Reference Manual for the Auditory Research Soundcard Application Programming Interface

Research Computing Core
Boys Town National Research Hospital
August 6, 2010

Table of Contents

ARSC API Overview

This reference manual describes an application programming interface (API) for auditory research soundcard (ARSC) applications. The set of functions provided by this API simplify the programming of soundcard input and output for typical auditory research requirements, including (1) output that is synchronized to visual cues and (2) output that is synchronized to input. These functions can be accessed directly from a C program by linking to the arsc.lib static library or accessed indirectly through the arsc.dll dynamic library.

Not all soundcards will synchronize input and output. A few (such as DAL CardDeluxe, Echo Indigo, and Lynx L22) are able to perform synchronous input/out (with consistent latency) using standard Windows drivers. Soundcards for which ASIO drivers are available will also perform synchronous i/o. The ARSC API can be used with any Windows soundcard, even when i/o is not synchronized properly. The ARSC API works with either ASIO or WDM drivers.

The ARSC API also works with ALSA drivers under Linux. Soundcards from Echo (such as Gina24 and Indigo) have ALSA drivers and have been tested to perform synchronous i/o with consistent latency. The ALSA website http://www.alsa-project.org provides a list of supported soundcards and, if needed, soundcard firmware. Under Linux, users should belong to the “audio” group to obtain permission to use the soundcard with ARSC applications.

ARSC Function List by Category

Input / Output functions

ar_io_open	Open device for i/o
ar_io_close	Close device
ar_io_prepare	Prepare device for i/o
ar_io_start	Start i/o
ar_io_stop	Stop i/o
ar_set_fmt	Specify app's data format
ar_set_xfer	Install data-transfer functions
ar_io_cur_seg	Get current segment number
ar_close_all	Close all devices

Output-only functions

ar_out_open	Open device for output
ar_out_prepare	Prepare device for output
ar_out_seg_fill	Refill output segments

Information functions

ar_find_dev	Find device given preferences
ar_find_dev_name	Find device given name
ar_dev_name	Get device name
ar_err_msg	Get error message
ar_get_cardinfo	Get soundcard information
ar_get_fmt	Get device's data format
ar_get_gdsr	Get device's good sampling rates
ar_get_rate	Get sampling rate
ar_adjust_rate	Return closest sampling rate
ar_get_vfs	Set ADC/DAC volts-full-scale
ar_num_devs	Get number of devices
ar_set_vfs	set ADC/DAC volts-full-scale
ar_version	Get ARSC version
ar_wind	Specify window to receive messages
ar_xruns	Get number of under & over runs

ARSC Examples

Code fragments of two short examples are included in this section to demonstrate the proper calling sequence of ARSC functions for simple applications. Testing of error codes returned by the ARSC functions is recommended, but such details have been omitted in these examples for clarity.

tstfm.c – Demonstrates continuous streaming output of a frequency-modulated tone with coordinated visual cues.

tstlat.c – Demonstrates synchronous averaging of the recorded response to a repeated stimulus output.

Other programming examples are included in the ARSC source code distribution, which can be downloaded from http://audres.org/downloads/arscsc.zip.

tstout.c – Demonstrates the use of multiple waveform segments to control the timing of a two-interval stimulus output with coordinated visual cues.

tstsio.c – Demonstrates synchronous averaging using an older API called SIO, which is supported in the ARSC library for backward compatibility.

tst3ch.c – Demonstrates *three-channel* continuous streaming output. Similar to tstfm.c

Streaming Output

This code fragment provides an example of continuous streaming output. For a complete version, see the file tstfm.c.

```
#include "arsclib.h"

double rate = 44100;    // sampling rate
int dvid = 0;           // device identifier
int stop = 0;           // terminates output
int nsmp = 2500;        // samples per segment
int nseg = 2;           // number of segments
long *stm[2];           // list of stimulus buffers

void
tone_seg(int seg)       // output callback function
{
    // fill output buffer here
}

void
test_tone(double f, double t) // tone frequency and duration
{
    long siz[2], fmt[2];
    void *out[4];
    int nchn = 2;
    int nswp = 0;       // 0 is for streaming

    // set up i/o

    dvid = ar_find_dev(ARSC_PREF_NONE);
    ar_out_open(dvid, rate, nchn);
    fmt[0] = ARSC_DATA_I4;    // 32-bit integer data
    fmt[1] = 0;               // samples non-interleaved
    ar_set_fmt(dvid, fmt);    // specify data format

    // set up stimulus

    rate = ar_get_rate(dvid);
    stm[0] = calloc(nsmp, sizeof(long));
    stm[1] = calloc(nsmp, sizeof(long));
    out[0] = out[1] = stm[0]; // put stm on left
    out[2] = out[3] = stm[1]; // put stm on right
    siz[0] = siz[1] = nsmp;

    // perform i/o

    ar_set_xfer(dvid, NULL, tone_seg); // specify callback
    ar_out_prepare(dvid, out, siz, nseg, nswp); // prepare
    ar_io_start(dvid);  // start

    while (!stop) {     // loop until user terminates
      chk_msg();        // check for user input
```

```
      cue();            // provide visual cue
      ar_io_cur_seg(dvid);
    };
    ar_io_close(dvid);

    // clean up stimulus

    free(stm[0]);
    free(stm[1]);
}
```

Synchronous Input/Ouput

This code fragment provides an example of simultaneous input and output. For a complete version see the file tstlat.c.

```
#include "arsclib.h"

int
test_io()
{
    double sr = 48000;  // sampling rate
    int d, m;
    long *sl, *rl, fmt[2];
    void *od[1], *id[1];
    int io_dev = 0;
    long nseg = 1;      // number of segments
    long nsmp = 4800;   // samples per segment
    long nswp = 1;      // number of sweeps

    // set up i/o

    d = io_dev;
    ar_io_open(d, sr, 1, 1);                    // one-channel i/o
    fmt[0] = ARSC_DATA_I4;                      // int32
    fmt[1] = 0;                                 // non-interleaved
    ar_set_fmt(d, fmt);
    sr = ar_get_rate(d);
    id[0] = rl = calloc(nsmp, sizeof(long));    // input data
    od[0] = sl = calloc(nsmp, sizeof(long));    // output data
    sl[0] = (long) pow(2, 31) - 1;              // click

    // perform i/o

    ar_io_prepare(d, id, od, &nsmp, nseg, nswp);
    ar_io_start(d))
    while (ar_io_cur_seg(d) < nseg) {
        continue;
    }

    // clean up i/o

    ar_io_stop(d);
    ar_io_close(d);
    free(rl);
    free(sl);
}
```

ARSC Functions

The following pages provide a description of each ARSC function in alphabetical order.

ar_adjust_rate

Adjust desired sampling rate to nearest usable rate

```
double ar_adjust_rate(
    long dev,
    double rate
    );
```

Parameters

`dev` Device identifier.

Return Value

The nearest sampling rate possible on the specified soundcard device.

Remarks

If the soundcard device can’t provide the desired rate, then it will be set to the closest rate possible.

See Also

`ar_io_get_rate, ar_io_set_rate`

ar_close_all

Close all devices

```
void ar_close_all(
    );
```

Parameters

None.

Return Value

None

Remarks

Use this function to close all open ASRC devices before program termination.

ar_dev_name

Get device name

```
long ar_dev_name(
    long dev,
    char *name,
    long len
    );
```

Parameters

`dev`	Device identifier.
`name`	Array to receive name of device.
`len`	Length of `name` array. Recommended length is `ARSC_NAMLEN`.

Return Value

Error code.

Remarks

Returns the name of the device associated with the specified identifier. Device may be opened or closed at the time this function is called.

ar_err_msg

Get error message

```
void ar_err_msg(
    long err,
    char *msg,
    long len
    );
```

Parameters

`err`	Error code.
`msg`	Array to receive name of device.
`len`	Length of `msg` array. Recommended length is `ARSC_MSGLEN`.

Return Value

None.

Remarks

Retrieves an error message for a specified error code.

ar_find_dev

Find device given preferences

```
long ar_find_dev(
    long flags
    );
```

Parameters

`dev` Device identifier.

`flags` Hints about desired device, which may be set to one of the following values.

- ARSC_PREF_NONE
- ARSC_PREF_SYNC
- ARSC_PREF_ASIO
- ARSC_PREF_ALSA
- ARSC_PREF_WIND

Return Value

Device identifier

Remarks

Retrieves a device identifier for a soundcard device that matches, if possible, with the indicated preference. Defaults to 0, which indicates the system default soundcard device.

ar_find_dev_name

Find device given name

```
long ar_find_dev_name(
    char *name
    );
```

Parameters

`name` Device name.

Return Value

Device identifier

Remarks

Retrieves a device identifier for a soundcard device which the specified name. Defaults to 0, which indicates the system default soundcard device.

ar_get_fmt

Get device's data format

```
long ar_get_fmt(
    long dev,
    long *fmt
    );
```

Parameters

`dev` Device identifier.

`fmt` Array of parameters to describe the format of the soundcard device's sample data. The length of this array is 2.

- `fmt[0]` *data type* — one of the following values:
  ARSC_DATA_I2 – 16-bit integer
  ARSC_DATA_I4 – 32-bit integer
  ARSC_DATA_F4 – 32-bit floating-point
  ARSC_DATA_F8 – 64-bit floating-point
- `fmt[1]` *channel interleave* — It's value will be 0 to indicate *non-interleaved* sample data or 1 to indicate *interleaved* sample data.

Return Value

Error code.

Remarks

Retrieves the format of the soundcard device's sample data, which can be *interleaved* or *non-interleaved* and one of four data types. The ARSC input/output functions will supply any conversions necessary to make the application's sample data format compatible that the format expected by the soundcard device.

See Also

`ar_set_fmt`

ar_get_cardinfo

Get soundcard information

```
long ar_get_cardinfo(
    long dev,
    CARDINFO *ci
    );
```

Parameters

`dev` Device identifier.

`ci` Pointer to CARDINFO structure:

```
typedef struct {
    char    name[MAX_CT_NAME];  /* card name            */
    int     bits;               /* precsion             */
    int     left;               /* MSB shift            */
    int     nbps;               /* bytes per sample     */
    int     ncad;               /* # A/D channels       */
    int     ncda;               /* # D/A channels       */
    int     gdsr;               /* good sampling rates  */
    double  ad_vfs[MAXNCH];     /* A/D volts full scale */
    double  da_vfs[MAXNCH];     /* D/A volts full scale */
} CARDINFO;
```

Return Value

The sequential number of the card type with the soundcard features that are returned in the CARDINFO structure.

Remarks

This soundcard information is based on hardcoded defaults for a few soundcards, which may be overridden or extended by the registry (on Windows) or the configuration file /usr/local/etc/arscrc (on Linux).

ar_get_gdsr

Get device's good sampling rate

```
long ar_get_gdsr(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

Bitwise list of good sampling rates or zero if device identifier is out of range or device has not yet been opened. Bits 0 – 26 in the returned value correspond to these sampling rates: 4000, 5512, 6000, 8000, 8269, 10000, 11025, 12000, 16000, 16538, 20000, 22050, 24000, 25000, 32000, 33075, 44100, 48000, 50000, 64000, 88200, 96000, 100000, 128000, 176400, 192000, 200000.

Remarks

Device should be opened prior to calling this function, but need not be still open when this function is called.

See Also

`ar_get_rate`

ar_get_rate

Get sampling rate

```
double ar_get_rate(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

The sampling rate currently being used by the specified soundcard device.

Remarks

The soundcard device's sampling rate will differ from the rate requested by the application when the device was opened. If the soundcard device can't provide the requested rate, then it will be set to the closest rate possible.

See Also

`ar_io_adjust_rate`, `ar_io_set_rate`

ar_get_sfs

Set sample-full-scale value for floating-point transfer buffers

```
void ar_get_sfs(
    long dev,
    double *i_sfs,
    double *o_sfs
    );
```

Parameters

`dev`	Device identifier.
`i_vfs`	input sample-full-scale.
`o_vfs`	output sample-full-scale.

Return Value

None.

Remarks

When ARSC inputs samples to a floating-point buffer or outputs samples from a floating-point buffer the floating-point value that corresponds to full-scale voltage at the convertor is i_vfs or o_vfs, respectively. The same sample-full-scale parameter is used for both single-precision and double-precision floating-point buffers. The default value of sample-full-scale value is 1 for both i_vfs and o_vfs.

See Also

`ar_set_sfs, ar_set_fmt, ar_set_xfer`

ar_get_vfs

Get ADC and DAC volts-full-scale

```
void ar_get_vfs(
    long dev,
    double *da_vfs,
    double *ad_vfs
    );
```

Parameters

`dev`	Device identifier.
`da_vfs`	DAC volts-full-scale array (size = 8).
`ad_vfs`	ADC volts-full-scale array (size = 8).

Return Value

None.

Remarks

If the name of the soundcard is one of a few recognized by the ARSC library (i.e., CardDeluxe, Gina3G, Indigo, or Layla3G), then default values for the full-scale voltage of the A/D and D/A converters will be set automatically. Otherwise, these values can be set by the `ar_set_vfs` function. The VFS values do not affect the operation of any ARSC input/output function. They are provided only for informational purposes.

See Also

`ar_set_vfs`

ar_io_close

Close device

```
long ar_io_close(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

Error code.

Remarks

Closes the specified soundcard device.

See Also

`ar_io_open, ar_close_all`

ar_io_cur_seg

Get current unwrapped segment

```
long ar_io_cur_seg(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

Returns the number of the input/output segment which has most recently been completed. This unwrapped segment number is not reset to 0 at the beginning of each new sweep. Instead it continues to increment with each successive sweep. The number of the segment within the current sweep can be recovered by taking the unwrapped segment number modulo the number of sweeps per segment.

Remarks

This function can be used to synchronize visual cues to the user with input/output segment boundaries. It is important that the application poll the `ar_io_cur_seg` function often because it also performs important input/output processing functions. This function needs to be called at least once during each input/output segment to insure the successive segment are queued for the soundcard device.

Under Windows, one way to implement calling of the `ar_io_cur_seg` function with sufficient frequency is to respond to WM_ARSC messages.

See Also

`ar_set_wind`

ar_io_open

Open device for i/o

```
long ar_io_open(
    long dev,
    double rate,
    long in_chan,
    long out_chan
    );
```

Parameters

`dev` Device identifier. This can be an identifier provided by the ar_find_dev or ar_find_dev name functions. It's value can also be zero to specify use of the system default soundcard.

`rate` Desired sampling rate in samples/second. If the soundcard can't provide the requested rate, the closes possible sampling rate will be used. The actual sampling rate can be obtained by calling ar_get rate.

`in_chan` Desired number of input channels. This indicates the number of input channels that will be expected in the application's input data buffers that are specified to ar_io_prepare. If the requested number of channels is more than the soundcard can provide, any excess channel data will be filled with zero. The value of in_chan can be set to zero if no input is desired.

`out_chan` Desired number of output channels. This indicates the number of output channels for which data will be provided in the application's output data buffers that are specified to ar_io_prepare. If the requested number of channels is more than the soundcard can provide, any excess channel data will be ignored. The value of out_chan can be set to zero if no input is desired.

Return Value

Error code.

Remarks

If calling ar_io_open is successful, then call ar_io_prepare to specify the input/output buffers and ar_io_start to initiate the input/output process. When both input and output is specifed, these processes will be synchronized, if possible.

See Also

`ar_io_close, ar_io_prepare, ar_out_open`

ar_io_prep

Prepare device for i/o

```
long ar_io_prepare(
    long dev,
    void *in_data[],
    void *out_data[],
    long size[],
    long nseg,
    long tseg
    );
```

Parameters

`dev`	Device identifier.
`in_data`	List of pointers to the input data buffers for each segment. The value in_data may be NULL to indicate no input. Or, the value of any pointer in the list may be NULL to indicate no input for a particular segment.
`out_data`	List of pointers to the output data buffers for each segment. The value out_data may be NULL to indicate no output. Or, the value of any pointer in the list may be NULL to indicate no output for a particular segment.
`size`	List of sizes indicating the length of each segment.
`nseg`	The number of input/output segments in each sweep.
`tseg`	The total number of segments, which equals the total number of sweeps times `nseg`. Set `tseg` to zero for continuous output.

Return Value

Error code.

Remarks

The data type of the input/output buffers can be *long* or *short* and the channels can be *interleaved* or *non-interleaved* depending on the format specifed through the ar_set_fmt function. If the data buffers are *non-interleaved*, then the in_data and out_data lists must provide pointers to each channel for each segment. Whenever the number of channels is greater than one, either the data or the pointers are interleaved. The input/output process is not initiated until ar_io_start is called. Output buffers are filled by this function, so call ar_set_xfer before this function.

See Also

`ar_io_start, ar_io_prepare, ar_out_prepare, ar_set_xfer`

ar_io_prepare

Prepare device for i/o

```
long ar_io_prepare(
    long dev,
    void *in_data[],
    void *out_data[],
    long size[],
    long nseg,
    long nswp
    );
```

Parameters

`dev`	Device identifier.
`in_data`	List of pointers to the input data buffers for each segment. The value in_data may be NULL to indicate no input. Or, the value of any pointer in the list may be NULL to indicate no input for a particular segment.
`out_data`	List of pointers to the output data buffers for each segment. The value out_data may be NULL to indicate no output. Or, the value of any pointer in the list may be NULL to indicate no output for a particular segment.
`size`	List of sizes indicating the length of each segment.
`nseg`	The number of input/output segments in each sweep.
`nswp`	The total number of input/output sweeps. Set `nswp` to zero for continuous output.

Return Value

Error code.

Remarks

The data type of the input/output buffers can be *long* or *short* and the channels can be *interleaved* or *non-interleaved* depending on the format specifed through the ar_set_fmt function. If the data buffers are *non-interleaved*, then the in_data and out_data lists must provide pointers to each channel for each segment. Whenever the number of channels is greater than one, either the data or the pointers are interleaved. The input/output process is not initiated until ar_io_start is called. Output buffers are filled by this function, so call ar_set_xfer before this function.

See Also

`ar_io_start, ar_io_prepare, ar_out_prepare, ar_set_xfer`

ar_io_start

Start i/o

```
long ar_io_start(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

Error code.

Remarks

The application must call ar_io_prepare or ar_io_open before calling ar_io_start.

See Also

`ar_io_prepare, ar_out_prepare`

ar_io_stop

Stop i/o

```
long ar_io_stop(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Value

Error code.

Remarks

Use this function to close all stop the input/output on a specified device immediately. Because the soundcard input/output also stops when the specified number of sweeps has completed or if the device is closed while still performing input/output, it is never necessary to call ar_io_stop. On the other hand, it is not an error to call ar_io_stop when input/output has already been stopped, provided that device is open. Calling ar_io_stop also *unprepares* the input/output process, to it can not be restarted without again calling ar_io_prepare or ar_out_prepare.

See Also

`ar_io_prepare, ar_io_start, ar_out_prepare`

ar_num_devs

Get number of devices.

```
long ar_num_devs(
    );
```

Parameters

None.

Return Value

Device count

Remarks

Returns the total number of soundcard devices currently present.

See Also

`ar_find_dev`

ar_out_open

Open device for output

```
long ar_out_open(
    long dev,
    double rate,
    long out_chan
    );
```

Parameters

`dev` Device identifier. This can be an identifier provided by the ar_find_dev or ar_find_dev name functions. It's value can also be zero to specify use of the system default soundcard.

`rate` Desired sampling rate in samples/second. If the soundcard can't provide the requested rate, the closes possible sampling rate will be used. The actual sampling rate can be obtained by calling ar_get rate.

`out_chan` Desired number of output channels. This indicates the number of output channels for which data will be provided in the application's output data buffers that are specified to ar_io_prepare. If the requested number of channels is more than the soundcard can provide, any excess channel data will be ignored.

Return Value

Error code.

Remarks

Calling ar_out_open is the same as calling ar_io_open with the number of input channels set to zero. It simplifies programming when no soundcard input is needed.

See Also

`ar_io_open`

ar_out_prepare

Prepare device for output

```
long ar_out_prepare(
    long dev,
    void *out_data[],
    long size[],
    long nseg,
    long nswp
    );
```

Parameters

`dev`	Device identifier.
`out_data`	List of pointers to the output data buffers for each segment. The value of any pointer in the list may be NULL to indicate no output for a particular segment.
`size`	List of sizes indicating the length of each segment.
`nseg`	The number of input/output segments in each sweep.
`nswp`	The total number of input/output sweeps. The value of nswps can be zero to indicate continuous output.

Return Value

Error code.

Remarks

Calling ar_out_prepare is the same as calling ar_io_ prepare with a NULL value for the list of input data buffers. It simplifies programming when no soundcard input is needed.

See Also

`ar_io_prepare`

ar_out_seg_fill

Refill output segments

```
long ar_out_seg_fill(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Values

Error code.

Remarks

This function can be used to change the contents of output buffers that have already been queued for output. It is only useful when the number of segments is greater than 2 and an output callback function has been specified through ar_set_xfer. Calling ar_out_seg_fill causes the applications output callback function to be called again for every segment except the segment currently being processed by the soundcard device and the next segment, in case a segment transition occurs during before the callback function has finished filling the output buffer. The ar_out_seg_fill function may be useful in streaming applications with many small output segments in order to reduce the delay in transitioning to a new type of output.

See Also

`ar_set_xfer`

ar_set_fmt

Specify app's data format

```
long ar_set_fmt(
    long dev,
    long *fmt
    );
```

Parameters

`dev` Device identifier.

`fmt` Array of parameters to describe the format of the application's sample data. The length of this array is 2.

- `fmt[0]` *data type* — one of the following values:
  ARSC_DATA_I2 – 16-bit integer
  ARSC_DATA_I4 – 32-bit integer
  ARSC_DATA_F4 – 32-bit floating-point
  ARSC_DATA_F8 – 64-bit floating-point
- `fmt[1]` *channel interleave* — It's value can be 0 to indicate *non-interleaved* sample data or 1 to indicate *interleaved* sample data.

Return Value

Error code.

Remarks

Sets the format of the soundcard device's sample data, which can be *interleaved* or *non-interleaved* and one of four data types. The ARSC input/output functions will supply any conversions necessary to make the application's sample data format compatible that the format expected by the soundcard device.

See Also

`ar_get_fmt`

ar_set_latency

Set ASIO device latency offset and return current latency setting

```
long ar_set_latency(
    long dev,
    long nsmp
    );
```

Parameters

`dev` Device identifier.

`nsmp` Number of samples added to ASIO latency. Number may be negative to decrease latency. Special value nsmp=ASIO_GET_LATENCY or nsmp=9999 bypasses setting just returns current value.

Return Value

Number of samples of current latency offset

Remarks

Some ASIO device drivers center the loopback impulse response at zero latency, which is less than the WDM loopback latency. The ar_set_latency function allows the ASIO interface to have the same latency as the WDM interface.

ar_set_sfs

Set sample-full-scale value for floating-point transfer buffers

```
void ar_set_sfs(
    long dev,
    double *i_sfs,
    double *o_sfs
    );
```

Parameters

`dev`	Device identifier.
`dev`	Device identifier.
`i_sfs`	input sample-full-scale.
`o_sfs`	output sample-full-scale.

Return Value

None.

Remarks

When ARSC inputs samples to a floating-point buffer or outputs samples from a floating-point buffer the floating-point value that corresponds to full-scale voltage at the convertor is i_vfs or o_vfs, respectively. The same sample-full-scale parameter is used for both single-precision and double-precision floating-point buffers. The default value of sample-full-scale value is 1 for both i_vfs and o_vfs.

See Also

`ar_get_sfs, ar_set_fmt, ar_set_xfer`

ar_set_vfs

Set ADC and DAC volts-full-scale

```
void ar_set_vfs(
    long dev,
    double *da_vfs,
    double *ad_vfs
    );
```

Parameters

`dev`	Device identifier.
`da_vfs`	DAC volts-full-scale array (size = 8).
`ad_vfs`	ADC volts-full-scale array (size = 8).

Return Value

None.

Remarks

If the name of the soundcard is one of a few recognized by the ARSC library (i.e., CardDeluxe, Gina24, Indigo, or WaveTerminal), then default values for the full-scale voltage of the A/D and D/A converters will be set automatically. Otherwise, these values can be set by the application. The VFS values do not affect the operation of any ARSC input/output function. They are provided only for informational purposes.

See Also

`ar_get_vfs`

ar_set_xfer

Specify app data transfer functions

```
long ar_set_xfer(
    long dev,
    void (*in_xfer)(long),
    void (*out_xfer)(long)
    );
```

Parameters

`dev` Device identifier.

`in_xfer` Input callback function which will be called for each input segment to allow the application to retrieve data from the input buffer. The unwrapped segment number is provided as an argument to this function. The value of in_xfer can be NULL if no input callback function is desired.

`out_xfer` Output callback function which will be called for each output segment to allow the application to fill the output buffer with data. The unwrapped segment number is provided as an argument to this function. The value of out_xfer can be NULL if no output callback function is desired.

Return Value

Error code.

Remarks

The function allows streaming input and/or output to be implemented. The input/output process can be made continuous by setting the number of sweeps to zero. The segment number within the current sweep is the unwrapped segment number provided to the callback function modulo the number of segments per sweep. To reduce the delay in switching to new output when many output segments have been queued, the ar_out_seg_fill function can be called to refill output data buffers with new data. Call this function prior to calling ar_io_prep or ar_io_prepare to allow output buffers to be filled.

See Also

`ar_io_prep, ar_io_prepare, ar_out_seg_fill`

ar_wind

Specify a window receives messages

```
void ar_wind(
    long hwind
    );
```

Parameters

`hwind` Handle to window that is to receive WM_ARSC messages.

Return Values

None.

Remarks

The specified window will receive WM_ARSC messages whenever an input or output segment has completed. The wParam associated with this message will be 1 for an input segment and 2 for an output segment. The lParam will have the value of the device identifier. The ar_io_cur_seg function can be called in response to this message to insure that it is called at least once per segment. Visual cue might also be provided to the user at this time that indicate a segment transition.

ar_version

Get ARSC version.

```
char *ar_version(
    );
```

Parameters

None.

Return Values

ARSC version string.

Remarks

The returned value is a string containing the ARSC version number and date, similar to the line below.

```
ARSC version 0.01, 24-Apr-05
```

ar_xruns

Get combined number of underruns and overruns.

```
void ar_xruns(
    long dev
    );
```

Parameters

`dev` Device identifier.

Return Values

-1 device identifier out of range
-2 device not open
>0 number of xruns since device was opened

Remarks

Because the ARSC functions handling input and output simultaneously, underruns and overruns are also occur simultaneously. These events are counted together as xruns.

ARSC DLL

The dynamic library for Windows, arsc.dll, contains the same static library functions described in the previous section; however, the Windows DLL calling convention is used in arsc.dll instead of the standard C calling convention. For example, the function declared in the static library as "`long ar_find_dev()`," is declared in the DLL as "`WIN32DLL_API long STDCALL ar_find_dev()`."

The DLL also contains additional functions to help support its use with languages other than C.

ar_wait_seg

Wait until current segment has finished.

```
WIN32DLL_API long STDCALL
ar_io_wait_seg(
    long dev
    );
```

Ordinarily the calling program polls to see when a segment has finished. This function provides a different methodology where the looping is done internal to the DLL, then control is released to the calling program only after the segment is finished.

Because this function resides within a windows DLL, an escape mechanism has been provided. Specifically, hit the ESC key any time the sound engine is running to stop.

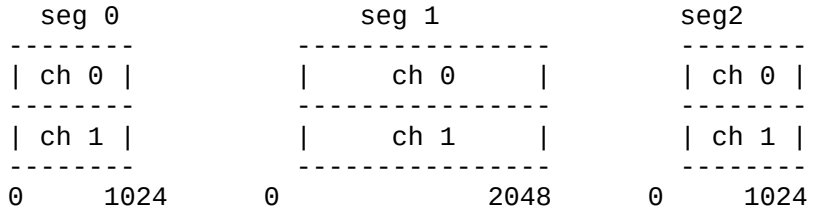
ar_io_prepare_vb

Wrapper to help Visual Basic perform I/O with ARSC.

```
WIN32DLL_API long STDCALL
ar_io_prepare_vb (
    long dev,                 // device identifier
    void *in_data,            // pointer to input data
    void *out_data,           // pointer to output data
    long size[],              // array of segment sizes
    long nseg,                // number of segments
    long nswp                 // number of sweeps
    );                        // returns error code
```

This wrapper assumes a pointer to a single block of memory then re-defines it to the ar_io_prepare() arguments. This has been tested and works well with VB.NET, MatLab, and even C#.

A visual interpretation may help to clarify how this works. Consider a scenario of 3 segments (0,1,2) and 2 channels (0,1) and that we are considering output only. The first segment is 1024; the second is 2048; the third is 1024 samples.

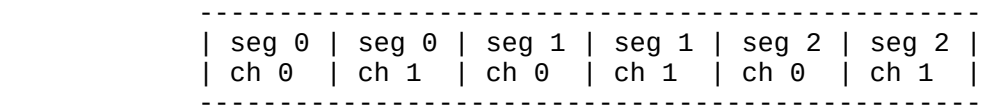


** FIGURE 1 **

Figure 1 shows the segments as separate entities. The underlying C library only needs pointers, so this layout shows the segments in a logical train of left to right.

Channels are interleaved in the underlying code. It is useful to think of channels as being layers of the segment.

This next figure shows how this function takes the data as one contiguous block.



** FIGURE 2 **

The sizes array is populated with segment lengths. This is independent of the number of channels. That is, if you have 1 or 1000 channels, the segment length must be the same.

For this example, sizes[3] = { 1024, 2048, 1024 }

ar_out_prepare_vb

Wrapper to ar_io_prepare_vb() to skip input channels.

```
WIN32DLL_API long STDCALL
ar_out_prepare_vb (
    long dev,                 // device identifier
    void *out_data,           // pointer to output data
    long size[],              // array of segment sizes
    long nseg,                // number of segments
    long nswp                 // number of sweeps
    );                        // returns error code
```

ar_set_xfer_stdcall

Special function for stdcall transfer (xfer) functions.

```
WIN32DLL_API long STDCALL
ar_set_xfer_stdcall (
    long dev,
    void (STDCALL *in_xfer)(long),
    void (STDCALL *out_xfer)(long)
    );
```

ar_err_msg_matlab

Help MATLAB access error message.

```
WIN32DLL_API char * STDCALL
ar_err_msg_matlab (
    long code            // error code
    );
```

ar_dev_name_matlab

Help MATLAB access device name.

```
WIN32DLL_API char * STDCALL
ar_dev_name_matlab (
    long dev             // device identifier
    );
```

ar_version_DLL

Returns the DLL version and the ARSC version.

```
WIN32DLL_API long STDCALL
ar_version_dll (
    char *astrVersionInfo,    // pointer to string
    long len                  // Length of string
    );
```

ar_fill_tone

Fill a memory block with a tone of frequency.

```
WIN32DLL_API long STDCALL
```

```
ar_fill_tone (
    long *ptrBlock,                 // Output
    long aintBlockSize,        // Memory block size (bytes)
    double adblDesiredFrequency,    // Frequency (Hz)
    double adblSampleRate,          // e.g. 44100.
    double adblAmplitude            // sine wave amplitude
    );                              // returns error code
```

SIO API

The SIO functions provide an alternative synchronous-I/O API that uses floating-point buffers. The SIO functions are implemented on top of the ARSC functions. The dynamic library file sio.dll functions contains the entire set of ARSC functions, in addition to the SIO functions listed below. However, the ARSC_DLL helper functions are not included.

sio_open

Initializes soundcard and internal SIO variables.

```
int sio_open(
    );          // returns non-zero if successful
```

sio_close

Terminate I/O and free any allocated resources.

```
void sio_close(
);
```

sio_get_nioch

Returns the number of I/O channels available on the soundcard.

```
int sio_get_nioch(
    int *ni,    // Number of input channels returned here
    int *no     // Number of output channels returned here
    );          // Device open?
```

sio_get_device

Obtain short description of I/O device.

```
int sio_get_device(
    char *s     // Device description returned here
    );          // Device open?
```

sio_set_device

Select I/O device.

```
int sio_set_device(
    int n      // card number
    );         // returns card number
```

sio_get_info

Obtain one-line of information about I/O device.

```
int sio_get_info(
    char *s    // Device information returned here
    );         // Device open?
```

sio_get_vfs

Obtain full-scale voltage of ADCs and DACs.

```
int sio_get_vfs(
    double *ad_vfs, // ADC volts-full-scale returned
    double *da_vfs  // DAC volts-full-scale returned
    );              // Device open?
```

sio_set_size

Specify size of buffers and gap between buffers.

```
void sio_set_size(
    int ns,    // size of I/O buffers (samples)
    int ng,    // size of gap between I/O buffers (samples)
    double pg  // size of gap before I/O (seconds)
    );
```

sio_set_output

Specify output buffers.

```
void sio_set_output(
    int nch,        // number of output channels to be used
    int bpc,        // number of buffers per output channel
    int *inc,       // increments for each output buffer
    float **bpt     // pointers to each output buffer
    );
```

sio_set_input

Specify input buffers.

```
void sio_set_input(
    int nch,        // number of input channels to be used
    int bpc,        // number of buffers per input channel
    int *inc,       // increments for each input buffer
    float **bpt     // pointers to each input buffer
    );
```

sio_set_average

Specify averaging.

```
void sio_set_average(
    float **apt,    // pointers to accumulate buffers
    int *acp        // number of sweeps in each buffer
    );
```

sio_set_rate

Specify sampling rate.

```
double sio_set_rate(
    double r   // desired rate (samples/second)
    );         // returns actual rate (samples/second)
```

sio_set_att_in

Specify attenuation on input all channels.

```
double sio_set_att_in(
    double a   // desired input attenutation (dB)
    );         // returns actual input attenuation (dB)
```

sio_set_att_out

Specify attenuation on output all channels.

```
double sio_set_att_out(
    double a   // desired output attenutation (dB)
    );         // returns actual output attenuation (dB)
```

sio_set_vfs

Specify full-scale voltage on ADCs and DACs

```
void sio_set_vfs(
    double *ad_vfs, // volts-full-scale for each ADC
    double *da_vfs  // volts-full-scale for each DAC
    );
```

sio_set_escape

Specify callback function for early I/O termination. The callback function `esc` requests an escape by returning a nonzero value.

```
void sio_set_escape(
    int (*esc)()    // escape callback function
    );
```

sio_io

Perform input/output/averaging.

```
void sio_io(
    int nskip, // samples to skip before averaging
    int nswps, // maximum total sweeps
    int navgs, // maximum sweeps averaged
    int nrejs  // maximum rejected
    );
```

sio_io_chk

Perform input/output/averaging with callback. The function `resp_check` is called at the completion of each sweep with a pointer to an “escape” flag `esc`. The `resp_check` function assesses the status of the average and requests termination by setting `esc` to a nonzero value.

```
void sio_io_chk(
    void (*resp_check)(int *esc) // sweep callback function
    );
```

sio_set_latency

Set ASIO internal latency.

```
int sio_set_latency(
    int nsmp
    );
```

sio_get_cardinfo

Get card info.

```
CARDINFO sio_get_cardinfo(
    int ct        // card type
    );
```

sio_set_cardinfo

Set card info.

```
void sio_set_cardinfo(
    CARDINFO ci,  // card info
    int ct        // card type
    );
```

The CARDINFO structure:

```
typedef struct {
    char    name[MAX_CT_NAME];  /* card name           */
    int     bits;               /* precsion            */
    int     left;               /* MSB shift           */
    int     nbps;               /* bytes per sample    */
    int     ncad;               /* # A/D channels      */
    int     ncda;               /* # D/A channels      */
    int     gdsr;               /* good sampling rates */
    double  ad_vfs[MAXNCH];     /* A/D volts full scale */
    double  da_vfs[MAXNCH];     /* D/A volts full scale */
} CARDINFO;
```

ARSC Pre-defined Constants

```
ARSC_MSGLEN         80
ARSC_NAMLEN         40

ARSC_PREF_NONE      0
ARSC_PREF_SYNC      1
ARSC_PREF_ASIO      2
ARSC_PREF_ALSA      3
ARSC_PREF_WIND      4

ARSC_DATA_UNKNOWN   0
ARSC_DATA_U1        1
ARSC_DATA_I2        2
ARSC_DATA_P3        3
ARSC_DATA_I4        4
ARSC_DATA_X3        5
ARSC_DATA_F4        6
ARSC_DATA_M1        7
ARSC_DATA_F8        8

WM_ARSC             (WM_USER+555)
```

ARSC Error Codes

These error codes may be returned by one or more of the ARSC functions. The corresponding error message can also be obtained by using the ar_error_message function.

```
  0  no errors
  1  device identifier out of range
  2  device not open
101  io_open - MMSYSERR = unspecified error
102  io_open - MMSYSERR = device ID out of range
103  io_open - MMSYSERR = driver enable failed
104  io_open - MMSYSERR = device already allocated
105  io_open – MMSYSERR = unknown error
106  io_open - MMSYSERR = no device driver present
107  io_open - MMSYSERR = memory allocation error
108  io_open - WAVERR = unsupported wave format
109  io_open – WIND unknown device identifier
110  io_open - WIND open error
120  io_open – ASIO open error
130  io_open - ALSA open error
201  io_prepare - device not opened
202  io_prepare - unsupported data format conversion
203  io_prepare - size matters
204  io_prepare - too many segments per sweep
205  io_prepare - too many sweeps
206  io_prepare - failed low level prepare
301  io_start - device not prepared
401  io_set_fmt - NULL format
402  io_set_fmt - unsupported data format
403  io_set_fmt - unsupported interleave
404  io_set_fmt - unsupported format conversion
```