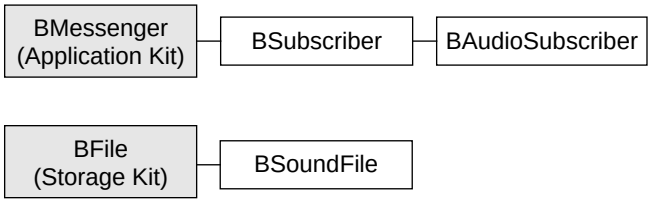
5 The Media Kit

Media Kit Inheritance Hierarchy



5 The Media Kit

The Media Kit gives you tools that let you generate, examine, manipulate, and realize (or *render*) medium-specific data in real-time. It also lets you synchronize the transmission of data to different media devices, allowing you to build applications that can easily incorporate and coordinate audio and video (for example).

There are three layers in the Media Kit:

- Through the classes provided by the *module* layer, you create data-generating and -manipulating modules that can be plugged into each other to create an ever-narrowing data-processing tree. The tree terminates at a global scheduling object. Every application can have its own processing tree, or it can share branches or even individual modules with other applications. Synchronization between data from different media is handled by the scheduler: All you have to do is define and hook up the data-processing modules.

- At the *subscriber* layer are classes that let you talk directly to the media servers that are provided by the Kit. For each distinct medium there's a distinct server—but there's only one server per medium per computer. Corresponding to each server is a BSubscriber-derived class. Through instances of these classes you can receive and send data to the server.

- The *stream* layer lets you access the "data-streaming" facilities of the Kit. A data stream (as used by the Kit) is a sequence of programming entities that each get access to a set of data buffers. There are no servers or other media-specific constraints at this layer; you can actually use the classes in the stream layer to design a streamlined, intra-computer, data-transmission application (currently, streams can't broadcast over a network).

These three layers are interconnected: The module layer is built on top of the subscriber layer, which is built on top of the stream layer. Most high-level media applications will want to use the module layer exclusively. If you need more control or greater efficiency, head for the subscriber layer. The stream layer is the least useful to media applications, but, as mentioned above, it may find a home in applications—media-specific or not—that want to set up an efficient, real-time data pipeline.

Currently, only the subscriber and stream layers of the Media Kit are implemented, and, in this release, only the subscriber layer is documented.

At the subscriber layer, the Kit provides two classes:

- BSubscriber defines the basic rules to which all subscribers must adhere. If you want to use the subscriber layer, this is where you start to learn about it.
- BAudioSubscriber provides additional functionality that speaks directly to the *Audio Server*. The Audio Server is a background application that manages sound data that arrives through the microphone or line-in jacks, and that sends sound data to the internal speaker and line-out jacks. All subscribers that you create, for now, will be instances of BAudioSubscriber.

The Kit also provides a BSoundFile class that lets you read the data in a sound file, and global functions that let you play sound files.

BAudioSubscriber

Derived from: public BSubscriber

Declared in: <media/AudioSubscriber.h>

Overview

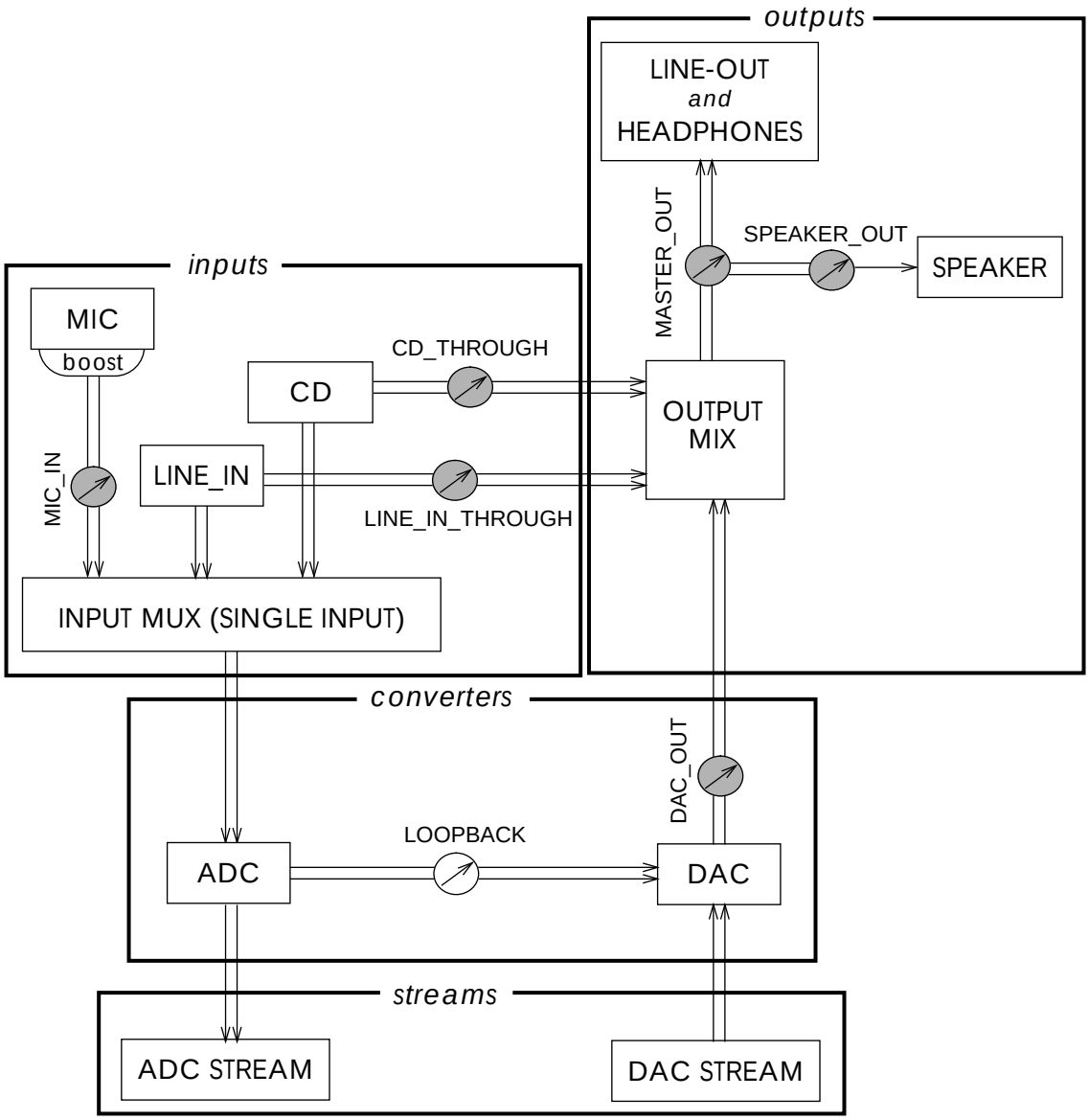
BAudioSubscriber objects perform two functions:

- They let your application receive, process, and broadcast sound data.
- They let you control certain parameters—such as volume and muting—of the sound hardware.

Ultimately, the first point is the more interesting of the two: Recording, generating, and manipulating sound data is a bit more amusing than simply setting the volume levels of the hardware devices. But to understand how and what data is received by your BAudioSubscriber objects, and what happens when you broadcast data through an object, you should first understand how the hardware is configured. The next section examines the sound hardware; following that is a description of the sound data that appears in your application.

Sound Hardware

The sound hardware consists of a number of physical devices (jacks, converters, and the like), a signal path that routes audio data between these devices, and "control points" along the signal path that let you adjust the format and flow of the audio data. These elements are depicted in the following illustration.



- The four large boxes (“inputs,” “converters,” “streams,” and “outputs”) divide the signal path into manageable territories; each territory is examined in separate sections, below.
- The smaller boxes (“MIC,” “CD,” and so on) are actual or virtual sound devices.
- The long arrowed lines show how the devices are connected. A single line indicates a single channel, a double line means stereo. The arrowhead at the end of each line indicates the direction of the signal.
- The circled arrows show where the software can exhibit gain control over a device. Each control point is labelled as it’s known to the Media Kit. A shaded circle means

the control point has a volume control *and* a mute. An unshaded circle signifies a mute but no volume control.

Inputs

There are three analog audio input devices:

- *The microphone*. The microphone jack at the back of the computer accepts a stereo mini-phone (1/8") plug. The analog microphone signal has its own volume control and mute, and also allows a 20 dB boost. The microphone signal then feeds into the input MUX.
- *Line-in*. The stereo line-in jacks at the back of the computer bring a line-level analog signal into the computer. This signal can be routed directly to the audio output devices, and fed to the MUX. The direct-to-output, or "through," path has its own volume control and mute; this control point is called **B_LINE_IN_THROUGH** by the Kit.
- *CD input*. The CD (analog) input has the same features as line-in: The CD signal can be sent through to the output (**B_CD_THROUGH**), and it can be fed to the MUX.

Note that the microphone signal *doesn't* have a through path.

To bring an analog signal into your application (so you can record it, for example), the signal must pass through the input MUX:

- The MUX is a "mutually exclusive" device that lets you choose a single (analog) input from among the three sources listed above. In other words, you can bring in the microphone signal *or* the line-in signal *or* the CD signal, but you can't bring in any two or all of them at the same time. The MUX passes the input signal to its output without conversion to digital representation or other modification.

Converters

There are two sound data converters, the analog-to-digital converter (ADC) and the digital-to-analog converter (DAC):

- The ADC takes the analog signal that it reads from the MUX and converts it to digital representation. It does this by producing a series of *samples*, or instantaneous measurements of the signal's amplitude. The ADC control point is called **B_ADC_IN.**
- The DAC converts digital sound data into a continuous analog signal. The DAC control point is called **B_DAC_OUT**.

Acting as a sort of "short-circuit" between these two devices is the loopback:

- The loopback path takes the digital signal straight out of the ADC and sends it to the DAC. This path (which can be muted, but doesn't have a volume control) is intended, primarily, to simulate a "through" path for the microphone signal. There's little reason to send the line-in or CD signal down the loopback path since they have actual through paths built in.

Streams

The ADC stream and DAC stream are the centerpieces of the BAudioSubscriber class. By subscribing to the ADC stream you can receive the samples that are emitted by the ADC; and by subscribing to the DAC stream, you can send buffers of digital sound data to the DAC.

To enter the ADC stream you must create a BAudioSubscriber, subscribe to the stream (by passing **B_ADC_STREAM** as the first argument to **Subscribe()**), and then call **EnterStream()**. At that point, your object will begin receiving buffers of ADC-converted data from the Audio Server. The buffers show up as arguments to the object's stream function.

Similarly, the DAC stream universe is broached by subscribing to and entering the **B_DAC_STREAM**.

If you're unfamiliar with the concepts of subscription, entering a stream, and the stream function, take a break and read the BSubscriber specification.

Outputs

The output devices take analog signals and broadcast them to hardware that can turn the signals into sound.

- The output mixer mixes the signal from the DAC with the signals from the line-in and CD through paths. You can control the output of this mix at the **B_MASTER_OUT** control point.
- The mixed signal is presented at the stereo line-out jacks at the back of the computer. This is the same signal that's presented at the headphone jack.
- The stereo signal is mixed to mono (and attenuated by 6 dB) and sent to the abysmal internal speaker. The speaker has its own volume and mute control (**B_SPEAKER_OUT**).

Controlling the Hardware

The BAudioSubscriber class defines a number of functions that control the sound hardware and that query the state of the hardware. Note that you can call these functions without first subscribing to one or the other of the audio streams.

Volume and Mute

To set the volume level of a particular sound device, you use BAudioSubscriber's **SetVolume()** function. The function takes three arguments:

- A constant that represents the device you want to control.
- A float that sets the volume level of the left channel of the device.
- A float that does the same for the right channel.

The device constants are listed below; they correspond to the named control points shown in the hardware diagram:

- **B_CD_THROUGH**
- **B_LINE_IN_THROUGH**
- **B_ADC_IN**
- **B_LOOPBACK**
- **B_DAC_OUT**
- **B_MASTER_OUT**
- **B_SPEAKER_OUT**

All volume levels are floating-point numbers in the range [0.0, 1.0], where 0.0 is inaudible, and 1.0 is maximum volume. If you're setting a single-channel device (the speaker), the left channel level is used—the value you pass as the right channel level is ignored. If you want to set one channel of a stereo device but leave the other at its present level, pass the **B_NO_CHANGE** constant for the no-change channel.

In the example below, a BAudioSubscriber is used to set the volume of the CD-through signal:

```
BAudioSubscriber *setter = BAudioSubscriber("setter");

/* Set the right channel of the CD through signal
 * to half the maximum volume, and leave the left channel
 * alone.
 */
setter->SetVolume(B_CD_THROUGH, B_NO_CHANGE, 0.5);
```

To mute a device, you disable it; or, more precisely, you set it to be not enabled. This is done through the **EnableDevice()** function. As with **SetVolume()**, the function's first argument is the constant that represents the device you want to control. The second argument is a boolean that states whether you want to enable (**TRUE**) or disable (**FALSE**) the device. For example:

```
/* Mute the internal speaker. */
setter->EnableDevice(B_SPEAKER_OUT, FALSE);
```

The **GetVolume()** and **IsDeviceEnabled()** functions retrieve the current volume and enabled state of a given device. (As a convenience, **GetVolume()** returns volume *and* enabled status; see the function description for details.)

The MUX and the Mic

To select the analog device that will feed into the MUX, you use the **SetADCInput()** function (the signal into the MUX goes to the ADC, hence the name of the function). The input devices are represented by these constants:

- **B_MIC_IN**
- **B_CD_IN**
- **B_LINE_IN**

The **ADCInput()** function returns the current input device.

The microphone's 20 dB boost is toggled through the **BoostMic()** function. The state of the boost is retrieved by **IsMicBoosted()**.

Sound Data

Sounds are propagated by the continuous fluctuation of air pressure. This fluctuation is called a sound wave. The digital representation of a sound wave consists of a series of discrete measurements of the instantaneous pressure (or amplitude) of the wave at precise, (typically) equally-spaced points in time. Each measurement is called a *sample*. There are five attributes that characterize a digital sound sample:

- The size of a single sound sample (the Media Kit expresses this measurement in bytes-per-sample).
- The order of bytes in a multiple-byte sample.
- The number of samples in a "frame" of sound, where each sample in the frame is meant to be played at the same time. For example, a stereo sound would have two samples-per-frame. Samples-per-frame is commonly called the *channel count*.
- The number of frames that should be played in a second. This is commonly called the *sampling rate*.
- The mapping from the value of a digital sample to a specific sound wave amplitude. The Media Kit calls this the *sample format*. Usually, the mapping is linear: When you double the value of a sample, you double the amplitude to which it corresponds.

The Be sound hardware (both the ADC and the DAC) allows the following sound attribute settings:

- Sample size can be one or two bytes-per-sample.
- Byte-ordering is either most-significant-byte first (**B_BIG_ENDIAN**), or least-significant-byte first (**B_LITTLE_ENDIAN**).
- The channel count can be one (mono) or two (stereo).

- The sampling rates, expressed as frames-per-second, that are supported by the hardware are: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.

- There are two sample formats: The linear format, represented by the constant **B_LINEAR_SAMPLES**, can be used with either one- or two-byte samples. The "mu-law" format (**B_MULAW_SAMPLES**) can only be used with one-byte samples. Mu-law is a quasi-exponential mapping that attempts to minimize quantization noise by dedicating more bits, proportionally, to low amplitude values than to high amplitude values.

The ADC and DAC use the same sampling rate. You can set the sampling rate through BAudioSubscriber's **SetSamplingRate()** function, but you can't specify which device you intend the setting to apply to: It always applies to both.

As for the other sound data parameters (sample size, byte order, channel count, and sample format), the ADC and the DAC maintain independent settings. For example, you can set the DAC to expect two-byte linear samples while the ADC produces one-byte mu-law samples. The functions that set these sound format attributes are **SetDACSampleInfo()** and **SetADCSampleInfo()**. Your BAudioSubscriber needn't subscribe before setting the DAC or ADC parameters.

Receiving and Broadcasting Sound Data

A BAudioSubscriber object receives buffers of sound data from one of the Audio Server's two buffer streams:

- The buffers that flow through the ADC stream are filled with sound data that's been brought into the computer, passed through the MUX, and converted by the ADC. Data buffers that are received by your objects will already be filled with this data. Although it's not forbidden, you usually don't modify the data in the ADC stream's buffers. BAudioSubscribers that enter the ADC stream do so, typically, to record or examine the data that they find there.

- The buffers that flow through the DAC stream are ultimately dumped into the DAC. The DAC stream's buffers are zeroed at the start of their journey; if a BAudioSubscriber wants to broadcast a sound, it enters the DAC stream and adds its sound data into the buffers as they flow past.

The ADC stream isn't automatically connected to the DAC stream. If you want to grab data from the ADC and send it to the DAC, you have to subscribe to both streams through two separate BAudioSubscriber objects, and then coordinate the hand off of data from the ADC subscriber to the DAC subscriber.

Constructor and Destructor

BAudioSubscriber()

BAudioSubscriber(const char **name***name***)**

Creates and returns a new BAudioSubscriber object. The object is given the name that you pass as *name*; the name is provided as a convenience and needn't be unique.

After creating a BAudioSubscriber, you typically do the following (in this order):

- Subscribe the object to one of the Audio Server's streams (either **B_ADC_STREAM** or **B_DAC_STREAM**) by calling **Subscribe()**.
- Allow the object to begin receiving buffers by calling **EnterStream()**.

See also: **BSubscriber::Subscribe()**, **BSubscriber::EnterStream()**

~BAudioSubscriber()

virtual **~BAudioSubscriber(**void**)**

Destroys the BAudioSubscriber.

Member Functions

ADCInput(), SetADCInput

long **ADCInput(**void**)**
long **SetADCInput(**long *device***)**

These functions get and set the device that feeds into the MUX (and so to the ADC, hence the name). Valid *device* constants are:

- **B_MIC_IN**
- **B_CD_IN**
- **B_LINE_IN**

You don't need to be subscribed to the ADC stream in order to call these functions.

BoostMic(), IsMicBoosted()

long **BoostMic(**bool *boost***)**
bool **IsMicBoosted(**void**)**

BoostMic() enables or disables the 20 dB boost on the microphone signal. **IsMicBoosted()** returns the state of the boost.

GetADCSampleInfo(), GetDACSampleInfo(), SamplingRate()

long **GetADCSampleInfo(**long **bytesPerSample*,
long **channelCount*,
long **byteOrder*,
long **sampleFormat***)**

long **GetDACSampleInfo(**long **bytesPerSample*,
long **channelCount*,
long **byteOrder*,
long **sampleFormat***)**

long **SamplingRate(**void**)**

These functions return the values of the various sound data parameters. **GetADC...** returns (by reference) the sound parameters that are used in the ADC stream. **GetDAC...** does the same for the DAC stream. **SamplingRate()** returns the sampling rate directly; the sampling rate is held in common by the two streams.

See the description of **SetADCSampleInfo()** for a list of parameter values that you can expect to see.

See also: **SetADCSampleInfo()**

GetDACSampleInfo() *see* GetADCSampleInfo()

SetADCSampleInfo(), SetDACSampleInfo(), SetSamplingRate()

long **SetADCSampleInfo(**long *bytesPerSample*,
long *channelCount*,
long *byteOrder*,
long *sampleFormat***)**

long **SetDACSampleInfo(**long *bytesPerSample*,
long *channelCount*,
long *byteOrder*,
long *sampleFormat***)**

long **SetSamplingRate(**long *samplingRate***)**

These functions set the values of the sound data attributes used by (respectively) the ADC stream (**SetADC...**), DAC stream (**SetDAC...**), and both streams (**SetSamplingRate()**). The arguments to the **SetADC...** and **SetDAC...** functions are:

- *bytesPerSample* is the size of a single sound sample measured in bytes. Acceptable values are 1 and 2.
- *channelCount* is the number of samples in a “frame” of sound. Acceptable values are 1(mono) and 2 (stereo).

- *byteOrder* is the order of bytes in a multiple-byte sample. The ordering is either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**.
- *sampleFormat* is the data format of a single sample. Linear format (**B_LINEAR_SAMPLES**) can be used for one- or two-byte samples; mu-law format (**B_MULAW_SAMPLES**) can be used for 1-byte samples.

The **SetSamplingRate()** function sets the sampling rate for both the ADC stream and the DAC stream:

- The following sampling rates are supported by the sound hardware: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.

These functions don't flinch at wildly inappropriate parameter settings. The values of the arguments that you pass in are always rounded to the nearest acceptable value for the particular parameter.

See also: **GetADCSampleInfo()**

SetDACSampleInfo() *see* SetADCSampleInfo()

SetVolume(), GetVolume(), EnableDevice(), IsDeviceEnabled()

long **SetVolume(**long *device*,
float *leftVolume*,
float *rightVolume***)**
long **GetVolume(**long *device*,
float **leftVolume*,
float **rightVolume***,*
bool *isEnabled***)**

long **EnableDevice(**long *device*, bool *enable***)**
bool **IsDeviceEnabled(**long *device***)**

These functions set and return (by reference) the left and right volume levels, and the enabled status of the device that's identified by the first argument. Valid device constants are:

- **B_ADC_IN**
- **B_CD_THROUGH**
- **B_LINE_IN_THROUGH**
- **B_LOOPBACK**
- **B_DAC_OUT**
- **B_MASTER_OUT**
- **B_SPEAKER_OUT**

Volume values are floating-point numbers that are clipped within the range [0.0, 1.0]. Across this range, the amplitude of a sound waveform is increased logarithmically; this results, perceptually, in a linear increase in volume.

Note that you can't set the volume of the **B_LOOPBACK** device (it doesn't have a volume control). Also, the speaker is monophonic; when you set or retrieve the volume of the **B_SPEAKER_OUT** device, only the *leftVolume* argument is used.

You needn't be subscribed to call these functions.

BSoundFile

Derived from: public BFile

Declared in: <media/SoundFile.h>

Overview

BSoundFile objects give you access to files that contain sound data. The BSoundFile functions let you examine the format of the data in the sound file, read the data, and position a "frame pointer" in the file. Notably absent from the list of a BSoundFile's talents is the ability to play itself and to record into itself. You *can* play a BSoundFile's data, but this requires the assistance of a BAudioSubscriber (as explained later). Currently, you can't record into a BSoundFile.

To use a BSoundFile, you set its ref (using the methods that are described in the BFile class), and then you open the file through a call to **Open()**. Since you can't record into a BSoundFile, you almost always open such files in **B_READ_ONLY** mode. None of the BSoundFile-defined functions work on an unopened file.

Sound File Formats

The BSoundFile class understands AIFF, WAVE, and "standard" UNIX sound files. When you tell a BSoundFile object to open its file, the object figures out the format of the file—you can't force it to assume a particular format. If it encounters a file that's in a format that it doesn't understand ("unknown" format), it assumes that the file contains 44.1 kHz, 16-bit stereo data, and that the file doesn't have a header (it assumes that the entire file is sound data). The admission of the unknown format means that *any* file can act as sound data. BSoundFile doesn't know the meaning of "inappropriate data."

The file formats are represented by the constants **B_AIFF_FILE**, **B_WAVE_FILE**, **B_UNIX_FILE**, and **B_UNKNOWN_FILE**. You can retrieve the file format from an open BSoundFile through the **FileFormat()** function.

Note: 8-bit WAVE data is, by definition, unsigned. However, when you read such data (through the **ReadFrames()** function, which will be discussed later), it's automatically shifted so that it *is* signed. This automatic conversion allows an 8-bit WAVE file to be mixed with other sound sources.

Sound Data Parameters

After opening your BSoundFile, you can ask for the parameters of its data by calling the various parameter-retrieving functions (**SamplingRate()**, **ChannelCount()**, **SampleSize()**, and so on). There's also a set of parameter-setting functions (**SetSamplingRate()**, **SetChannelCount()**, **SetSampleSize()**, ...), but note that these functions don't actually modify the data in the file (or in the BSoundFile object); they simply set the object's impression of the sort of data that it contains so other objects that act on your BSoundFile will interpret the data correctly. This should only be necessary if the file format is unknown.

For example, let's say you have your own sound file format. Your format defines a header that lists the usual information—the size of the samples, where the data starts, and so on. BSoundFile won't recognize your format, of course, but through a combination of the **Read()** function (so you can read the header yourself) and the sample parameter-setting functions defined by BSoundFile, you can tell the object what sort of data it contains.

If you're creating your own BSoundFile-derived class to encapsulate your own sound file format, you would put the header-reading code in your implementation of the **Open()** function. For example:

```
long MySoundFile::Open(long mode)
{
    long result;

    if ((result = BSoundFile::Open(mode)) < B_NO_ERROR)
        return result;

    /* ReadHeader() is assumed to be implemented
     * by MySoundFile--it isn't a BSoundFile function.
     */
    if (FileFormat() == B_UNKNOWN_FILE)
        result = ReadHeader());
    return result;
}
```

By invoking the BSoundFile version of **Open()**, you allow your object to represent the standard file formats in addition to your own. (Keep in mind that BSoundFile sets the file format in its **Open()** implementation.)

Playing a Sound File

There are two methods for playing a sound file:

- The easy way is to call the **play_sound()** function. The function takes a **record_ref** argument (in addition to others), and plays the data that it finds in the referred to file—you don't need to create a BSoundFile object in order to call **play_sound()**. (The complete documentation for **play_sound()** and related functions can be found in the final section of this chapter.)

- A much more amusing approach is to create a BSoundFile object, open it, read the data contained within, and add the data into the DAC stream. Obviously, this is a bit more involved than the simple **play_sound()** (which a dead dog would have no trouble using), but it gives you more control over the sound: Since you're reading the sound data yourself, the BSoundFile approach lets you manipulate the sound as you're throwing it into the stream.

A demonstration of the second approach is given below. To understand the example, you must be familiar with the subscription and stream-entering mechanisms described in the BSubscriber class.

An Example

In this example, we show how to read data from a BSoundFile and add it to the DAC stream for playback. In addition, we'll allow dynamic amplitude control of the sound. For the sake of brevity, we'll restrict the example to 16-bit data.

First, we define an object called SoundPlayer that will be used to coordinate the Media Kit objects. Notice that SoundPlayer needn't derive from a Kit class:

```
class SoundPlayer : public BObject
{
    public:
        long SetSoundFile(record_ref ref);
        void Play(void);
        void SetAmpScale(double value);

    private:
        static bool _play_back(void *arg, char *sound,
            long size);
        bool Playback(short *sound, long sample_count);

        BAudioSubscriber *a_sub;
        BSoundFile s_file;
        char transfer_buf[B_PAGE_SIZE];
        double amp_scale;
};
```

There are three public functions: **SetSoundFile()** let's you set the soundfile that you want to play, **Play()** plays it, and **SetAmpScale()** will control the amplitude. In this implementation, the file is always allowed to play to completion—aborting the playback is left as an exercise for the reader.

The private **_play_back()** function will be the BAudioSubscriber's literal stream function. **Playback()** will be called from within **_play_back()**; it will do the actual stream work. The private **transfer_buf** will be used to transfer data between the file and the audio stream (a page at a time), and **amp_scale** will hold the amplitude scaling value.

Opening the File and Subscribing

In the implementation of **SetSoundFile()**, we set the BSoundFile's ref and open the file...

```
long SoundPlayer::SetSoundFile(record_ref ref)
{
    /* Set the BSoundFile's ref and open the object. */
    s_file.SetRef(ref);
    s_file.Open(B_READ_ONLY);
    if (s_file.Error() < B_NO_ERROR)
        return B_ERROR;

    /* Check for 16-bit data (given in bytes). */
    if (s_file.SampleSize() != 2)
        return B_ERROR;
```

...and then we create the BAudioSubscriber and subscribe it to the DAC stream and set the stream's sample parameters to match the data that's in the file:

```
    a_sub = new BAudioSubscriber("SoundFile Player");
    if(!a_sub->Subscribe(B_DAC_STREAM, B_SHARED_SUBSCRIBER_ID,
        FALSE) < B_NO_ERROR)
        return B_ERROR;

    a_sub->SetSamplingRate(s_file.SamplingRate());
    a_sub->SetDACSampleInfo(s_file.SampleSize(),
                            s_file.CountChannels(),
                            s_file.ByteOrder(),
                            s_file.SampleFormat());
```

Next, we set the size of the stream's buffers to match that of our transfer buffer. The arguments to **SetStreamBuffers()** are buffer size, buffer count. The buffer count we use here (8, the same as the Audio Server default) is unimportant in this example:

```
    a_sub->SetStreamBuffers(B_PAGE_SIZE, 8);
```

Finally, we initialize the amp scaler and return:

```
    amp_scale = 1.0;
    return B_NO_ERROR;
}
```

By setting the DAC stream's sample info and buffer size as shown in here, we make the stream function's job quite a bit easier—it won't have to convert the samples as it reads them from the file, or keep track of how many samples it has read. However, you should be aware that some other BAudioSubscriber could come along and reset the DAC stream at any time, thus screwing up the playback. For now, we'll live with the danger.

Entering the Stream

The **Play()** function enters the BAudioSubscriber into the DAC stream. This causes buffers to be sent to the stream function, which we'll implement in the next section.

```
void SoundPlayer::Play(void)
{
    a_sub->EnterStream(NULL, /* no neighbor */
                       TRUE, /* head of the stream */
                       this, /* arg for the stream function */
                       _play_back, /* the stream function */
                       NULL, /* no completion function */
                       TRUE); /* run in the background */
}
```

While we're at it, we'll implement the **SetAmpScale()** function:

```
void SoundPlayer::SetAmpScale(double scale)
{
    amp_scale = min(1.0, max(0.0, scale));
}
```

Reading and Playing the File

Now comes the fun part. First we implement the literal stream function, **_play_back()**:

```
bool SoundPlayer::_play_back(void *arg, char *sound, long size)
{
    return (((SoundPlayer *)arg)->Playback((short *)sound,
                                           size/2));
}
```

As **_play_back()** receives buffers from the DAC stream, it forwards them (cast as 16-bit data) to the guts of the operation, **Playback()**. At each invocation, **Playback()** reads the correct number of frames from the sound file, scales their amplitudes, and adds the samples into the DAC stream buffer. First, we set up some variables:

```
bool SoundPlayer::Playback(short *sound, long sample_count)
{
    long frames_read, counter;
    long channel_count = s_file.CountChannels();
    long frame_count = sample_count / channel_count;
    short *tb_ptr = (short *)transfer_buf;
```

Now we read **frame_count** sample frames from the file and place them in the transfer buffer. (We should check to make sure that the transfer buffer can accommodate the number of frames read—but, for this example, we'll assume that the stream's buffer size hasn't changed since we set it to be the same size as the transfer buffer.) If **ReadFrames()** returns less than the number of frames that we asked for, we're at the end of the file. **ReadFrames()** returns the number of frames that it actually read, or an error code (as usual, a negative number) if something went wrong:

```
    frames_read = s_file.ReadFrames(transfer_buf, frame_count);

    if (frames_read <= 0)
        return FALSE;
```

Finally, we get to write into the sound buffer. We loop over the samples in the transfer buffer, scale each by the **amp_scale** value, and then write the scaled value into the sound buffer:

```
        for (counter = 0; counter < frames_read; counter++) {
            *sound++ += *tb_ptr++ * amp_scale; /* left or mono */
            if (channel_count == 2)
                *sound++ += *tb_ptr++ * amp_scale; /* right */
        }
```

Once again we examine the **frames_read** count. If it's less than what we expected to have read, we've reached the end of the file, and so return **FALSE**. Otherwise we return **TRUE**:

```
        if (frames_read < frame_count)
            return FALSE;
        else
            return TRUE;
    }
```

Obviously, this example is neither robust nor efficient. In particular, the file-reading mechanism should probably read more than one page at a time—if you were to play more than a couple files simultaneously with this code, the constant file seeking could cause your hard disk to burn a hole right through to Australia. Or to California, if you live in Perth. The point of this exercise was to demonstrate the basic procedures of playing a sound file.

Constructor and Destructor

BSoundFile()

BSoundFile(void**)**
BSoundFile(record_ref *ref***)**

Creates and returns a new BSoundFile object. The first version of the constructor must be followed by a call to **SetRef()**.

~BSoundFile()

virtual **~BSoundFile(**void**)**

Closes the BSoundFile's sound file and destroys the object. The data in the sound file isn't affected.

Member Functions

CountFrames()

long **CountFrames(**void**)**

Returns the number of frames of sound that are in the object's file. If the object's file isn't open, this returns **B_ERROR**.

FileFormat()

long **FileFormat(**void**)**

Returns a constant that identifies the type of sound file that this object is associated with. Currently, three types of sound files are recognized: **B_AIFF_FILE, B_WAVE_FILE, B_UNIX_FILE** and **B_UNKNOWN_FILE**. AIFF is the Apple-defined sound format, WAVE is a popular PC format, the **B_UNIX_FILE** constant represents the sound file format that's used on many UNIX-based computers. **B_UNKNOWN_FILE** is returned for all other formats.

B_UNKNOWN_FILE isn't as useless as it sounds: Any file that is so identified is considered to contain "raw" sound data. You can accept the default values of the data format parameters (see **SamplingRate()** for a list of these values), or you can shape the data into a recognizable format by setting the data format parameters directly, through calls to **SetSamplingRate()**, **SetChannelCount()**, and so on. In this case, you'll need to position the frame pointer to the first frame—in other words, you have to read past the file's header, if any—yourself. Thus primed, subsequent calls to ReadFrames() will read the proper sequences of samples.

If the BSoundFile isn't open, this returns **B_ERROR**.

FrameIndex() *see* SeekToFrame()

FramesRemaining()

long **FramesRemaining(**void**)**

Returns the number of unread frames in the file, or **B_ERROR** if the object isn't open.

ReadFrames()

virtual long **ReadFrames(**char ***buffer***,** long *frameCount***)**

Reads (as many as) *frameCount* frames of data into *buffer.* The function returns the number of frames that were actually read and increments the frame pointer by that amount. When you hit the end of the file, the function returns 0.

Note that *buffer* shouldn't be the sound buffer that's passed to you in a stream function. If you read directly into a stream function's sound buffer, you'll be clobbering the data that's already there. If you're calling **ReadFrames()** from within a stream function, you must first read into a "transfer buffer", and then add the contents of this buffer into the sound buffer.

If the BSoundFile object isn't open, this returns **B_ERROR**.

SamplingRate(), CountChannels(), SampleSize(), FrameSize(), ByteOrder(), SampleFormat()

long **SamplingRate(**void**)**
long **CountChannels(**void**)**
long **SampleSize(**void**)**
long **FrameSize(**void**)**
long **ByteOrder(**void**)**
long **SampleFormat(**void**)**

These functions return information about the format of the data that's found in the object's sound file:

- **SamplingRate()** returns the sampling rate.
- **CountChannels()** returns the number of channels of sound.
- **SampleSize()** returns the size, in bytes, of a single sample.
- **FrameSize()** is a convenience function that give the number of bytes in a single frame of sound (it's the same as **CountChannels()** * **SampleSize()**).
- **ByteOrder()** returns a constant that represents the order of samples within a frame. It's either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**.
- **SampleFormat()** returns a constant that represents the data format of a single sample. It's one of: **B_LINEAR_SAMPLES**, **B_MULAW_SAMPLES**, **B_FLOAT_SAMPLES**, or **B_UNDEFINED_SAMPLES**.

These functions returns default values if the object isn't associated with a file. The defaults are:

- 44100 frames per second
- 2 channels
- 2 bytes per sample (16-bit samples)
- 4 bytes per frame
- Bytes are ordered MSB first (**B_BIG_ENDIAN**)
- The sample format is **B_LINEAR_SAMPLES**

If the BSoundFile object isn't open, these functions return **B_ERROR**.

SeekToFrame(), FrameIndex()

virtual long **SeekToFrame(**ulong *index***)**
long **FrameIndex(**void**)**

Theses function set and return the location of the "frame pointer." The frame pointer points to the next frame that will be read from the file. The first frame in a file is frame 0.

If you try to set the frame pointer to a location that's outside the bounds of the data, the pointer is set to the frame at the nearest extreme.

If the BSoundFile object isn't open, this returns **B_ERROR**.

SetSamplingRate(),SetChannelCount(), SetSampleSize() SetByteOrder(), SetSampleFormat()

virtual long **SetSamplingRate(**long *samplingRate***)**
virtual long **SetChannelCount(**long *channelCount***)**
virtual long **SetSampleSize(**long *bytesPerSample***)**
virtual long **SetByteOrder(**long *byteOrder***)**
virtual long **SetSampleFormat(**long *sampleFormat***)**

If the file format of your BSoundFile is **B_UNKNOWN_FILE**, you can use these functions to tell the object how to interpret the format of its data. These functions don't change the actual data—neither as it's represented within the object, nor as it resides in the file—they simply prime the object for subsequent reads of the data.

The candidate values for the functions are:

- *samplingRate* can be any number, but will be rounded to the nearest hardware-supported sampling rate when the data is played. The sampling rates that the hardware supports are: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.
- *channelCount* is usually 1 (mono) or 2 (stereo). You can set the data to a higher count but the hardware can play no more than 2 channels at a time.
- *sampleSize* is usually 2 (16 bit samples). But it can also be 1 (the usual setting for mu-law encoding) or 4 (floating-point data).
- *byteOrder* is either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**
- *sampleFormat* is one of **B_LINEAR_SAMPLES**, **B_MULAW_SAMPLES**, **B_FLOAT_SAMPLES**, or **B_UNDEFINED_SAMPLES.**

Each function returns the value that was actually implanted. If the BSoundFile object isn't open, they return **B_ERROR**.

BSubscriber

Derived from: public BObject

Declared in: <media/Subscriber.h>

Overview

BSubscriber objects receive and process buffers of media-specific data. These buffers are allocated and sent (to the BSubscriber) by a media server; for example, buffers of audio data are sent by the *Audio Server.* Each server can control more than one *buffer stream* (the Audio Server has a sound-in stream and a sound-out stream). A BSubscriber can receive buffers from only one stream.

More than one BSubscriber can "subscribe" to the same stream. The collection of same-stream BSubscribers stand shoulder-to-shoulder and pass buffers down the stream, in the style of a bucket brigade. When a BSubscriber receives a buffer it does something to it—typically, it examines, adds to, or filters the data it finds there—and then passes it to the next BSubscriber (or, more accurately, lets the server pass it to the next BSubscriber).

The media servers take care of managing the data buffers in their streams—they allocate new buffers, pass them between BSubscribers, clear existing buffers for re-use, and so on. A BSubscriber's primary tasks are these (and in this order):

- Identifying the media server that it wants to get buffers from.
- Applying for aceptance into one of the server's streams (this is called "subscribing").
- Entering the stream. At the moment a BSubscriber enters a stream, the object begins receiving data buffers from the server.
- Processing the data that it finds in the buffers that it receives.

The BSubscribers that subscribe to the same stream needn't belong to the same application. This means that your BSubscriber may be examining, adding to, or filtering data that was generated in another application.

Most buffer streams need to "flow" quickly and uninterruptedly (this is especially true of the Audio Server's streams). The processing that a single BSubscriber performs when it receives a buffer from the server should be as brief and efficient as possible.

Identifying a Server

BSubscriber is an abstract class—you never construct instances of BSubscriber directly. Instead, you construct instances of one of its derived classes. Each BSubscriber-derived class provided by the Media Kit corresponds to a particular media server. Identifying a server, therefore, is implied by the act of choosing a BSubscriber-derived class with which you instantiate an object.

Currently, the only BSubscriber-derived class that's supplied by the Media Kit is BAudioSubscriber. Instances of this class receive buffers from, obviously enough, the Audio Server.

Subscribing

The first thing you do with your BSubscriber object, once you've constructed it, is to ask its server's permission to be sent buffers of data. This is performed through the **Subscribe()** function. Subscription doesn't cause buffers to actually be sent, but it does get the BSubscriber into the ballpark. The act by which a BSubscriber receives buffers (the **EnterStream()** function) depends on a successful subscription.

As part of a BSubscriber's subscription, it must tell the server which stream it wants to enter, which other BSubscribers it's willing to share the buffer stream with, and whether it's willing to wait for "undesirable" brethren to get out of the stream before it gets in. The object's opinions on these topics are registered through arguments to the **Subscribe()** function:

long **Subscribe(**long *stream*, subscriber_id *clique*, bool *willWait***)**

The arguments are discussed in the following sections.

The Stream

A server can shepherd more than one stream. For example, the Audio Server controls access to two streams: The sound-out stream terminates at the DAC, the sound-in stream begins at the ADC. You identify the stream you want by using one of the stream constants defined by the server. The Audio Server defines the constants **B_DAC_STREAM** for sound-out and **B_ADC_STREAM** for sound-in.

A BSubscriber may only subscribe to one stream at a time.

The Clique

A BSubscriber's clique (passed as the *clique* argument to **Subscribe()**) identifies the cabal of BSubscribers that the calling object is willing to share the server's buffer stream with. The value of *clique* acts as a "key" to the stream: To gain access to the stream, you have to have the proper key.

Here's how it works: The first BSubscriber that calls **Subscribe()** passes some value as the *clique* argument. This value becomes the key to the buffer stream; any other BSubscriber that wants to subscribe to that stream must pass the same clique value (unless you want to be an "invisible" subscriber, as described in the next section). The actual value that's used to represent the clique is irrelevant; matching is the only concern. A given clique is enforced until all subscribed objects have *unsubscribed* (through the **Unsubscribe()** function) at which point the next object that subscribes will establish a new clique value.

Note: The *clique* argument is type cast as a **subscriber_id**. Such values are tokens that uniquely identify BSubscriber objects among all extant BSubscribers of the same class (across all applications). That the clique is represented as a **subscriber_id** is primarily a convenience: Just as the actual clique value has no significance, neither does its type imply any special properties about the clique.

Choosing a Clique Value

With regard to cliques, there are four types of BSubscribers: Those that want utterly exclusive access to the buffer stream, those that are willing to share access with certain (but not all) other BSubscribers, those that will share with any other BSubscriber, and those that want to crash the party. The clique value that you choose depends on which of these characterizations describes your BSubscriber:

- *Exclusive access.* If a BSubscriber wants to have exclusive access to the stream—if it doesn't want any other BSubscriber to be able to enter the stream while it's subscribed—then the object passes some value as the *clique* argument, but keeps the value a secret. Typically, the object's own **subscriber_id** value is used as the argument; the **ID()** function supplies this value:

```
/* FirstSubscriber is assumed to be a valid BSubscriber
 * object (currently, it must be an instance of
 * BAudioSubscriber).
 */
subscriber_id firstID = FirstSubscriber->ID();
FirstSubscriber->Subscribe(firstID, ...);
```

- *Selective sharing.* If the first subscriber wants to share the stream with subsequent subscribers, the initial clique value must be used in those subsequent subscriptions:

```
/* First... */
subscriber_id firstID = FirstSubscriber->ID();
FirstSubscriber->Subscribe(firstID, ...);
...

/* Notice that the second subscriber passes the
 * first subscriber's ID value as the clique argument.
 */
SecondSubscriber->Subscribe(firstID, ...);
```

To share the stream with BSubscribers in other applications, the first subscriber's application would have to broadcast the first subscriber's ID value (through a BMessage, for example).

- *Indiscriminate sharing.* To share the stream between all BSubscribers in all applications is easy: You pass the **B_SHARED_SUBSCRIBER_ID** constant as the value for *clique*:

  ```
  FirstSubscriber->Subscribe(B_SHARED_SUBSCRIBER_ID, ...);
  ```

  Note, however, that the **B_SHARED_SUBSCRIBER_ID** clique doesn't *guarantee* that every BSubscriber will be allowed in the stream. If an unsharing BSubscriber has already set the clique to some other value, a BSubscriber that passes **B_SHARED_SUBSCRIBER_ID** will be turned down. Conversely, if the clique is set to **B_SHARED_SUBSCRIBER_ID** and a BSubscriber comes along that tries to subscribe with a less generous clique value, it's subscription will be denied.

- *Gate crashing.* If you just don't care who's in the stream or whether they like you or not, use the constant **B_INVISIBLE_SUBSCRIBER_ID** as the *clique* value. This will get you in regardless of—and without changing—the current clique setting. If you're the first subscriber, the next subscriber will be allowed in regardless of his clique specification (and the stream's clique will be set to this subsequent value).

Waiting for Access

If a BSubscriber is denied access to a server because it didn't pass the clique test, it can either give up immediately, or wait for the current clique members to unsubscribe. This is expressed in **Subscribe()**'s final argument, the boolean *willWait*:

- If *willWait* is **FALSE**, **Subscribe()** returns immediately, regardless of its success in gaining access to the server. (The measure of its success is given by the function's return value.)

- If it's **TRUE**, the function doesn't return until the BSubscriber has successfully subscribed. There's no time-out provision, so the wait is indefinite. (Yes, there is a **SetTimeout()** function; no, it doesn't apply to subscription.)

Entering the Stream

Having successfully subscribed to a server's stream, the BSubscriber's next task is to enter the stream. By this, the object will begin receiving buffers of data. You do this through the **EnterStream()** function:

virtual long **EnterStream(**subscriber_id *neighbor*,
bool *before*,
void **arg*,
StreamFn *streamFunction*,
CompletionFn *completionFunction*,
bool *background***)**

The function's operations and arguments are described in the following sections.

Positioning your BSubscriber

The first two **EnterStream()** arguments position the BSubscriber with respect to the other BSubscriber objects that are already in the stream (if any):

EnterStream(subscriber_id *neighbor*, bool *before*, ...**)**

The *neighbor* argument identifies the BSubscriber (by its ID number, as returned by the **ID()** function) that you want the entering BSubscriber to stand next to; *before* places the entering object before (**TRUE**) or after (**FALSE**) the neighbor. The neighbor needn't belong to the same application as the entering object, but it must already have entered the stream.

If you want to place the BSubscriber at one or the other end of the stream (or to add the first BSubscriber to the stream), you pass **NULL** as the neighbor. A before value of **TRUE** thus places the BSubscriber at the "front" of the stream (the object will be the first to receive each buffer that flows through the stream), and a value of **FALSE** places it at the "back" (it's the last to receive buffers before they're realized or recycled).

A BSubscriber's position in the stream can't be locked. If, for example, you place your BSubscriber to stand at the back of the stream, some other BSubscriber—from some other application, possibly—can come along later and also claim the back. Your object will be bumped forward (towards the front of the stream) in deference to the newcomer.

Receiving and Processing Buffers

After your BSubscriber has entered the buffer stream, it will begin receiving buffers of data. The third, fourth, and last arguments to **EnterStream()** pertain to the means by which your object receives these buffers:

EnterStream(..., void **arg*, StreamFn *streamFunction*, ..., bool *background***)**

The arguments, taken out of order, are:

- *streamFunction* is a pointer to a boolean function (the complete protocol is given below) that will be invoked once for each buffer that's received.
- *arg* is a pointer-sized value that will be passed as an argument to *streamFunction*.

- The value of *background* is used to determine whether *streamFunction* will be executed in a separate thread (**TRUE**) or in the same thread (**FALSE**) as that in which **EnterStream()** was called. If you run in the background, **EnterStream()** returns immediately; if not, the function doesn't return until the object has exited the stream.

Of initial interest, here, is the "stream function" that you must supply: This is global C function or static C++ member function that's invoked once for each buffer that the BSubscriber receives. The protocol for the function (which is **typedef**'d as **StreamFn**) is:

bool ***stream_function*(**void **arg*, char **buffer*, long *count***)**

- *arg* is the same as the *arg* argument that you passed to **EnterStream()**.
- *buffer* is a pointer to the buffer that has just arrived.
- *count* is the number of bytes of data in the buffer.

You have to implement the stream function yourself; the Media Kit doesn't supply any stream function candidates. From within your implementation of the function, you're expected to process the data in *buffer* as fits your intentions. As mentioned earlier, your processing should be designed with efficiency in mind. The only rule by which you should abide is this:

Don't Clear the Buffer

If you're generating data, you should *add* it into the data that you find in the buffer. Thank-you.

When you're done with your processing, you simply return from the stream function. You don't have to do anything to send the buffer to the next BSubscriber in the stream; the Media Kit takes care of that for you. The value that the stream function returns is important: If it returns **TRUE**, the BSubscriber continues receiving buffers; if it returns **FALSE**, the object is removed from the stream.

Exiting the Stream

There are two ways to remove a BSubscriber from a stream. The first was mentioned above: Return **FALSE** from the stream function. The second method is to call **ExitStream()** directly. The **ExitStream()** function is particularly useful if you're running the stream function in the background and you want to pull the trigger from another thread.

Whichever method is used, the BSubscriber's "completion function" is invoked upon exiting the stream. This is an optional call-back function, similar to the stream function in its application, that you supply as the fifth argument to **EnterStream()**:

EnterStream(..., CompletionFn *completionFunction*, ...**)**

The protocol for the completion function is:

long ***completion_function*(**void **arg*, long *error***)**

- The *arg* value is, again, taken from the **EnterStream()** call.
- *error* is a code that explains why the BSubscriber is exiting the stream.

Normally, *error* is **B_NO_ERROR**. This means that the BSubscriber is exiting naturally: Either because the stream function returned **FALSE** or because **ExitStream()** was called. If error is **B_TIMED_OUT**, then the BSubscriber is exiting because of a delay in receiving the next buffer. (You set the time-out limit through BSubscriber's **SetTimeout()** function, specifying the limit in microseconds; by default the object will wait forever.) Any other error code will have been generated by a lower-level entity and can be lumped into the general category of "something went wrong."

The completion function is executed in the same thread as the stream function. If this isn't a background thread, the value returned by the completion function is then returned by **EnterStream()**. If you *are* using a background thread, the return value is lost.

You can perform whatever clean-up is necessary in your implementation of the completion function. The only thing that you mustn't do in the completion function is delete the BSubscriber itself.

Processing Data in a Member Function

Typically, the stream functions is implemented as a "dummy" static member function of some class. In this case, **EnterStream()**'s *arg* argument is a pointer to an instance of that class. In the implemention of the static function, the "real" stream function is invoked on the *arg* pointer that the function receives. The class that implements the functions derive from BSubscriber.

For example, in the (fictitious) SoundDuller class, a static function called **_dull_sound()** and a non-static function **DullSound()** are defined. Both of these functions are private. In addition, it defines public **Start()** and Stop() functions that will run the show, and some private variables—including a BAudioSubscriber object—that it requires to perform:

```
class SoundDuller : public BObject
{
    public:
        void Start(void);
        void Stop(void);

    private:
        static bool _dull_sound(void *arg,
                                char *buf,
                                long count);
        bool DullSound(char *buf, long count);

        BAudioSubscriber a_sub;
        short previous;
}
```

The implementation of **_dull_sound()** casts the arg pointer and then invokes **DullSound()**:

```
bool SoundDuller::_dull_sound(void *arg, char *buf, long count)
{
    return (((SoundDuller *)arg)->DullSound(buf,count));
}
```

DullSound() performs the actual stream data processing. The function shown here implements a simple low-pass filter (the "HelloWorld" of signal processing). The function assumes that the stream data is one channel of 16-bit sound:

```
bool SoundDuller::DullSound(char *buf, long count)
{
    long short_count = count/2;
    short *s_buf = (short *)buf;

    while (short_count-- > 0) {
        *s_buf += previous;
        previous = *s_buf++;
    }
}
```

The **Start()** function initializes the BAudioSubscriber and the **previous** variable, and then calls **EnterStream()**:

```
void SoundDuller::Start(void)
{
    if (a_sub.Subscribe(B_DAC_STREAM, B_SHARED_SUBSCRIBER_ID,
                        FALSE) < B_NO_ERROR)
        return;
    previous = 0;

    /* Enter at the stream's tail; run in the background. */
    a_sub.EnterStream(NULL, FALSE,
                        this, _make_dull, NULL, TRUE);
}
```

Stop() removes the subscriber from the stream by calling **ExitStream()**. The function's argument says whether we want to wait until the object is *really* out of the stream; it's always a good idea to re-synchronize if the subscriber is running in the background:

```
void SoundDuller::Stop(void)
{
    a_sub.ExitStream(TRUE);
    a_sub.Unsubscribe();
}
```

Sound details used in this example, such as the meaning of the **B_DAC_STREAM** constant, are explained in the BAudioSubscriber class. For another example of a stream function implementation, see the BSoundFile class.

Constructor and Destructor

BSubscriber()

BSubscriber(const char **name* = NULL**)**

Creates and returns a new BSubscriber object. The object can be given a name; the name needn't be unique.

After creating a BSubscriber, you typically do the following (in this order):

- Subscribe the object to a buffer stream by calling **Subscribe()**.
- Allow the object to begin receiving buffers by calling **EnterStream()**.

The construction of a BSubscriber never fails. This function doesn't set the object's **Error()** value.

See also: **Subscribe(), EnterStream()**

~BSubscriber()

virtual **~BSubscriber(**void**)**

Destroys the BSubscriber. You should never delete a BSubscriber from within an implementation of the object's stream function or completion function.

It isn't necessary to tell the object to exit the buffer stream or to unsubscribe it before deleting. These actions will happen automatically.

Member Functions

Clique()

subscriber_id **Clique(**void**)**

Returns the clique (a **subscriber_id** value) that this BSubscriber used in its most recent attempt to subscribe. The attempt need not have been successful, nor is there any guarantee that the object hasn't since unsubscribed. If the object hasn't attempted to subscribe, this returns **B_NO_SUBSCRIBER_ID**.

See also: **Subscribe()**

EnterStream()

virtual long **EnterStream(**subscriber_id *neighbor*,
bool *before*,
void **arg*,
StreamFn *streamFunction*,
CompletionFn *completionFunction*,
bool *background***)**

Causes the BSubscriber to begin receiving buffers of data from its stream. The object must have successfully subscribed (through a call to **Subscribe()**) for this function to succeed.

The arguments to this function (and the function in general) is the topic of most of the overview to this class; look there for the whole story. Briefly, the arguments are:

- *neighbor* identifies the BSubscriber that this object will stand next to in the buffer stream. If neighbor is **NULL**, this BSubscriber will be positioned at the front or the back of the stream (depending on the value of the next argument).

- *before*, if **TRUE**, places this BSubscriber immediately before neighbor in the stream. If it's **FALSE**, this object is placed after neighbor. If neighbor was **NULL**, this object is placed at the front or back of the stream as *before* is **TRUE** or **FALSE**.

- *arg* is a pointer-sized value that's forwarded as an argument to the stream and completion functions (specified in the next two arguments to **EnterStream()**).

- *streamFunction* is a global function that's called once for every buffer that's sent to the BSubscriber. The protocol for the function is:

  bool ***stream_function(***void **arg*, char **buffer*, long *count***)**

  The *arg* argument, here, is taken literally as the *arg* value passed to **EnterStream()**. A pointer to the buffer itself is passed as *buffer*; *count* is the number of bytes of data in the buffer. If the stream function returns **TRUE**, the object continues to receive buffers; if it returns **FALSE**, it exits the stream.

- *completionFunction* is a global function that's called after the BSubscriber has finished processing its last buffer. Its protocol is:

  long ***completion_function(***void **arg*, long *error***)**

  arg, again, is taken from the argument to **EnterStream()**. *error* is a code that describes why the object is leaving the stream: **B_NO_ERROR** means that the object has received an **ExitStream()** call, or that the stream function returned **FALSE**; an error of **B_TIMED_OUT** means the time limit between buffer receptions (as set through **SetTimeout()**) has expired. If the function isn't running in the background (as described in the next argument), the value returned by the completion function becomes the value that's returned by **EnterStream().**

  The completion function is optional. A value of **NULL** is accepted.

- *background*, if **TRUE**, causes the stream and completion functions to be executed in a separate thread (the Kit spawns the thread for you). In this case, **EnterStream()** returns immediately. If it's **FALSE**, the functions are executed synchronously within the **EnterStream()** call.

If the designated neighbor isn't in the buffer stream, **EnterStream()** returns **B_SUBSCRIBER_NOT_FOUND**. If the BSubscriber is already in the stream, **B_BAD_SUBSCRIBER** is returned.

If *background* is **TRUE**, **EnterStream()** immediately returns **B_NO_ERROR**; if it's **FALSE**, **EnterStream()** returns the value returned by the completion function. If a completion function isn't supplied, **EnterStream()** returns a value that indicates the success of the communication with the server; unless something's gone wrong, it should return **B_NO_ERROR**. In all cases, the **Error()** value is set to the value returned here.

See also: **ExitStream()**

Error()

long **Error(**void**)**

Returns an error code that reflects the success of the function that was most recently invoked upon this object. The error codes that a particular function uses are listed in that function's description.

ExitStream()

virtual long **ExitStream(**bool *andWait* = FALSE**)**

Causes the BSubscriber to leave the buffer stream after it completes the processing of its current buffer. If *andWait* is **TRUE**, the function doesn't return until the object has completed processing this final buffer and has actually left the stream. If a completion function was supplied in the **EnterStream()** invocation, it will run to completion before **ExitStream()** returns. If *andWait* is **FALSE** (the default), **ExitStream()** returns immediately.

If the object isn't in the stream, the **B_SUBSCRIBER_NOT_FOUND** is returned. Otherwise the function returns **B_NO_ERROR**.

Note: In release 1.1d7, **ExitStream()** doesn't return a reliable value—but it does set the error code properly.

See also: **EnterStream()**

ID()

subscriber_id **ID(**void**)**

Returns the **subscriber_id** value that uniquely identifies this BSubscriber. A subscriber ID is issued when the object subscribes to a stream; it's withdrawn when the object unsubscribes. ID values are used, primarily, to position a BSubscriber with respect to some other BSubscriber within a buffer stream.

If the BSubscriber isn't currently subscribed to a stream, **B_NO_SUBSCRIBER_ID** is returned.

IsInStream()

bool **IsInStream(**void**)**

Returns **TRUE** if the object is currently in a stream; otherwise it returns **FALSE.**

Name()

const char ***Name(**void**)**

Returns a pointer to the name of the BSubscriber. The name is set through an argument to the BSubscriber constructor.

SetTimeout(), Timeout()

void **SetTimeout(**double *microseconds***)**
double **Timeout(**void**)**

These functions set and return the amount of time, measured in microseconds, that a BSubscriber that has entered the buffer stream is willing to wait from the time that it finishes processing one buffer till the time that it gets the next. If the time limit expires before the next buffer arrives, the BSubscriber exits the stream and the completion function is called with its *error* argument set to **B_TIMED_OUT.**

A time limit of 0 (the default) means no time limit—the BSubscriber will wait forever for its next buffer.

See also: **EnterStream()**

StreamParameters()

long **StreamParameters(**long ******bufferSize***,**
long ******bufferCount***,**
bool ******isRunning***,**
long ******subscriberCount***,**
subscriber_id ******clique***)**

Returns information about the stream to which the BSubscriber is currently subscribed:

- *bufferSize* is the size, in bytes, of the buffers that the object will receive.
- *bufferCount* is the number of buffers that are used in the stream.
- *isRunning* is **TRUE** if the stream is currently running, and **FALSE** if it isn't.
- *subscriberCount* is the number of BSubscriber objects that are currently subscribed to the stream (whether or not they've actually entered).
- *clique* is the currently enforced clique value for the stream.

You can set the buffer size and buffer count parameters (and so fine-tune the latency of the stream) through the **SetStreamBuffers()** function. *isRunning* can be toggled through calls to **StartStreaming()** and **StopStreaming()**. The other two parameters (*subscriberCount* and *clique*) vary as subscribers come and go.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_BAD_SUBSCRIBER** is returned. Otherwise, the function returns **B_NO_ERROR**.

SetStreamBuffers()

long **SetStreamBuffers(**long *bufferSize*, long *bufferCount***)**

Sets the size (in bytes) and number of buffers that are used to transport data through the stream. Although it's up to the server to provide reasonable default values, you can fine-tune the performance of the stream by fiddling with this function:

- By decreasing the size and/or number of buffers, you can decrease the maximum latency of the stream (the time it takes for a buffer to get from one end of the stream to the other). However, if you go too far in this direction, you run the risk of falling out of real time.
- By increasing the buffer size and count, you help ensure the real-time integrity of the stream, but you increase its maximum latency.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_RESOURCE_UNAVAILABLE** is returned. Otherwise, the function returns **B_NO_ERROR**.

The Audio Server initializes its streams to use eight buffers (per stream), where each buffer is a single page (4096 bytes). Currently, there's no way to automatically restore these default values after you've mangled one of the audio streams.

StartStreaming(), StopStreaming()

long **StartStreaming(**void**)**
long **StopStreaming(**void**)**

Starts and stops the passing of buffers through the stream to which the BSubscriber is subscribed. By default, the stream begins running when the first BSubscriber enters it, and it stops when the final remaining BSubscriber exits. You should only need to call **StartStreaming()** or **StopStreaming()** if you want to interrupt this automation.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_RESOURCE_UNAVAILABLE** is returned. Otherwise, the function returns **B_NO_ERROR**.

Subscribe()

virtual long **Subscribe(**long *stream*, subscriber_id *clique*, bool *willWait***)**

Asks for admission into the server's list of BSubscribers to which it (the server) will send buffers of data. Subscribing doesn't cause the BSubscriber to begin receiving buffers, it simply gives the object the *right* to do so. (To receive buffers, you must invoke **EnterStream()** on a BSubscriber that has successfully subscribed.)

The arguments are described fully in the overview to this class. Briefly, they are:

- *stream* is a constant that identifies the specific stream within the server that you wish to subscribe to. The Audio Server provides two stream constants: **B_DAC_STREAM** (sound-out), and **B_ADC_STREAM** (sound-in).
- The *clique* argument is used as a "key" to the server. If there are no other currently-subscribed objects, any clique value is accepted and the BSubscriber is admitted. Subsequent subscriptions (by other BSubscribers) are then denied if they don't match this clique value. Conversely, if some other object has successfully subscribed (and hasn't since unsubscribed) this object must pass the clique value by which the currently-subscribed object gained admittance. The special **B_INVISIBLE_SUBSCRIBER_ID** value, when used as the clique, will let you invade any stream, any time.
- The *willWait* argument tells the server whether this BSubscriber will wait for the coast to clear if the immediate attempt to subscribe is denied.

A successful subscription returns **B_NO_ERROR**. If the subscription is denied (because *stream* doesn't identify a valid stream, or the *clique* value isn't acceptable) and the BSubscriber isn't waiting, **Subscribe()** returns **RESOURCE_NOT_AVAILABLE**. The **Error()** value is set to the value returned directly here.

Note: The timeout value that you can set through the **SetTimeout()** function doesn't apply to subscription (it only applies to the inter-buffer lacuna). A BSubscriber that's willing to wait for admission might be waiting a long time.

See also: **Unsubscribe()**

Timeout() *see* SetTimeout()

Unsubscribe()

virtual long **Unsubscribe(**void**)**

Revokes the BSubscriber's access to its media server and sets its subscriber ID to **B_NO_SUBSCRIBER_ID**. If the object is currently in a stream, it automatically exits the stream and the object's completion function is called.

When you delete a BSubscriber, it's automatically unsubscribed.

If the object isn't currently subscribed, the function returns **B_BAD_SUBSCRIBER**. Otherwise, it returns **B_NO_ERROR**.

See also: **Subscribe()**

Global Functions, Constants, and Defined Types

This section lists parts of the Media Kit that aren't contained in classes.

Global Functions

beep()

<media/Beep.h>

sound_handle **beep(**void**)**

beep() plays the system beep. The sound is played in a background thread and **beep()** returns immediately. If you want to re-synchronize with the sound playback, pass the **sound_handle** token (returned by this function) as the argument to **wait_for_sound()**. This will cause your thread to wait until the sound has finished playing.

beep() will mix other sounds, but it never waits if the immediate attempt to play is thwarted.

play_sound()

<media/Beep.h>

sound_handle **play_sound(**record_ref *soundRef*,
bool *willMix*,
bool *willWait*,
bool *background***)**

Plays the sound file identified by *soundRef*. The *willMix* and *willWait* arguments are used to determine how the function behaves with regard to other sounds:

- If you want your sound to play all by itself, set *willMix* to **FALSE**. If you don't care if it's mixed with other sounds, set it to **TRUE**.
- If you want your sound to play immediately (whether or not you're willing to mix), set *willWait* to **FALSE**. If you're willing to wait for the sound playback resources to become available, set *willWait* to **TRUE**.

Note that setting *willMix* to **TRUE** doesn't ensure that your sound will play immediately. If the sound playback resources are claimed for exclusive access by some other process, you'll be blocked, even if you're willing to mix.

The background argument, if **TRUE**, tells the function to spawn a thread in which to play the sound. The function, in this case, returns immediately. If background is **FALSE**, the sound is played synchronously and **play_sound()** won't return until the sound has finished.

The **sound_handle** value that's returned is a token that represents the sound playback. This token is only valid if you're playing in the background; you would use it in a subsequent call to **stop_sound()** or **wait_for_sound()**. If the ref doesn't represent a file, or if the sound couldn't be played, for whatever reason, **play_sound()** returns a negative integer.

stop_sound()

<media/Beep.h>

long **stop_sound(**sound_handle *handle***)**

Stops the playback of the sound identified by *handle*, a value that was returned by a previous call to **beep()** or **play_sound()**. The return value can be ignored.

wait_for_sound()

<media/Beep.h>

long **wait_for_sound(**sound_handle *handle***)**

Causes the calling thread to block until the sound identified by *handle* has finished playing. The *handle* value should have been returned by a previous call to **beep()** or **play_sound()**. Currently, **wait_for_sound()** always returns **B_NO_ERROR**.

Constants

Byte Order Constants

<media/MediaDefs.h>

Constant	Meaning
B_BIG_ENDIAN	MSB first
B_LITTLE_ENDIAN	LSB first

These constants are used by BAudioSubscriber and BSoundFile objects to describe the order of bytes within a sound sample.

Sound File Formats

<media/SoundFile.h>

Constant	Meaning
B_UNKNOWN_FILE	The file contains “raw” data
B_AIFF_FILE	AIFF format
B_WAVE_FILE	WAVE format
B_UNIX_FILE	Sun/NeXT/SGI etc. format

These constants represent the sound file formats that are recognized by the BSoundFile class.

Media Thread Priority

<media/MediaDefs.h>

Constant	Value
B_MEDIA_LEVEL	Same as **B_REAL_TIME_PRIORITY**

All threads that are spawned by the Media Kit are given a priority of **B_MEDIA_LEVEL**; this is the same as **B_REAL_TIME_PRIORITY**, the highest priority defined by the Kernel Kit.

No-Change Constant

<media/MediaDefs.h>

Constant	Meaning
B_NO_CHANGE	Don’t change the value of this parameter

The **B_NO_CHANGE** constant is used in multiple-parameter-setting functions (such as BAudioSubscriber’s **SetSampleParameters()** to indicate that you don’t want a particular parameter to change its current setting (while changing the values of other parameters).

Sample Format Constants

<media/MediaDefs.h>

Constant	Meaning
B_LINEAR_SAMPLES	Linear quantization
B_FLOAT_SAMPLES	Floating-point samples
B_MULAW_SAMPLES	Mu-law encoding
B_UNDEFINED_SAMPLES	Anything else

These constants represent the sample formats that are recognized by the sound hardware.

Subscriber IDs

<media/MediaDefs.h>

Constant	Meaning
B_SHARED_SUBSCRIBER_ID	Share the stream with other subscribers.
B_INVISIBLE_SUBSCRIBER_ID	Subscribe to the stream regardless of the clique.
B_NO_SUBSCRIBER_ID	The BSubscriber object isn't subscribed.

The first two subscriber ID constants are most commonly used as "clique" values, passed to the **EnterStream()** function. The final ID, **B_NO_SUBSCRIBER_ID**, is the default, subscriber-isn't-subscribed subscriber ID value.

The subscriber ID constants are type as **subscriber_id** values.

Defined Types

sound_handle

<media/Beep.h>

typedef sem_id **sound_handle**

The **sound_handle** type is a token that represents sounds that are currently being played through calls to **beep()** or **play_sound()**.

subscriber_id

<media/MediaDefs.h>

typedef sem_id **subscriber_id**

The **subscriber_id** type is a token that uniquely identifies—system-wide—a BSubscriber object for a particular server.