

**Purpose:**

Set the direct input monitoring state.

**Parameter:**

*params* pointer to **ASIOInputMonitor** structure.

**Returns:**

*ASE\_SUCCESS* if request is accepted or otherwise *ASE\_NotPresent*

**Note:**

If the hardware does not support patching and mixing a straight 1 to 1 routing is suggested. The driver should ignore all the information of **ASIOInputMonitor** it cannot deal with, usually these might be either or all of *output*, *gain*, *pan*.

*Output* is the base channel of a stereo channel pair, i.e. *output* is always an even channel (0,2,4...). If an odd input channel should be monitored and no panning or output routing can be applied, the driver has to use the next higher output (imply a hard right pan).

```
struct ASIOInputMonitor
{
    long    input;           this input was set to monitor (or off), -1: all
    long    output;         suggested output for monitoring the input (if so)
    long    gain;           suggested gain, ranging 0 - 0x7fffffffL (-inf to
+12
                                dB, 0x20000000 equals 0 dB)
    ASIOBool state;        ASIOTrue => on, ASIOFalse => off
    long    pan;           suggested pan, 0 => left, 0x7fffffff => right
};
```

**kAsioSetIoFormat****Purpose:**

Set the I/O format of the device.

**Parameter:**

*params* pointer to **ASIOIoFormat** structure.

**Returns:**

*ASE\_SUCCESS* if request is accepted or otherwise *ASE\_NotPresent*

**Note:**

See Appendix F for further details.

**kAsioGetIoFormat****Purpose:**

Get the I/O format of the device.

**Parameter:**

*params* pointer to **ASIOIoFormat** structure.

**Returns:**

*ASE\_SUCCESS* if request is accepted or otherwise *ASE\_NotPresent*

**Note:**

See Appendix F for further details.

**kAsioCanDoIoFormat****Purpose:**

Query the device for support of the given I/O format.

**Parameter:**

*params* pointer to **ASIOIoFormat** structure.

**Returns:**

*ASE\_SUCCESS* if request is accepted or otherwise *ASE\_NotPresent*

**Note:**

See Appendix F for further details.

**kAsioCanReportOverload****Purpose:**

Query the driver if the device can detect overload conditions and report them back to the host. It is not a constraint to support this, but we encourage driver implementors to do so, as we think that the driver is the best place to do this. It is essential for users to know if overloads and thus drop-outs in the audio stream occurred. If the driver doesn't provide this information, the host has to implement some overload detection itself, which will most likely be inferior and less reliable.

**Parameter:**

*params*            ignore

**Returns:**

*ASE\_SUCCESS* if the driver detects overload conditions and will report them to the host.

**Note:**

If the driver returns *ASE\_SUCCESS* the host will switch off its own overload detection and solely relies on the driver's reports.

Overloads are reported to the host via the *AsioMessage()* callback and the *kAsioOverload* message selector.

**kAsioGetInternalBufferSamples****Purpose:**

Query the driver for the internal buffering of the device. Additionally to the by ASIO demanded double buffer, driver designs might provide for even further internal buffering due to technical constraints. Knowing this additional buffering may help the host in e.g. projecting the system's performance headroom. Though it is not mandatory, but we encourage every driver manufacturer to support *kAsioGetInternalBufferSamples*.

**Parameter:**

*params*            pointer to **ASIOInternalBufferInfo** structure..

**Returns:**

*ASE\_SUCCESS* if *ASIOInternalBufferInfo* structure got filled up with pertinent information. Returns *ASE\_InvalidParameter* if not supported.

**Note:**

```
struct ASIOInternalBufferInfo
{
    long inputSamples; // size of driver's internal input buffering which
                      // is included in ASIOGetLatencies
    long outputSamples; // size of driver's internal output buffering
                       // which is included in ASIOGetLatencies
};
```

## 6. Callbacks

-----  
**void (\*bufferSwitch) (long doubleBufferIndex, ASIOBool directProcess);**

**Purpose:**

bufferSwitch indicates that both, input and output are to be processed.

**Parameter:**

*doubleBufferIndex* The current buffer half index (0 or 1) determines the output buffer that the host should start to fill. The other buffer will be passed to output hardware regardless of whether it got filled in time or not. The addressed input buffer is now filled with incoming data. Note that because of the synchronicity of i/o, the input always has at least one buffer latency in relation to the output.

*directProcess* suggests to the host whether it should immediately start processing (*directProcess* == ASIOTrue), or whether its process should be deferred because the call comes from a very low level (for instance, a high level priority interrupt), and direct processing would cause timing instabilities for the rest of the system. If in doubt, *directProcess* should be set to ASIOFalse.

**Returns:**

No return value.

**Notes:**

bufferSwitch() may be called at interrupt time for highest efficiency.

-----  
**ASIOTime\* (\*bufferSwitchTimeInfo) (ASIOTime\* params, long doubleBufferIndex, ASIOBool directProcess);**

**Purpose:**

bufferSwitchTimeInfo indicates that both input and output are to be processed. It also passes extended time information (time code for synchronization purposes) to the host application and back to the driver.

**Parameter:**

*params* pointer to ASIOTime structure

*doubleBufferIndex* The current buffer half index (0 or 1) determines the output buffer that the host should start to fill. The other buffer will be passed to output hardware regardless of whether it got filled in time or not. The addressed input buffer is now filled with incoming data. Note that because of the synchronicity of i/o, the input always has at least one buffer latency in relation to the output.

*directProcess* suggests to the host whether it should immediately start processing (*directProcess* == ASIOTrue), or whether its process should be deferred because the call comes from a very low level (for instance, a high level priority interrupt), and direct processing would cause timing instabilities for the rest of the system. If in doubt, *directProcess* should be set to ASIOFalse.

**Returns:**

Pointer to ASIOTime structure with "output" time code information.

**Notes:**

bufferSwitchTimeInfo() may be called at interrupt time for highest efficiency.

This new callback with time info makes ASIOGetSamplePosition() and various calls to ASIOGetSampleRate obsolete, and allows for timecode sync etc. Therefore it is the preferred callback; it will be used if the driver successfully calls asioMessage with selector kAsioSupportsTimeInfo.

```
struct ASIOTime {
    long          reserved[4];      must be 0

    AsioTimeInfo timeInfo;         required

    ASIOTimeCode timeCode;         optional, evaluated if (timeCode.flags & kTcValid)
};
```

```
struct AsioTimeInfo {
    double        speed;            absolute speed (1. = nominal)

    ASIOTimeStamp systemTime;      system time related to samplePosition, in nanoseconds
                                         On Windows, must be derived from timeGetTime()

    ASIOSamples  samplePosition;   sample position since ASIOStart()

    ASIOSampleRate sampleRate;     current rate unsigned

    long         flags;             see AsioTimeInfoFlags

    char         reserved[12];     must be 0
};
```

**AsioTimeInfoFlags**

kSystemTimeValid	set if system time field is valid - must always be valid
kSamplePositionValid	set if sample position valid - must always be valid
kSampleRateValid	set if sample rate field is valid
kSpeedValid	set if speed field is valid
kSampleRateChanged	set if sample rate changed in between callbacks
kClockSourceChanged	set if clock source changed in between callbacks

**struct ASIOTimeCode {**

double	speed;	speed relation (fraction of nominal speed) optional; set to 0. or 1. if not supported
ASIOSamples	timeCodeSamples;	time in samples unsigned
long	flags;	see ASIOTimeCodeFlags
char	future[64];	set to 0

**};****ASIOTimeCodeFlags**

kTcValid	set if time code data is valid
kTcRunning	set if time code is running
kTcReverse	set if reverse time code is detected
kTcOnspeed	set if time code is on speed (optional)
kTcStill	set if still frame time code is received
kTcSpeedValid	set if speed field is valid

-----  
**void (\*sampleRateDidChange) (ASIOSampleRate sRate);**

**Purpose:**

This callback will inform the host application that a sample rate change was detected (e.g. sample rate status bit in an incoming S/PDIF or AES/EBU signal changes).

**Parameter:**

*sRate*                      The detected sample rate. 0 when sample rate is not known (for instance, clock loss when externally synchronized).

**Returns:**

No return value.

**Notes:**

The host application usually will just store the information. Actual action of the host application is not specified.

-----  
**long (\*asioMessage) (long selector, long value, void\* message, double\* opt);**

**Purpose:**

Generic callback use for various purposes, see selectors below.

**Parameter:**

*selector*                      what kind of message is send.

*value*                              single value. Specific to each *selector* as defined below.

*message*                          message parameter. Specific to each *selector* as defined below.

*opt*                                  optional parameter. Specific to each *selector* as defined below.

**Returns:**

Specific to the *selector*, undefined selectors will return 0.

**Notes:**

This callback was not implemented in the first ASIO host implementation of Cubase VST 3.0 for Macintosh, all other ASIO host application will have this callback.

asioMessage **selectors**

**kAsioSelectorSupported****Purpose:**

Test whether the host application supports a specific selector.

**Parameter:**

*value* selector to be test for.

**Host Application Returns:**

1L if selector is supported or 0 otherwise

**kAsioEngineVersion****Purpose:**

Ask the host application for its ASIO implementation.

**Parameter:**

None.

**Host Application Returns:**

Host ASIO implementation version, 2 or higher

**kAsioResetRequest****Purpose:**

Requests a driver reset. If accepted, this will close the driver (ASIOExit() ) and re-open it again (ASIOInit() etc.) at the next "safe" time (usually during the application IDLE time). Some drivers need to reconfigure for instance when the sample rate changes, or some basic changes have been made in ASIOControlPanel().

**Parameter:**

None.

**Host Application Returns:**

Return value is always 1L.

**Note:**

The request is merely passed to the application, there is no way to determine if it gets accepted at this time (but it usually will be).

**kAsioBufferSizeChange****Purpose:**

Informs the application that the driver has a new preferred buffer size.

**Parameter:**

*value*                    new buffer size.

**Host Application Returns:**

1L if request is accepted or 0 otherwise

**Note:**

If the request is not accepted but the buffer size changed, the driver should send kAsioResetRequest.

**kAsioResyncRequest****Purpose:**

The driver went out of sync, such that the timestamp is no longer valid. This is a request to re-start the engine and slave devices (sequencer).

**Parameter:**

None.

**Host Application Returns:**

1L if request is accepted or 0 otherwise

**kAsioLatenciesChanged****Purpose:**

Informs the application that the driver's latencies have changed. The engine will re-fetch the latencies.

**Parameter:**

None.

**Host Application Returns:**

Returns always 1L if the selector is supported is accepted or 0 otherwise

**kAsioSupportsTimeInfo****Purpose:**

Ask the application whether it supports the `bufferSwitchTimeInfo()` callback.

**Parameter:**

None.

**Host Application Returns:**

1L if `bufferSwitchTimeInfo()` is supported or 0 otherwise

**kAsioSupportsTimeCode****Purpose:**

Ask the application whether it supports time code reading in the `bufferSwitchTimeInfo()` callback.

**Parameter:**

None.

**Host Application Returns:**

1L if time code reading is supported or 0 otherwise

**Note:**

This requires that the `bufferSwitchTimeInfo()` callback is supported by the host application.

**kAsioOverload****Purpose:**

Notify the host that the driver detected an overload condition, i.e. an interrupt or drop-out in the audio stream.

**Parameter:**

None.

**Host Application Returns:**

Nothing, respectively the return value can be ignored

**Note:**

It is not mandatory for a driver to send overload messages unless previously the driver returned `ASE_SUCCESS` to `kAsioCanReportOverload` during an `ASIOFuture()` call.

## 7. Type definitions

---

### Sample Types

#### 1. Big Endian formats

The basic types, sample data is left aligned in the data word. The most significant Byte of the data word is stored first in memory.

<i>ASIOSTInt16MSB</i>	16 bit data word (fewer than 16 bits are not supported)
<i>ASIOSTInt24MSB</i>	This is the packed 24 bit format. 2 data words will spawn consecutive 6 bytes in memory. (Used for 18 and 20 bits as well, if they use this packed format)
<i>ASIOSTInt32MSB</i>	This format should also be used for 24 bit data, if the sample data is left aligned. Lowest 8 bit should be reset or dithered whatever the hardware/software provides.
<i>ASIOSTFloat32MSB</i>	IEEE 754 32 bit float, as found on PowerPC implementation
<i>ASIOSTFloat64MSB</i>	IEEE 754 64 bit double float, as found on PowerPC implementation

Right aligned variations of the *ASIOSTInt32MSB* data type.

These are used for 32 bit data transfers with different alignment of the sample data inside the 32 bit data word. This supports right aligned sample data in the 32-bit data word, most significant bits should be sign extended. (32 bit PCI bus systems can be more easily used with these)

<i>ASIOSTInt32MSB16</i>	sample data fills the least significant 16 bits, the other bits are sign extended
<i>ASIOSTInt32MSB18</i>	sample data fills the least significant 18 bits, the other bits are sign extended
<i>ASIOSTInt32MSB20</i>	sample data fills the least significant 20 bits, the other bits are sign extended
<i>ASIOSTInt32MSB24</i>	sample data fills the least significant 24 bits, the other bits are sign extended

## 2. Little Endian formats

The basic types, sample data is left aligned in the container. The least significant Byte of the data word is stored first in memory.

*ASIOSTInt16LSB*      16 bit data word

*ASIOSTInt24LSB*      This is the packed 24 bit format. 2 data words will spawn consecutive 6 bytes in memory. (Used for 18 and 20 bits as well, if they use this packed format)

*ASIOSTInt32LSB*      This format should also be used for 24 bit data, if the sample data is left aligned. Lowest 8 bit should be reset or dithered whatever the hardware/software provides.

*ASIOSTFloat32LSB*    IEEE 754 32 bit float, as found on Intel x86 architecture

*ASIOSTFloat64LSB*    IEEE 754 64 bit double float, as found on Intel x86 architecture

Right aligned variations of the *ASIOSTInt32LSB* data type.

These are used for 32 bit data transfers with different alignment of the sample data inside the 32 bit data word. This supports right aligned sample data in the 32-bit data word, most significant bits should be sign extended. (32 bit PCI bus systems can be more easily used with these)

*ASIOSTInt32LSB16*    32 bit data with 16 bit sample data right aligned

*ASIOSTInt32LSB18*    32 bit data with 18 bit sample data right aligned

*ASIOSTInt32LSB20*    32 bit data with 20 bit sample data right aligned

*ASIOSTInt32LSB24*    32 bit data with 24 bit sample data right aligned

### 3. Sony DSD formats

See Appendix F for further explanations.

*ASIOSTDSDInt8LSB1* DSD 1 bit data, 8 samples per byte. First sample in Least significant bit.

*ASIOSTDSDInt8MSB1* DSD 1 bit data, 8 samples per byte. First sample in Most significant bit.

*ASIOSTDSDInt8NER8* DSD 8 bit data, 1 sample per byte. No Endianness required.

-----  
**ASIOSamples ASIOTimeStamp**

Number of samples - data type is 64-bit integer

Timestamp data type is 64-bit integer. The Time format is Nanoseconds.

**Note:**

The 64 bit data is passed via a structure with two 32-bit values.

```
struct {
    unsigned long hi; // most significant bits (Bits 32..63)
    unsigned long lo; // least significant bits (Bits 0..31)
};
```

Unfortunately the ASIO API was implemented it before compiler supported consistently 64 bit integer types. By using the structure the data layout on a little-endian system like the Intel x86 architecture will result in a "non native" storage of the 64 bit data. The most significant 32 bit are stored first in memory, the least significant bits are stored in the higher memory space. However each 32 bit is stored in the native little-endian fashion.

-----  
**ASIOSampleRate**

Sample rates are expressed in IEEE 754 64 bit double float, native format as host computer

-----  
**ASIOBool** ASIOFalse     indicates a false value  
 ASIOTrue                 indicates a true value

**Error codes**

<i>ASE_OK</i>	This value will be returned whenever the call succeeded
<i>ASE_SUCCESS</i>	unique success return value for ASIOFuture calls
<i>ASE_NotPresent</i>	hardware input or output is not present or available
<i>ASE_HWMalfunction</i>	hardware is malfunctioning (can be returned by any ASIO function)
<i>ASE_InvalidParameter</i>	input parameter invalid
<i>ASE_InvalidMode</i>	hardware is in a bad mode or used in a bad mode
<i>ASE_SPNotAdvancing</i>	hardware is not running when sample position is inquired
<i>ASE_NoClock</i>	sample clock or rate cannot be determined or is not present
<i>ASE_NoMemory</i>	not enough memory for completing the request

## IV. Host Utility API Reference

### 1. AsioDrivers Class

The AsioDrivers class implements the unified driver enumeration and instantiation on the different OS platforms. However the implementation is currently limited to one active driver. Also it implies that a driver names will have max. 32 characters including the terminating '\0'.

```
AsioDrivers();  
~AsioDrivers();  
Constructor/Destructor for the driver manager class
```

---

```
bool loadDriver(char *name);
```

**Purpose:**

Instantiates an ASIO driver. Loads the driver executable into memory and resolves any OS specific linkage between the host application executable and the driver executable. The driver will be accessed via the global pointer *theAsioDriver* from inside "asio.cpp".

**Parameter:**

*name* name of the driver to load, *name* is to be obtained from getDriverNames().

**Returns:**

Returns true if the driver was instantiated successfully, false otherwise.

---

```
long getDriverNames(char **names, long maxDrivers);
```

**Purpose:**

Retrieves the names of the available drivers for the host application.

**Parameter:**

*names* an array of strings with up to 32 character each.

*maxDrivers* number of entries in the *names* array.

**Returns:**

Returns the number of filled entries of the *names* array. The value 0 described the fact that no driver is available.

-----  
**bool getCurrentDriverName(char \*name);**

**Purpose:**

Retrieves the name of the currently instantiated driver.

**Parameter:**

*name*                    an array with space for 32 characters.

**Returns:**

Returns *true* if a driver is currently instantiated, *false* if no driver is instantiated.

-----  
**long getCurrentDriverIndex();**

**Purpose:**

Retrieves the index into the *names* array of the currently instantiated driver. See *getDriverNames()*.

**Parameter:**

*None.*

**Returns:**

Returns index into the *names* array. If no driver is instantiated -1 will be returned.

-----  
**void removeCurrentDriver();**

**Purpose:**

Removes the currently instantiated driver from memory any further access to the driver will be denied.

**Parameter:**

*None.*

**Returns:**

*None.*

**Note:**

Successful removal of the driver is implied, even if no driver was instantiated.

## V. Appendix

### A. Using the `bufferSwitchTimeInfo()` callback

It is recommended to use the new method with time info even if the ASIO device does not support timecode; continuous calls to `ASIOGetSamplePosition()` and `ASIOGetSampleRate()` are avoided, and there is a defined relationship between callback time and the time info.

See the example code for a driver on the next page.

To initiate time info mode, after you have received the callbacks pointer in `ASIOCreateBuffers()`, you will call the `asioMessage()` callback with `kAsioSupportsTimeInfo` as the argument. If this returns 1, the host has accepted time info mode. Now the host expects the new callback `bufferSwitchTimeInfo` to be used instead of the old `bufferSwitch` method. The `ASIOTime` structure is assumed to be valid and accessible until the callback returns.

#### Using time code:

If the device supports reading time code, it will call the host's `asioMessage()` callback with `kAsioSupportsTimeCode` as the selector. It may then fill the according fields and set the `kTcValid` flag.

The host will call the future method with the `kAsioEnableTimeCodeRead` selector when it wants to enable or disable time-code reading by the device.

#### Note:

The `AsioTimeInfo/ASIOTimeCode` pair is supposed to work in both directions. As a matter of convention, the relationship between the sample position counter and the time code at buffer switch time is (ignoring offset between time-code and sample position when time-code is running):

on input:      sample 0 -> input buffer sample 0 -> time code 0  
on output:     sample 0 -> output buffer sample 0 -> time code 0

This means that for 'real' calculations, one has to take into account the according latencies.

**Example:**

```
ASIOTime asioTime;

in createBuffers() {
    memset(&asioTime, 0, sizeof(ASIOTime));
    AsioTimeInfo* ti = &asioTime.timeInfo;
    ti->sampleRate = theSampleRate;
    ASIOTimeCode* tc = &asioTime.timeCode;
    tc->speed = 1.; tc->frameRate = AsioFPS_30;    // frame rate
    timeInfoMode = false;
    canTimeCode = false;
    if(callbacks->asioMessage(kAsioSupportsTimeInfo, 0, 0, 0) == 1) {
        timeInfoMode = true;
    }
    #if kCanTimeCode
        if(callbacks->asioMessage(kAsioSupportsTimeCode, 0, 0, 0) == 1) {
            canTimeCode = true;
        }
    #endif
}

void switchBuffers(long doubleBufferIndex, bool processNow) {
    if(timeInfoMode) {
        AsioTimeInfo* ti = &asioTime.timeInfo;
        ti->flags = kSystemTimeValid | kSamplePositionValid | kSampleRateValid;
        ti->systemTime = theNanoSeconds;
        ti->samplePosition = theSamplePosition;
        if(ti->sampleRate != theSampleRate)
            ti->flags |= kSampleRateChanged;
        ti->sampleRate = theSampleRate;

    #if kCanTimeCode
        if(canTimeCode && timeCodeEnabled) {
            ASIOTimeCode* tc = &asioTime.timeCode;
            tc->timeCodeSamples = tcSamples;
            tc->flags = kTcValid | kTcRunning;    // if so...
        }
        ASIOTime* bb = callbacks->bufferSwitchTimeInfo(&asioTime,
            doubleBufferIndex, processNow ? ASIOTrue : ASIOFalse);
    #else
        callbacks->bufferSwitchTimeInfo(&asioTime, doubleBufferIndex, processNow ?
            ASIOTrue : ASIOFalse);
    #endif
    } else
        callbacks->bufferSwitch(doubleBufferIndex, ASIOFalse);
}
```

```
ASIOError ASIOFuture(long selector, void *params) {
    switch(selector) {
        case kAsioEnableTimeCodeRead:
            timeCodeEnabled = true;
            return ASE_SUCCESS;
        case kAsioDisableTimeCodeRead:
            timeCodeEnabled = false;
            return ASE_SUCCESS;
    }
    return ASE_NotPresent;
};
```



**B. ASIOGetLatencies() vs. Sample Placement**

This appendix covers the actual meaning of the latency values for ASIOGetLatencies(). To ease the understanding of these latencies the first sample inside the audio data buffer of the bufferSwitch() callback is used as reference into the continuous stream of audio samples.

The latency is required by the application to align the incoming audio data to the outgoing audio data to the outside world. The application considers the audio data as continuous stream of samples. It has to send the output data ahead of time to the driver in order to have the right samples at the output at the right time. The incoming audio data needs to be placed at the appropriate position during recording.

Below is a sample situation when a sample at sample position 256 is examined. The audio driver/hardware reports an output latency of 230 samples and an input latency of 239 samples.

```

Output      -----*
Buffers | 128 | 256 | 384 | 512 | ....
Input      *-----| will occur at position 495 in the stream
    
```

The output sample will appear at the output with a delay of 230 samples. Therefore it needs to be placed by the application into the audio buffer at position 26.

Input signal at sample position 256 will appear at sample position 495 inside the audio buffers. Since the driver informed the application about this delay it can place the input signal in relation to its internal time line (time reference).

**Output latency:**

The output latency specifies the time in number sample frames a sample will reach the output after it was passed from the application to the driver in an audio data buffer during a buffer switch. In order to provide the correct latency value for ASIOGetLatencies() you have to perform the following calculation:

$$(number\ of\ queued\ audio\ buffers * size\ of\ buffer) + hardware\ dependent\ latency$$

In case the hardware implements a double buffer and accesses the same memory as the application/driver data buffer usually one audio buffer is queued up, only the hardware dependent latency as found in digital transmitter or DAC's needs to be added to the audio buffer size.

If the hardware has to copy the data from the ASIO data buffers to another buffer you should look at ASIOOutputReady() whether you can use this notification mechanism to keep one audio data buffer as fixed latency.

**Input latency:**

The input latency specifies the number of sample frames an incoming audio sample is passed to the application during a buffer switch callback.

(number of queued audio buffers \* size of buffer) + hardware dependent latency

In case the hardware implements a double buffer and accesses the same memory as the application/driver data buffer usually one audio buffer is queued up, only the hardware dependent latency as found in digital receivers or ADC's needs to be added to the audio buffer size.

In order to reduce the input latency if the driver needs to copy the audio data to the ASIO data buffer, the copy operation should be executed immediately before the `bufferSwitch()` callback.

**Note:** The latency is always constant for a specific driver/hardware combination and audio data buffer size. It is not affected by the time the buffer switch callback is actually issued and it does not reflect the time the samples of the current `bufferSwitch()` callback will require to pass through the system. This is due to the fact that the audio stream is continuously processed and the audio data buffer of each buffer switch is adjacently joined to audio data from the previous buffer switch.

### **C. Driver Test Methods**

#### **1. Audio input to output placement (bounce)**

Recording audio output to input should be placed on time ( $\pm 1$  ms)

1. Make sure that the output is routed back to the input. (outboard connection or patchbay)
2. In Cubase VST place a test signal (preferably a single impulse) 1 bar right to the left locator.
3. Move the song position 1 bar before the left locator.
4. Activate Auto-Punch In record
5. Start playback, when the song position reaches the left locator recording will start.
6. Stop after some time
7. Check that the recorded signal is placed at the same time as the source signal

Whenever the recorded signal is not aligned with the source signal, the input and/or output latency has to be refined.

#### **2. Audio to MIDI sync (Audio click/MIDI click and output latency)**

Similar test as above, however a MIDI tone generator created signal should be recorded, too. We suggest a stereo recording of left channel returned audio playback and right channel from the MIDI tone generator. As audio source the audio click and for the MIDI tone generator the MIDI click can be used (both can be found in the Metronome setting window of Cubase VST).

If MIDI and audio are not aligned, please check the ASIOGetLatency results. Please note that common tone generators like Yamaha MU-80 or Roland Sound Canvas have an inherent 5 ms delay between the computer sending a MIDI event and the sound appearing at the tone modules output. Therefore it is sufficient if the audio and MIDI signal are aligned with a deviation of 5 ms. If the deviation looks more like a multiple of the audio buffer size, an incorrect result of ASIOGetLatency is likely.

#### **3. MIDI timing stability**

Same test as above. Use Audio click as signal for the audio hardware (Metronome Dialog)

MIDI sound generator created audio events should deviate within  $\pm 2$  ms from the audio click. If MIDI notes are out of range, please check the results of ASIOGetSamplePosition(), also check that the driver does not influence the systems reference timer or blocks the system for too long in other ways.

**4. MIDI/Audio drift**

Same test as above. Use Audio click as signal for the audio hardware (Metronome Dialog) MIDI notes and audio clicks should not move apart. If they do, the timestamp/sample-position result from ASIOGetSamplePos() needs to be checked.

1. MIDI/Audio offset
2. Stereo recording left channel audio click/right MIDI click
3. Bus assignment
4. Open last and first output/input bus and check for the right routing

## D. Platform/OS Differences

### MacOS 8/9

Beginning with ASIO 2.3 support for MacOS 8/9 got dropped.

### Windows

1. ASIO `bufferSwitch()` is usually implemented in its own thread, created by the driver. Thread execution is either invoked by event notification from the kernel driver or by issuing an APC in the thread context. The driver should set an appropriate high priority for the thread.

Starting with Windows Vista, Microsoft introduced the Multimedia Class Scheduler Service (MMCSS). On Windows Vista or any newer Windows version, ASIO driver threads must be in the "Pro Audio" class of MMCSS and their priority set to "CRITICAL". This guarantees the highest execution priority for the `bufferSwitch()`.

It lies within the sole responsibility of the ASIO driver to set the priorities of the threads it owns. An ASIO host shall by no means alter these priorities.

2. It is advised that the driver provides enough audio buffer size choices to avoid audio dropouts on a busy system. Due to the implementation of the thread scheduler in Windows 95/98 low priority threads can block the high priority callback thread. The time quantum is between 15 and 19 ms. We observed that a buffer size of around 50 ms (2048 samples at 48 kHz sample rate) will provide enough head room to avoid drop outs on a heavy loaded system. Though higher buffer sizes might be required depending on the constraints of the driver. The driver can implement smaller audio buffer sizes if the user can select them.

3. Win16Mutex contention should be avoided. Please keep in mind that calling the Win16 sub system can block thread execution for a long time. Several Windows 95/98 sub-systems like GDI and the window management are implemented in the Win16 sub-system which is secured against re-entrancy by the Win16Mutex. Graphic animations like the "fading" Tool Tips will execute for their entire duration in the Win16 sub system and can halt the Win16Mutex for a very long time (we observed durations of up to 1.5 s). Therefore it is necessary to avoid Win16 sub system calls in the driver threads.

4. On Windows the time reference for the timestamp is the multimedia timer, `timeGetTime()`, with a resolution of 1 ms. The time base for this timer can be obtained at the kernel level with the `VMMCall GetSystemTime` or `GetUpdatedSystemTimeAddress`.

5. The `asioMessage()` callback will be present for all Windows ASIO host applications. The initial Macintosh ASIO host application, Cubase VST 3.0 for Macintosh, did not provide this callback.

## E. AsioDriver class for the driver implementation

```
class AsioDriver {
public:
    AsioDriver(); ~AsioDriver();

    ASIOBool init(void *);
    void getDriverName(char *name);
    long getDriverVersion();
    void getErrorMessage(char *string);
    ASIOError start();
    ASIOError stop();
    ASIOError getChannels(long *numInputChannels, long *numOutputChannels);
    ASIOError getLatencies(long *inputLatency, long *outputLatency);
    ASIOError getBufferSize(long *minSize, long *maxSize, long *preferredSize, long *granularity);
    ASIOError canSampleRate(ASIOSampleRate sampleRate);
    ASIOError getSampleRate(ASIOSampleRate *sampleRate);
    ASIOError setSampleRate(ASIOSampleRate sampleRate);
    ASIOError getClockSources(ASIOClockSource *clocks, long *numSources);
    ASIOError setClockSource(long reference);
    ASIOError getSamplePosition(ASIOSamples *sPos, ASIOTimeStamp *tStamp);
    ASIOError getChannelInfo(ASIOChannelInfo *info);
    ASIOError createBuffers(ASIOBufferInfo *bufferInfos, long numChannels,
        long bufferSize, ASIOCallbacks *callbacks);
    ASIOError disposeBuffers();
    ASIOError controlPanel();
    ASIOError future(long selector, void *opt);
    ASIOError outputReady();
};
```

### Differences to the ASIO C interface

Nearly all member functions match the ASIO-API name with the ASIO-prefix removed. The exceptions are:

#### AsioDriver()

The constructor will actually just perform any system dependent tasks. On Windows it is actually the constructor for the COM object of the driver. See the separate documentation about the COM specific details on the Windows platform.

#### init(), getDriverName(), getDriverVersion()

These three member functions are a split up version of the ASIOInit() call. However not all parameter of the ASIOInit() call are provided.

Missing are asioVersion and driverVersion from the ASIODriverInfo.

init(void \*sysRef) receives on Windows the applications frame window handle.

Since the driver can not determine the host applications ASIO version until it receives the

ASIOCallbacks structure via the createBuffers() method, it has to assume ASIO 1.0 compliance. During the createBuffers() call the driver can query the host for its version via asioMessage with the **kAsioEngineVersion** selector.

**~AsioDriver()**

The desctructor implies ASIOExit(). The driver will not receive the ASIOExit() call, instead the destructor will be invoked.

## F. Sony DSD Support

Definition by Steinberg/Sony Oxford

### 1. DSD operation and buffer layout

We have tried to treat DSD as PCM and so keep a consistent structure across the ASIO interface.

DSD's sample rate is normally referenced as a multiple of 44.1Khz, so the standard sample rate is referred to as 64Fs (or 2.8224Mhz). We looked at making a special case for DSD and adding a field to the ASIOFuture that would allow the user to select the Over Sampling Rate (OSR) as a separate entity but decided in the end just to treat it as a simple value of 2.8224Mhz and use the standard interface to set it.

The second problem was the "word" size, in PCM the word size is always a greater than or equal to 8 bits (a byte). This makes life easy as we can then pack the samples into the "natural" size for the machine. In DSD the "word" size is 1 bit. This is not a major problem and can easily be dealt with if we ensure that we always deal with a multiple of 8 samples.

DSD brings with it another twist to the endianness religion. How are the samples packed into the byte. It would be nice to just say the most significant bit is always the first sample, however there would then be a performance hit on little endian machines. Looking at how some of the processing goes...

Little endian machines like the first sample to be in the Least Significant Bit,  
this is because when you write it to memory the data is in the correct format to be shifted in and out of the words.

Big endian machine prefer the first sample to be in the Most Significant Bit,  
again for the same reason.

And just when things were looking really muddy there is a proposed extension to DSD that uses 8 bit word sizes. It does not care what endianness you use.

### 2. Some notes on how to use ASIOIoFormatType.

The caller will fill the format with the request types. If the board can do the request then it will leave the values unchanged. If the board does not support the request then it will change that entry to Invalid (-1).

So to request DSD then:

```
ASIOIoFormat NeedThis={kASIODSDFormat};

if (ASE_SUCCESS != ASIOFuture(kASioSetIoFormat, &NeedThis) )
{
    If the driver did not accept one of the parameters then the whole call will fail and the failing
    parameter will have had its value changes to -1.
}
```

**Note:** Switching between the formats need to be done before the "prepared" state (see ASIO 2 finite state machine) is entered.

## **G. Microsoft Windows 64 bit**

Definition by Cakewalk / Steinberg

On Windows 64 bit systems with WOW6432 ASIO will work for both 32 bit and 64 bit host applications. It is required that an ASIO driver's COM portion is available as 32 bit and 64 bit binary.

A 32 bit host application will query the 32 bit Windows registry portion (Wow6432). A 64 bit host application will query the normal Windows registry.

An ASIO driver is required to be available as 32 bit and 64 bit COM implementation. This will ensure compatibility for 32 bit and 64 bit host applications on Windows 64 systems.

### **The Windows Registry**

The 64 bit ASIO driver needs to add the following entries to the Registry:

HKEY\_LOCAL\_MACHINE\Software\ASIO

The 32 bit ASIO driver needs to add the following entries to the Registry:

HKEY\_LOCAL\_MACHINE \Software\Wow6432Node\ASIO

If the driver uses DllRegisterServer/DllUnregisterServer (like register.cpp in the SDK) for adding itself to the Registry, Windows will automatically places the information in the right registry locations.

If the driver uses a registry file “.reg”, the registry file need to contain information for

HKEY\_LOCAL\_MACHINE\Software\ASIO

and

HKEY\_LOCAL\_MACHINE \Software\Wow6432Node\ASIO

Windows will take care about selecting the appropriate Registry information for 32 bit and 64 bit host applications.

Info:

- The CLSID of the driver can be identical for the 32 bit and the 64 bit driver.
- Both 32 bit and 64 bit registry entries will use the InprocServer32 value. There is no InprocServer64 value.