The NCDaudio Programming Library

Applications may communicate with the NCDaudio server using a library of C-language routines. The functions in this library generate network protocol packets and provide a number of convenience utilities to simplify operations performed by most applications.

Using the NCDaudio API

Programs that wish to use the NCDaudio service should include the following file:

```
#include <audio/audiolib.h>
```

and install the `audio` directory where it can be found by the C preprocessor (*e.g.*, in `/usr/include` or use `-I/other/directory` on the compiler command line).

This header file provides a number of symbolic constants and function prototypes for both the low-level routines and the higher level utilities used to communicate with the NCDaudio server.

The NCDaudio library is called `libaudio.a` and should be referenced on the link line before any X Window System libraries, *e.g.*:

```
cc -o myprog myprog.o libaudio.a \
    -lXm -lXt -lXext -lX11
```

If you have installed `libaudio.a` in one of the directories known by the linker, you can refer to it as `-laudio` instead. If you are not using X within your audio program, you must link with the library `libXau.a` after `libaudio.a`.

Sample Application Use

The NCDaudio service is designed to support a wide range of applications. However, many applications may find the high-level utility routines in the library to be sufficient. In particular, programs that simply wish to play pre-recorded sound data require only a few lines of code.

For example, a program that uses the X Window System to manage its graphical user interface would use the audio library to perform the following operations:

1. In the initialization portion of the program, open a connection to the audio server (after having opened the X display).
2. If the X toolkit is handling the program's main loop, add an input handler to deal with any events from the audio server.

3. When a sound is to be played, call one of the utility routines to read in the file, send the data to the server, and direct the data to a speaker.

More sophisticated applications can use the lower level routines to construct their own audio buckets and audio data flows.

Major Data Types

↳ Many applications can ignore the details of the different data types and object attributes and just skip to the next chapter.

The NCDaudio application programming interface makes heavy use of the following data types:

AuID – This is a generic resource identifier, used to refer to objects in the audio server. The special value **AuNone** indicates no object.

AuDeviceID – This is an identifier referring to a physical or virtual device.

AuBucketID – This is a kind of device identifier that refers to a ***bucket*** object in the server. Buckets are used to store audio data in the server, typically for small sounds that are played many times.

AuFlowID – This is an identifier referring to a ***flow*** object in the server. Flows contain the instructions for moving audio data from one or more sources to one or more destinations.

AuTime – This is the server time in milliseconds.

AuMask – This is a set of bits that are ORed together.

AuBool – This is a simple boolean whose values are **AuFalse** and **AuTrue**.

AuStatus – This is a number representing a status code.

String data is manipulated using the following structure:

```
struct _AuString {
    int         type;
    int         len;
    char        *data;
} AuString;
```

The *type* field specifies the format in which string data is encoded. Currently, the only defined value is:

AuStringLatin1 – 8-bit characters encoded in ISO8859/1; for convenience, the library also null-terminates the *data* field in any **AuString** objects it creates or fills in.

The *len* field specifies the number of bytes in the *data* field, which specifies the actual description text.

Non-integer values (used to represent data constants and device gain percentages) are expressed using fixed-point numbers:

AuFixedPoint – This is a signed long whose contents are in increments of 65536.

which can be created using the following macros:

```
AuFixedPoint AuFixedPointFromSum (
    short           integralpart,
    unsigned short  fractionalpart)
```

or

```
AuFixedPoint AuFixedPointFromFraction (
    short           numerator,
    unsigned short  denominator)
```

The integer value of a fixed point number can be obtained using:

```
int AuFixedPointRoundDown (
    AuFixedPoint   fp)
```

```
int AuFixedPointRoundUp (
    AuFixedPoint   fp)
```

The following macros can be used to obtain the signed integral and unsigned fractional portions of the sum that could be used to recreate the value:

```
int AuFixedPointIntegralAddend (
    AuFixedPoint   fp)
```

```
unsigned int AuFixedPointFractionalAddend (
    AuFixedPoint   fp)
```

Most applications will only need to use the two creator macros.

Status Codes

The following status codes are used in NCDaudio:

AuSuccess – No problems were encountered.

AuBadRequest – An invalid request was received.

AuBadValue – An invalid numeric value was used.

AuBadDevice – An invalid AuDeviceID was given.

AuBadBucket – An invalid AuBucketID was given.

AuBadFlow – An invalid AuFlowID was given.

AuBadElement – An invalid flow element number was used.

AuBadMatch – Elements with differing numbers of tracks were hooked together.

AuBadAccess – An attempt was made to perform an operation that is not permitted.

AuBadAlloc – Insufficient resources available.

AuBadConnection – The operating system reported an unrecoverable error on the connection to the server.

AuBadIDChoice – An invalid identifier value was given in a Create request.

AuBadName – An invalid name or description text was given.

AuBadLength – A request packet was received with the wrong amount of data.

AuBadImplementation – A requested operation is not implemented in this version.

Other constants are explained in the various sections where they are used below.

Audio Data Formats

Audio data consists of a sequence of numbers ranging in value from -1.0 to +1.0, encoded in a variety of ways. The following audio data formats are supported by the NCDaudio service:

AuFormatULAW8 – 8 bits per value, encoded as the sign and logarithm of each sample.

AuFormatLinearUnsigned8 – 8 bits per value, encoded as [0..255].

AuFormatLinearSigned8 – 8 bits per value, encoded as [-127..+127]. It is less commonly used than the unsigned version.

AuFormatLinearSigned16MSB – 16 bits per value, stored most-significant byte first, encoded as [-32767..+32767].

AuFormatLinearUnsigned16MSB – 16 bits per value, stored most-significant byte first, encoded as [0..65535].

AuFormatLinearSigned16LSB – 16 bits per value, stored least-significant byte first, encoded as [-32767..+32767].

AuFormatLinearUnsigned16LSB – 16 bits per value, stored least-significant byte first, encoded as [0..65535].

Of the 16 bits per sample formats, the MSB versions are typically used on computers sold by Sun, NeXT, and Apple. The LSB versions are usually used on Intel-based personal computers and computers sold by DEC.

Flow Elements

The following element types may be used in a flow of audio data:

`AuElementTypeImportClient` – Data sent from the client over the network becomes an input source in the flow.

`AuElementTypeImportDevice` – Data from a device becomes an input source in the flow.

`AuElementTypeImportBucket` – Data that has been stored in the server becomes an input source in the flow.

`AuElementTypeImportWaveForm` – Data of a given wave form becomes an input source in the flow.

`AuElementTypeImportRadio` – Data from a "station" being broadcast over the local area network becomes an input source in the flow.

`AuElementTypeBundle` – Tracks of data from one or more branches in the flow are bundled together to form a new branch.

`AuElementTypeMultiplyConstant` – Values flowing along a given branch are multiplied by a given constant.

`AuElementTypeAddConstant` – Values flowing along a given branch are added to a given constant.

`AuElementTypeSum` – Values flowing along two or more branches in the flow are summed together.

`AuElementTypeExportClient` – Data from the flow is made available to the client for fetching over the network.

`AuElementTypeExportDevice` – Data from the flow is sent to a device attached to the server.

`AuElementTypeExportMonitor` – Data from the flow is periodically summarized to the client via a `MonitorNotify` event.

`AuElementTypeBucket` – Data from the flow is stored in a bucket object in the server.

`AuElementTypeRadio` – Data from the flow is directed to a given local area network audio broadcaster.

Wave Form Generators

Several virtual input sources are provided by the audio server:

`AuWaveFormSquare` – A square wave from -1.0 to +1.0.

`AuWaveFormSine` – A sine wave from -1.0 to +1.0.

`AuWaveFormSaw` – A saw wave from -1.0 to +1.0.

`AuWaveFormConstant` – A constant value of 1.0.

The amplitude of a wave form can easily be changed by multiplying the output of the wave form generator by a constant.

Input and Output Components

The NCDaudio service supports a variety of input sources (including physical hardware such as microphones and CD players) and output destinations (such as speakers). Some components, particularly the buckets used to store audio data in the server, can be used as both inputs and outputs.

These components are broken into three classes of objects: ***devices***, ***buckets***, and ***radios***, all of which share a common set of base attributes:

```
typedef struct _AuCommonPart {
    AuMask          value_mask;
    AuMask          changable_mask;
    AuID            id;
    unsigned int    kind;
    AuMask          use;
    int             format;
    int             num_tracks;
    AuMask          access;
    AuString        description;
} AuCommonPart;
```

The *value_mask* field specifies which fields are set for this object. The fields not listed in this mask are set to zero. The *changable_mask* field specifies which fields may be set by the application. Both fields consist of the union of the following mask values:

AuCompCommonIDMask
AuCompCommonKindMask
AuCompCommonUseMask
AuCompCommonFormatMask
AuCompCommonNumTracksMask
AuCompCommonAccessMask
AuCompCommonDescriptionMask

The *id* field specifies the resource identifier associated with this device. This value is used in any flow elements or events.

The *kind* field specifies what type of device this object represents:

AuComponentKindPhysicalInput
AuComponentKindPhysicalOutput
AuComponentKindBucket
AuComponentKindRadio

The *use* field is a mask representing the types of elements in which this device may be used:

AuComponentUseImportMask
AuComponentUseExportMask

The *format* field specifies the format currently in use by this device.

The *num_tracks* field specifies the number of sequences of audio data used by this device. Mono devices use 1 track; stereo devices use 2 (of which the left channel is usually listed first). The number of tracks is implementation dependent.

The *access* field specifies what clients other than the creator of the device are permitted to do with it:

AuAccessImportMask
AuAccessExportMask
AuAccessDestroyMask
AuAccessListMask

The *description* field specifies a text string that describes this device

Devices

Physical devices that are attached to the X terminal or PC are described using the common part listed above as well as the following:

```
typedef struct _AuDevicePart {
    unsigned int   min_sample_rate;
    unsigned int   max_sample_rate;
    AuMask         location;
    AuFixedPoint   gain;
    int            line_mode;
    int            num_children;
    AuDeviceID     *children;
} AuDevicePart;

typedef struct _AuDeviceAttributes {
    AuCommonPart   common;
    AuDevicePart   device;
} AuDeviceAttributes;
```

The following masks are used in the common *value_mask* and *changable_mask* fields:

AuCompDeviceMinSampleRateMask
AuCompDeviceMaxSampleRateMask
AuCompDeviceLocationMask
AuCompDeviceGainMask

AuCompDeviceLineModeMask
AuCompDeviceChildrenMask

The *min_sample_rate* and *max_sample_rate* fields specify the range of sample rates that this device can handle.

The *location* field is a mask of the following values specifying a hint as to the physical location of the device relative to the user:

AuDeviceLocationLeftMask
AuDeviceLocationCenterMask
AuDeviceLocationRightMask
AuDeviceLocationTopMask
AuDeviceLocationMiddleMask
AuDeviceLocationBottomMask
AuDeviceLocationBackMask
AuDeviceLocationFrontMask
AuDeviceLocationInternalMask
AuDeviceLocationExternalMask

The *gain* field specifies a hardware volume control.

The *line_mode* field specifies which type of amplification circuit should be used for input devices. CD players and microphones are typically attached to a higher level circuit than are record players.

AuDeviceLineModeNone
AuDeviceLineModeLow
AuDeviceLineModeHigh

The *num_children* and *children* fields are used by aggregate devices to specify the subdevices that are actually used.

Buckets

Audio data can be stored in the server in objects called ***buckets,*** which are described using the common part listed above as well as the following:

```
typedef struct _AuBucketPart {
    unsigned short sample_rate;
    unsigned long  num_samples;
} AuBucketPart;

typedef struct _AuBucketAttributes {
    AuCommonPart   common;
    AuBucketPart   bucket;
} AuBucketAttributes;
```

The following masks are used in the common *value_mask* and *changable_mask* fields:

AuCompBucketSampleRateMask
AuCompBucketNumSamplesMask

The *sample_rate* field specifies the granularity of the audio samples stored in the bucket.

The *num_samples* field specifies the number of samples stored in the bucket.

Radios

Audio data that is broadcast on a local area network using UDP/IP datagrams can be captured using special objects called a ***radios,*** which are described using the common part listed above as well as the following:

```
typedef struct _AuRadioPart {
    int             station;
} AuRadioPart;

typedef struct _AuRadioAttributes {
    AuCommonPart   common;
    AuRadioPart    radio;
} AuRadioAttributes;
```

The following masks are used in the common *value_mask* and *changable_mask* field:

AuCompRadioStationMask

The *station* field describes which of the various broadcast streams should be used.

Connecting to the Audio Server

Like the X Window System, NCDaudio requires applications to open a connection to the server before any operations can be performed.

Opening a Connection to the Audio Server

Before an application can send or manipulate sound data, it must create a network connection to the audio server using:

```
AuServer *
AuOpenServer (
    const char      *name,
    int             num_authproto,
    const char      *authproto,
    int             *num_authdata,
    const char      *authdata,
    char            **server_message)
```

This routine attempts to open a connection to the audio server specified by *name*. If this parameter is not NULL, its format depends upon the network to be used:

- ❑ ***TCP/IP*** – the string should be "tcp/*hostname:portnum*"; where *hostname* is the name or numeric IP address of the machine on which the audio server is running, and *portnum* is the TCP/IP port number on which the server is listening. If the "tcp/" prefix is not given (making the name have the same style as an X server name), 8000 is added to *portnum*.
- ❑ ***DECnet*** – the string should be "decnet/*nodename::num*"; where *nodename* is the name or numeric DECnet address of the machine on which the audio server is running, and *num* is the DECnet task listening on AUDIO$*num*. If the "decnet" prefix is not given, it is added automatically.

If *name* is NULL, this routine looks at the contents of the AUDIOSERVER environment variable and uses that if it is set. If that variable is not set, the routine looks at the contents of the DISPLAY environment variable.

When used with X applications, *name* is typically set to either the value passed to `XOpenDisplay` or the results of calling `DisplayString(dpy)`.

The strings *authproto* and *authdata* specify the authorization protocol and data to use in making a connection to the server. If they are both NULL, the routine will look for the data that an X application would use were it connecting to the X server.

If a connection can be made, a pointer to an **AuServer** object describing the connection is returned and **server_message* is set to NULL.

Otherwise, **server_message* is set to a string if an error was returned by the server or NULL if the routine was unable to open a connection or allocate enough memory. The returned string may be freed using `AuFree()`.

Closing a Connection

When an application no longer wishes to use the audio server, it can close its connection and release any associated memory using:

```
void
AuCloseServer (AuServer *audio)
```

This routine deallocates the memory pointed to by *audio*.

Registering the Audio Connection with an Xt Input Handler

Audio can most easily be added to X Window System applications written with X toolkits such Motif, OLIT, or MoOLIT by letting the X toolkit continue to control the main dispatching loop. The routine described below sets up an input handler that will call into the NCDaudio library whenever an audio event is received.

To use this routine, the application should include the following header files:

```
/* somewhere after <audio/audiolib.h> */
#include <audio/Xtutil.h>
```

The latter file provides a function prototype for the following routine:

```
XtInputID
AuXtAddAudioHandler (
    XtAppContext   app_context,
    AuServer       *audio)
```

This routine should be called after **AuOpenServer()**. It arranges for the audio library to be called whenever events are received from the audio server. It also sets up an Xt work procedure that flushes the audio library's network buffers before Xt attempts to block waiting for data from X or NCDaudio.

Information Returned from AuOpenServer()

↳ Many applications can ignore these fields and skip to the next chapter.

The **AuServer** object describes all of the information pertaining to a given connection to the audio server. Its contents should not be referenced explicitly by applications; instead the following macros should be used. Many applications can ignore this information and simply use the convenience routines.

```
int
AuServerConnectionNumber (
    AuServer        *audio)
```

This macro returns the file descriptor used to communicate with the audio server.

```
const char *
AuServerString (AuServer *audio)
```

This macro returns a string giving the name of the audio server, suitable for passing to `AuOpenServer()`.

```
const char *
AuServerVendor (AuServer *audio)
```

This macro returns a string giving the name of the organization that implemented the audio server.

```
int
AuServerProtocolMajorVersion (
    AuServer        *audio)
```

This macro returns the major version of the protocol expected by the server. This number is incremented whenever incompatible changes are made in the underlying NCDaudio protocol.

```
int
AuServerProtocolMinorVersion (
    AuServer        *audio)
```

This macro returns the minor version of the protocol expected by the server. This number is incremented whenever compatible changes are made in the underlying NCDaudio protocol.

```
int
AuServerMinSampleRate (AuServer *audio)

int
AuServerMaxSampleRate (AuServer *audio)
```

These two macros return the minimum and maximum sample rates supported by the server.

Tracks

The following macro returns the maximum number of tracks per device or flow element supported by the server:

```
int AuServerNumTracks (AuServer *audio)
```

NCD Network Display Stations initially support up to 32 tracks.

Formats

The following macros return information about the formats supported by the server:

```
int AuServerNumFormats (AuServer *audio)
```

This macro returns the number of formats available.

```
int
AuServerFormat (
    AuServer        *audio,
    int             num)
```

This macro returns the *num*th format supported by the server (beginning with 0).

```
int AuSizeofFormat (int format)
```

This macro returns the size in bytes of a sample encoded in the specified format.

Device Attributes

The following macros return information about the physical devices provided by the server:

```
int
AuServerNumDevices (AuServer *audio)
```

This macro returns the number of devices available.

```
AuDeviceAttributes *
AuServerDevice (
    AuServer        *audio,
    int             num)
```

This macro returns a pointer to a structure describing the *num*th device (beginning with 0).

Bucket Attributes

The following macros return information about the buckets that are built into the server:

```
int AuServerNumBuckets (AuServer *audio)
```

This macro returns the number of built-in buckets.

```
AuBucketAttributes *
AuServerBucket (
    AuServer        *audio,
    int             num)
```

This macro returns a pointer to a structure describing the *num*th built-in bucket (beginning with 0).

Radio Attributes

The following macros return information about the radio objects provided by the server:

```
int AuServerNumRadios (AuServer *audio)
```

This macro returns the number of radios available.

```
AuRadioAttributes *
AuServerRadio (
    AuServer        *audio,
    int             num)
```

This macro returns a pointer to a structure describing the *num*th radio (beginning with 0).

Wave Forms

The following macros return information about the wave form generators that are provided by the server:

```
int
AuServerNumWaveForms (
    AuServer        *audio)
```

This macro returns the number of wave forms available.

```
int
AuServerWaveForm (
    AuServer        *audio,
    int             num)
```

This macro returns the *num*th (beginning with 0) wave form value.

Element Types

The following macros return information about the element types that may be used in a flow object:

```
int AuServerNumElements (AuServer *audio)
```

This macro returns the number of types of elements supported.

```
int
AuServerElement (
    AuServer        *audio,
    int             num)
```

This macro returns the *num*th supported element type (beginning with 0).

Action Types

The following macros return information about the actions that may be attached to flow elements in this server:

```
int AuServerNumActions (AuServer *audio)
```

This macro returns the number of actions that are supported.

```
int
AuServerAction (
    AuServer        *audio,
    int             num)
```

This macro return the *num*th supported action type (beginning with 0).

Manipulating Audio Data Files

Several utility routines are provided for applications that simply wish to play from or record to a file.

How to Use

Programs that wish to use the utilities described in this chapter should include the following header file:

```
/* somewhere after <audio/audiolib.h> */
#include <audio/soundlib.h>
```

This file provides function prototypes for routines that read and write audio data files in a variety of formats.

Playing Audio Files

Several routines are provided for directly playing from and recording to audio files that use the .SND, .WAV, or .VOC formats:

```
AuEventHandlerRec *
AuSoundPlayFromFile (
    AuServer          *audio,
    const char        *filename,
    AuDeviceID        device,
    AuFixedPoint      volume,
    void              (*done_callback)(),
    AuPointer         callback_data,
    AuFlowID          *ret_flow,
    int               *ret_mult_elem,
    int               *ret_monitor_elem,
    AuStatus          *ret_status)
```

This routine loads audio data from the specified *filename* and plays it to the specified *device*. If *device* is **AuNone**, the routine will select an output device that matches the number of tracks in the data.

An initial *volume* at which the data can be specified. To play the data at the level at which it was recorded, this parameter should be set to **AuFixedPointFromSum(1,0)**.

Applications that do not wish to find out when the sound has finished playing or change its volume may specify NULL for the last three parameters to this routine.

If *done_callback* is not NULL, that routine will be called when the data has finished playing:

```
void
done_callback (
     AuServer     *audio,
     AuEventHandlerRec *which,
     AuEvent      *event,
     AuPointer    callback_data)
```

If *ret_flow, ret_mult_elem, ret_monitor_elem* are not NULL, they are set to contain the id of the flow being used to play the sound, the element number of a multiplier that can be adjusted to dynamically change the volume using **`AuSetElementParameters()`**, and the element number of a monitor that can be used to keep track of data as it is played. The playing can be aborted using **`AuSetElement-States()`**.

The following utility routine is provided for those times when the application doesn't care being able to do other things while it is playing audio data:

```
AuBool
AuSoundPlaySynchronousFromFile (
    AuServer        *audio,
    const char      *filename,
    int             volume_percent)
```

This routine takes a *filename* and a volume expressed as a percentage (*i.e.*, from 0 to 100) and returns `AuTrue` if was able to play the data stored in the file; otherwise it returns `AuFalse`.

Recording Audio Files

The following routine is provided for recording data directly from an input device:

```
AuEventHandlerRec *
AuSoundRecordToFile (
    AuServer         *audio,
    const char       *filename,
    AuDeviceID       device,
    AuFixedPoint     gain,
    void             (*done_callback)(),
    AuPointer        callback_data,
    int              line_mode,
    unsigned long    file_type,
    const char       *comment,
    unsigned long    sample_rate,
    int              sample_format,
    AuFlowID         *ret_flow,
    int              *ret_volume_mult,
    AuStatus         *ret_status)
```

This routine records audio data from the specified *device* and stores it into the specified *filename*. The *file_type* specifies the kind of data file that should be written out:

SND_MAGIC
WAVE_MAGIC
VOC_MAGIC

The *sample_rate* and *sample_format* specify the rate and format of the data retrieved from the audio server.

Storing Sound Data in Memory

An alternate set of routines are provided for applications that wish to keep the audio data stored in client memory. The following structure is used to store the header information describing how the data is formatted:

```
typedef struct _SoundHeader {
    unsigned long     sound_type;
    unsigned long     format;
    unsigned long     dataOffset;
    unsigned long     dataSize;
    unsigned long     sampleRate;
    unsigned short    tracks;
    unsigned short    bitsPerSample;
    char              swapped;
    char              is_file;
} SoundHeader;
```

The *sound_type* field specifies one of the file types **SND_MAGIC**, **WAVE_MAGIC**, or **VOC_MAGIC**. The *format* field specified the `AuFormat` that should be used.

The *dataOffset* field specifies where the data should be located relative to the beginning of the file in which it is stored. The *dataSize* field specifies the total length in bytes of the audio data.

The *sampleRate* field specifies the number of samples per second at which the data was recorded.

The *tracks* field specifies the number of tracks of samples in the data. The *bitsPerSample* field specifies size in bits of each sample (8, 16, or 32).

The *swapped* field indicates whether or not the data has been byte-swapped. The *is_file* field indicates whether or not the data came from a file or other random access source.

The number of bytes required per sample can be determined using:

```
int SoundSizeofFormat (int format)
```

This returns the number of bytes in a given `AuFormat` datum.

Reading Sound Files

The following routine can be used to read audio files:

```
FILE *
SoundOpenFileForReading (
    const char        *filename,
    SoundHeader       *get_header,
    char              **ret_comment)
```

This routine opens the specified *filename* for reading. If *get_header* is not NULL, the header information is copied into **get_header*. If *get_comment* is not NULL, then **ret_comment* is set to a malloced copy of the comment string in the sound file. If successful, the open file handle is returned; otherwise NULL is returned.

The audio data is located at byte position *header.dataOffset* in the file. Applications can directly use **fread()** to obtain the data, and **fseek()** to move around in the file. When the application is finished with the file, it can call:

```
void SoundCloseFile (FILE *fp)
```

to close the file descriptor.

Writing Sound Files

The following routine can be used to write audio files:

```
FILE *
SoundOpenFileForWriting (
    const char        *filename,
    SoundHeader       *header,
    const char        *comment)
```

This routine opens the specified *filename* and writes the specified *header* and optional *comment*. Applications can then use **fwrite()** to write the data to the file descriptor. When the application is finished with the file, it can call **SoundCloseFile()** to close the file descriptor.

Storing Sounds in the Server

Several routines are provided for storing sound data in the audio server in objects called ***buckets*** which can later be used as a source for playing or a destination for recording.

Creating Buckets

The following utility routines can be used to create a bucket from audio data stored in a file or application memory:

```
AuBucketID
AuSoundCreateBucketFromFile (
    AuServer              *audio,
    const char            *filename,
    unsigned long         access,
    AuBucketAttributes    **ret_attr,
    AuStatus              *ret_status)
```

This routine reads data from the specified *filename* and creates a bucket for it in the server. The *access* mask specifies which operations (import, export, destroy, and list) may be performed by other clients. If *ret_attr* is non-NULL, it is set to point to a description of the attributes of the created bucket. The bucket, if successfully created, is returned. Otherwise, `AuNone` is returned.

```
AuBucketID
AuSoundCreateBucketFromData (
    AuServer              *audio,
    SoundHeader           *header,
    AuPointer             data,
    const char            *description,
    unsigned long         access,
    AuBucketAttributes    **ret_attr,
    AuStatus              *ret_status)
```

This routine is similar to the previous one except that it uses the *header*, *data*, and *description* provided by the caller instead of reading them from a file.

Destroying a Bucket

When a bucket is no longer needed, the following routine should be called:

```
void AuDestroyBucket (
    AuServer       *audio,
    AuBucketID     bucket,
    AuStatus       *ret_status)
```

Using Buckets

Buckets are used as both sources and destinations in flows.

Playing from Buckets

The following convenience routine can be used to play a sound that has been stored in a bucket:

```
AuEventHandlerRec *
AuSoundPlayFromBucket (
    AuServer        *audio,
    AuBucketID      bucket,
    AuDeviceID      device,
    AuFixedPoint    volume,
    void            (*callback)(),
    AuPointer       callback_data,
    int             count,
    AuFlowID        *ret_flow,
    int             *ret_mult_elem,
    int             *ret_monitor_elem,
    AuStatus        *ret_status)
```

This routine is similar to **AuSoundPlayFromFile** except that it plays the data from the specified *bucket* the number of times specified by *count* (it will always be played at least once).

Recording to Buckets

The following convenience routine can be used to record a sound into a bucket:

```
AuEventHandlerRec *
AuSoundRecordToBucket (
    AuServer        *audio,
    AuBucketID      bucket,
    AuDeviceID      device,
    AuFixedPoint    gain,
    void            (*callback)(),
    AuPointer       callback_data,
    int             line_mode,
    AuFlowID        *ret_flow,
    int             *ret_mult_elem,
    AuStatus        *ret_status)
```

This routine is similar to **AuSoundRecordToFile** except that it records the data to the specified *bucket*.

Reading Data From Buckets

The following routines can be used to read sound data from a bucket back to the application program:

```
AuBool
AuSoundCreateFileFromBucket (
    AuServer        *audio,
    const char      *filename,
    int             sound_format,
    AuBucketID      bucket,
    AuStatus        *ret_status)
```

This routine creates the specified *filename*, retrieves the contents of the indicated *bucket*, and stores it in the given *sound_format*. If an error is encountered, `AuFalse` is returned and *ret_status* is set to an audio error code. Otherwise, `AuTrue` is returned.

```
AuPointer
AuSoundCreateDataFromBucket (
    AuServer        *audio,
    AuBucketID      bucket,
    SoundHeader     *header,
    char            **ret_comment,
    AuStatus        *ret_status)
```

This routine retrieves the contents of the specified *bucket*, sets the contents of *header*, sets *ret_comment* to point to a string that can be released using **`AuFree()`**, and returns the data. If an error is encountered, NULL is returned.

Listing Buckets

The following routine can be used to list buckets that have already been stored in the server:

```
AuBucketAttributes *
AuListBuckets (
    AuServer              *audio,
    AuMask                mask,
    AuBucketAttributes    *match_attr,
    int                   *ret_num_buckets,
    AuStatus              *ret_status)
```

This routine looks for all buckets which match the attributes indicated by *mask* stored in *match_attr*. A list of bucket attributes is returned, the count of which is stored in *ret_num_buckets*. If an error is encountered, NULL is returned and *ret_status* is set to an audio error code. The list of bucket attributes can be freed using the routine **`AuFreeBucketAttributes()`**.

Freeing Bucket Attributes

The following routine can be used to free a bucket attributes block:

```
void
AuFreeBucketAttributes (
    AuServer            *audio,
    int                 num_bucket_attrs,
    AuBucketAttributes  *bucket_attrs)
```

This routine releases the memory used by the list of bucket attributes stored in *bucket_attrs*.