miniaudio API Documentation

Version: 0.11.23 (2025-09-11) **Author:** David Reid (mackron@gmail.com) **Website:** https://miniaud.io
License: Public Domain or MIT-0

This document provides comprehensive API reference for implementing a Prolog wrapper for miniaudio. APIs are organized by implementation priority.

Table of Contents

Core Types and Constants

Version Information

```
#define MA_VERSION_MAJOR    0
#define MA_VERSION_MINOR    11
#define MA_VERSION_REVISION 23
#define MA_VERSION_STRING   "0.11.23"
```

Result Codes

```
typedef enum {
    MA_SUCCESS                      =  0,
    MA_ERROR                        = -1,
    MA_INVALID_ARGS                 = -2,
    MA_INVALID_OPERATION            = -3,
    MA_OUT_OF_MEMORY                = -4,
    MA_OUT_OF_RANGE                 = -5,
    MA_ACCESS_DENIED                = -6,
    MA_DOES_NOT_EXIST               = -7,
    MA_ALREADY_EXISTS               = -8,
    MA_TOO_MANY_OPEN_FILES          = -9,
    MA_INVALID_FILE                 = -10,
    MA_TOO_BIG                      = -11,
    MA_PATH_TOO_LONG                = -12,
    MA_NAME_TOO_LONG                = -13,
    MA_AT_END                       = -17,
```

```
    MA_NO_SPACE                         = -18,
    MA_BUSY                             = -19,
    MA_IO_ERROR                         = -20,
    MA_TIMEOUT                          = -34,

    /* miniaudio-specific errors */
    MA_FORMAT_NOT_SUPPORTED             = -200,
    MA_DEVICE_TYPE_NOT_SUPPORTED        = -201,
    MA_NO_BACKEND                       = -203,
    MA_NO_DEVICE                        = -204,
    MA_INVALID_DEVICE_CONFIG            = -206,
    MA_DEVICE_NOT_INITIALIZED           = -300,
    MA_DEVICE_ALREADY_INITIALIZED       = -301,
    MA_DEVICE_NOT_STARTED               = -302,
    MA_DEVICE_NOT_STOPPED               = -303
} ma_result;
```

Audio Formats

```
typedef enum {
    ma_format_unknown = 0,
    ma_format_u8      = 1,  // 8-bit unsigned integer [0, 255]
    ma_format_s16     = 2,  // 16-bit signed integer [-32768, 32767] (most
widely supported)
    ma_format_s24     = 3,  // 24-bit signed integer (tightly packed, 3
bytes per sample)
    ma_format_s32     = 4,  // 32-bit signed integer [-2147483648,
2147483647]
    ma_format_f32     = 5,  // 32-bit floating point [-1, 1]
    ma_format_count
} ma_format;
```

Channel Positions

```
typedef enum {
    MA_CHANNEL_NONE               = 0,
    MA_CHANNEL_MONO               = 1,
    MA_CHANNEL_FRONT_LEFT         = 2,
    MA_CHANNEL_FRONT_RIGHT        = 3,
    MA_CHANNEL_FRONT_CENTER       = 4,
    MA_CHANNEL_LFE                = 5,
    MA_CHANNEL_BACK_LEFT          = 6,
    MA_CHANNEL_BACK_RIGHT         = 7,
    MA_CHANNEL_FRONT_LEFT_CENTER  = 8,
    MA_CHANNEL_FRONT_RIGHT_CENTER = 9,
    MA_CHANNEL_BACK_CENTER        = 10,
    MA_CHANNEL_SIDE_LEFT          = 11,
    MA_CHANNEL_SIDE_RIGHT         = 12,
    MA_CHANNEL_LEFT               = MA_CHANNEL_FRONT_LEFT,
```

```
    MA_CHANNEL_RIGHT             = MA_CHANNEL_FRONT_RIGHT
} ma_channel;
```

Standard Sample Rates

```
typedef enum {
    ma_standard_sample_rate_8000   = 8000,
    ma_standard_sample_rate_11025  = 11025,
    ma_standard_sample_rate_16000  = 16000,
    ma_standard_sample_rate_22050  = 22050,
    ma_standard_sample_rate_24000  = 24000,
    ma_standard_sample_rate_32000  = 32000,
    ma_standard_sample_rate_44100  = 44100,   // CD quality
    ma_standard_sample_rate_48000  = 48000,   // Most common
    ma_standard_sample_rate_88200  = 88200,
    ma_standard_sample_rate_96000  = 96000,
    ma_standard_sample_rate_176400 = 176400,
    ma_standard_sample_rate_192000 = 192000,
    ma_standard_sample_rate_352800 = 352800,
    ma_standard_sample_rate_384000 = 384000
} ma_standard_sample_rate;
```

Device Types

```
typedef enum {
    ma_device_type_playback = 1,
    ma_device_type_capture  = 2,
    ma_device_type_duplex   = 3,  // playback | capture
    ma_device_type_loopback = 4
} ma_device_type;
```

Key Data Structures

ma_device_id

Identifies a physical audio device. Backend-specific union.

```
typedef union {
    ma_wchar_win32 wasapi[64];
    char alsa[256];
    char pulse[256];
    char coreaudio[256];
    ma_int32 aaudio;
    // ... other backend-specific fields
} ma_device_id;
```

ma_device_info

Basic information about an audio device.

```
typedef struct {
    ma_device_id id;
    char name[256];         // Device name (null-terminated)
    ma_bool32 isDefault;    // Whether this is the default device

    ma_uint32 nativeDataFormatCount;
    struct {
        ma_format format;
        ma_uint32 channels;
        ma_uint32 sampleRate;
        ma_uint32 flags;
    } nativeDataFormats[64];
} ma_device_info;
```

Priority 1: Core/Essential APIs

Version Functions

ma_version

```
void ma_version(ma_uint32* pMajor, ma_uint32* pMinor, ma_uint32*
pRevision);
```

Description: Retrieves the version number of miniaudio.

Parameters:

- `pMajor` (out): Pointer to receive major version number
- `pMinor` (out): Pointer to receive minor version number
- `pRevision` (out): Pointer to receive revision number

ma_version_string

```
const char* ma_version_string(void);
```

Description: Returns the version string (e.g., "0.11.23").

Returns: Version string pointer (do not free)

Context Management

The context represents the backend at a global level and is used for device enumeration and initialization.

ma_context_config_init

```
ma_context_config ma_context_config_init(void);
```

Description: Initializes a context config with default values.

Returns: Initialized context config structure

ma_context_init

```
ma_result ma_context_init(const ma_backend backends[],
                          ma_uint32 backendCount,
                          const ma_context_config* pConfig,
                          ma_context* pContext);
```

Description: Initializes a context for device enumeration and initialization.

Parameters:

- `backends` (in): Optional array of backend priorities. NULL uses defaults
- `backendCount` (in): Number of backends in array (0 if NULL)
- `pConfig` (in): Optional context configuration. NULL uses defaults
- `pContext` (out): Pointer to context structure to initialize

Returns: MA_SUCCESS on success, error code otherwise

Notes:

- Context must be uninitialized with `ma_context_uninit()`
- Backends tried in order until one succeeds
- Default backend order is platform-specific

ma_context_uninit

```
ma_result ma_context_uninit(ma_context* pContext);
```

Description: Uninitializes a context.

Parameters:

- `pContext` (in): Pointer to context to uninitialize

Returns: MA_SUCCESS on success

Notes: All devices must be uninitialized before calling this

ma_context_sizeof

```
size_t ma_context_sizeof(void);
```

Description: Returns the size in bytes of the ma_context structure.

Returns: Size in bytes

ma_context_get_log

```
ma_log* ma_context_get_log(ma_context* pContext);
```

Description: Retrieves a pointer to the log object associated with this context.

Returns: Pointer to log object

Device Enumeration

ma_context_get_devices

```
ma_result ma_context_get_devices(ma_context* pContext,
                                 ma_device_info** ppPlaybackDeviceInfos,
                                 ma_uint32* pPlaybackDeviceCount,
                                 ma_device_info** ppCaptureDeviceInfos,
                                 ma_uint32* pCaptureDeviceCount);
```

Description: Retrieves a list of available playback and capture devices.

Parameters:

- `pContext` (in): Pointer to context
- `ppPlaybackDeviceInfos` (out): Receives pointer to playback device info array (do not free)
- `pPlaybackDeviceCount` (out): Receives number of playback devices
- `ppCaptureDeviceInfos` (out): Receives pointer to capture device info array (do not free)
- `pCaptureDeviceCount` (out): Receives number of capture devices

Returns: MA_SUCCESS on success

Notes:

- Returned arrays are managed internally - do not free
- Arrays become invalid when context is uninitialized

ma_context_get_device_info

```
ma_result ma_context_get_device_info(ma_context* pContext,
                                     ma_device_type deviceType,
                                     const ma_device_id* pDeviceID,
                                     ma_device_info* pDeviceInfo);
```

Description: Retrieves detailed information about a specific device.

Parameters:

- `pContext` (in): Pointer to context
- `deviceType` (in): Type of device (playback or capture)
- `pDeviceID` (in): Device ID, or NULL for default device
- `pDeviceInfo` (out): Receives device information

Returns: MA_SUCCESS on success

ma_context_enumerate_devices

```
ma_result ma_context_enumerate_devices(ma_context* pContext,
                                       ma_enum_devices_callback_proc
callback,
                                       void* pUserData);
```

Description: Enumerates devices using a callback.

Parameters:

- `pContext` (in): Pointer to context
- `callback` (in): Callback function called for each device
- `pUserData` (in): User data passed to callback

Callback Signature:

```
typedef ma_bool32 (*ma_enum_devices_callback_proc)(ma_context* pContext,
                                                   ma_device_type
deviceType,
                                                   const ma_device_info*
pInfo,
                                                   void* pUserData);
```

Returns: MA_SUCCESS on success

Notes: Return MA_FALSE from callback to stop enumeration

ma_context_is_loopback_supported

```
ma_bool32 ma_context_is_loopback_supported(ma_context* pContext);
```

Description: Checks if the backend supports loopback devices.

Returns: MA_TRUE if supported, MA_FALSE otherwise

Priority 2: Basic I/O

Device Configuration

ma_device_config_init

```
ma_device_config ma_device_config_init(ma_device_type deviceType);
```

Description: Initializes a device config with default values.

Parameters:

- `deviceType` (in): Type of device (playback, capture, duplex, or loopback)

Returns: Initialized device config structure

Notes:

- Set `config.playback.format` (ma_format_f32, ma_format_s16, etc.)
- Set `config.playback.channels` (0 = device default)
- Set `config.sampleRate` (0 = device default, typically 44100 or 48000)
- Set `config.dataCallback` to your audio callback function
- Set `config.pUserData` for user data accessible in callback

ma_device_config Structure

```
struct ma_device_config {
    ma_device_type deviceType;
    ma_uint32 sampleRate;
    ma_uint32 periodSizeInFrames;
    ma_uint32 periodSizeInMilliseconds;
    ma_uint32 periods;
    ma_performance_profile performanceProfile;
    ma_device_data_proc dataCallback;
    ma_device_notification_proc notificationCallback;
    void* pUserData;

    struct {
```

```
        const ma_device_id* pDeviceID;
        ma_format format;
        ma_uint32 channels;
        ma_channel* pChannelMap;
        ma_share_mode shareMode;
    } playback;

    struct {
        const ma_device_id* pDeviceID;
        ma_format format;
        ma_uint32 channels;
        ma_channel* pChannelMap;
        ma_share_mode shareMode;
    } capture;

    // Backend-specific configs (wasapi, alsa, pulse, coreaudio, etc.)
};
```

Device Initialization and Control

ma_device_init

```
ma_result ma_device_init(ma_context* pContext,
                         const ma_device_config* pConfig,
                         ma_device* pDevice);
```

Description: Initializes an audio device.

Parameters:

- `pContext` (in): Pointer to context, or NULL to create internal context
- `pConfig` (in): Device configuration
- `pDevice` (out): Pointer to device structure to initialize

Returns: MA_SUCCESS on success

Notes:

- Device starts in stopped state - call `ma_device_start()` to begin
- Must call `ma_device_uninit()` when done
- If pContext is NULL, an internal context is created

ma_device_uninit

```
void ma_device_uninit(ma_device* pDevice);
```

Description: Uninitializes a device.

Parameters:

- `pDevice` (in): Pointer to device to uninitialize

Notes:

- Automatically stops device if started
- Frees all internal resources

ma_device_start

```
ma_result ma_device_start(ma_device* pDevice);
```

Description: Starts the device (begins audio thread and callbacks).

Parameters:

- `pDevice` (in): Pointer to device to start

Returns: MA_SUCCESS on success

Notes:

- Do not call from within data callback (will deadlock)
- Device must be initialized first

ma_device_stop

```
ma_result ma_device_stop(ma_device* pDevice);
```

Description: Stops the device (stops audio thread and callbacks).

Parameters:

- `pDevice` (in): Pointer to device to stop

Returns: MA_SUCCESS on success

Notes: Do not call from within data callback (will deadlock)

ma_device_is_started

```
ma_bool32 ma_device_is_started(const ma_device* pDevice);
```

Description: Checks if device is started.

Returns: MA_TRUE if started, MA_FALSE otherwise

ma_device_get_state

```
ma_device_state ma_device_get_state(const ma_device* pDevice);
```

Description: Gets the current device state.

Returns: Device state enum value

States:

```
typedef enum {
    ma_device_state_uninitialized = 0,
    ma_device_state_stopped       = 1,
    ma_device_state_started       = 2,
    ma_device_state_starting      = 3,
    ma_device_state_stopping      = 4
} ma_device_state;
```

Device Information

ma_device_get_info

```
ma_result ma_device_get_info(ma_device* pDevice,
                             ma_device_type type,
                             ma_device_info* pDeviceInfo);
```

Description: Gets device information.

Parameters:

- `pDevice` (in): Pointer to device
- `type` (in): Device type (playback or capture)
- `pDeviceInfo` (out): Receives device info

Returns: MA_SUCCESS on success

ma_device_get_name

```
ma_result ma_device_get_name(ma_device* pDevice,
                             ma_device_type type,
                             char* pName,
```

```
                                    size_t nameCap,
                                    size_t* pLengthNotIncludingNullTerminator);
```

Description: Gets the device name.

Parameters:

- `pDevice` (in): Pointer to device
- `type` (in): Device type
- `pName` (out): Buffer to receive name
- `nameCap` (in): Size of buffer
- `pLengthNotIncludingNullTerminator` (out): Actual length (optional, can be NULL)

Returns: MA_SUCCESS on success

ma_device_get_context

```
ma_context* ma_device_get_context(ma_device* pDevice);
```

Description: Gets the context associated with the device.

Returns: Pointer to context

ma_device_get_log

```
ma_log* ma_device_get_log(ma_device* pDevice);
```

Description: Gets the log object associated with the device.

Returns: Pointer to log object

Volume Control

ma_device_set_master_volume

```
ma_result ma_device_set_master_volume(ma_device* pDevice, float volume);
```

Description: Sets the master volume.

Parameters:

- `pDevice` (in): Pointer to device
- `volume` (in): Volume level (linear scale, 1.0 = 100%)

Returns: MA_SUCCESS on success

ma_device_get_master_volume

```
ma_result ma_device_get_master_volume(ma_device* pDevice, float* pVolume);
```

Description: Gets the master volume.

Parameters:

- `pDevice` (in): Pointer to device
- `pVolume` (out): Receives volume level

Returns: MA_SUCCESS on success

ma_device_set_master_volume_db

```
ma_result ma_device_set_master_volume_db(ma_device* pDevice, float
gainDB);
```

Description: Sets the master volume in decibels.

Parameters:

- `pDevice` (in): Pointer to device
- `gainDB` (in): Gain in decibels (0 = unity, negative = quieter, positive = louder)

Returns: MA_SUCCESS on success

ma_device_get_master_volume_db

```
ma_result ma_device_get_master_volume_db(ma_device* pDevice, float*
pGainDB);
```

Description: Gets the master volume in decibels.

Parameters:

- `pDevice` (in): Pointer to device
- `pGainDB` (out): Receives gain in decibels

Returns: MA_SUCCESS on success

Data Callback

The data callback is where audio data is read from or written to the device.

```
typedef void (*ma_device_data_proc)(ma_device* pDevice,
                                    void* pOutput,
                                    const void* pInput,
                                    ma_uint32 frameCount);
```

Callback Parameters:

- `pDevice` (in): Pointer to device
- `pOutput` (out): Output buffer (write audio here for playback)
- `pInput` (in): Input buffer (read audio from here for capture)
- `frameCount` (in): Number of frames to read/write

Important Notes:

- **Playback:** Write to `pOutput`, ignore `pInput` (NULL)
- **Capture:** Read from `pInput`, ignore `pOutput` (NULL)
- **Duplex:** Read from `pInput`, write to `pOutput`
- **Frame:** One sample per channel (stereo = 2 samples per frame)
- **Interleaved:** Multi-channel data is always interleaved (LRLRLR...)
- **Never call** `ma_device_init/uninit/start/stop` from callback

Priority 3: File Decoding

Decoder Configuration

ma_decoder_config_init

```
ma_decoder_config ma_decoder_config_init(ma_format outputFormat,
                                         ma_uint32 outputChannels,
                                         ma_uint32 outputSampleRate);
```

Description: Initializes decoder config.

Parameters:

- `outputFormat` (in): Desired output format (ma_format_unknown = native)
- `outputChannels` (in): Desired channel count (0 = native)
- `outputSampleRate` (in): Desired sample rate (0 = native)

Returns: Initialized decoder config

ma_decoder_config_init_default

```

```

```
ma_decoder_config ma_decoder_config_init_default(void);
```

Description: Initializes decoder config with all defaults (native format).

Returns: Initialized decoder config

Decoder Initialization

ma_decoder_init_file

```
ma_result ma_decoder_init_file(const char* pFilePath,
                               const ma_decoder_config* pConfig,
                               ma_decoder* pDecoder);
```

Description: Initializes a decoder from a file path.

Parameters:

- `pFilePath` (in): Path to audio file (WAV, MP3, FLAC, etc.)
- `pConfig` (in): Optional decoder config (NULL = defaults)
- `pDecoder` (out): Pointer to decoder structure

Returns: MA_SUCCESS on success

Supported Formats:

- WAV (always supported)
- MP3 (requires MA_ENABLE_MP3 or default build)
- FLAC (requires MA_ENABLE_FLAC or default build)
- Vorbis (requires MA_ENABLE_VORBIS)
- Opus (requires MA_ENABLE_OPUS)

ma_decoder_init_file_w

```
ma_result ma_decoder_init_file_w(const wchar_t* pFilePath,
                                 const ma_decoder_config* pConfig,
                                 ma_decoder* pDecoder);
```

Description: Initializes a decoder from a wide-character file path (Windows).

ma_decoder_init_memory

```
ma_result ma_decoder_init_memory(const void* pData,
                                 size_t dataSize,
```

```
                                    const ma_decoder_config* pConfig,
                                    ma_decoder* pDecoder);
```

Description: Initializes a decoder from memory buffer.

Parameters:

- `pData` (in): Pointer to encoded audio data
- `dataSize` (in): Size of data in bytes
- `pConfig` (in): Optional decoder config (NULL = defaults)
- `pDecoder` (out): Pointer to decoder structure

Returns: MA_SUCCESS on success

Notes: Buffer must remain valid for lifetime of decoder

ma_decoder_init_vfs

```
ma_result ma_decoder_init_vfs(ma_vfs* pVFS,
                              const char* pFilePath,
                              const ma_decoder_config* pConfig,
                              ma_decoder* pDecoder);
```

Description: Initializes a decoder using virtual file system.

Parameters:

- `pVFS` (in): Pointer to VFS object (NULL = default VFS)
- `pFilePath` (in): File path
- `pConfig` (in): Optional decoder config
- `pDecoder` (out): Pointer to decoder structure

Returns: MA_SUCCESS on success

ma_decoder_init

```
ma_result ma_decoder_init(ma_decoder_read_proc onRead,
                          ma_decoder_seek_proc onSeek,
                          void* pUserData,
                          const ma_decoder_config* pConfig,
                          ma_decoder* pDecoder);
```

Description: Initializes a decoder with custom read/seek callbacks.

Parameters:

- `onRead` (in): Read callback function

- `onSeek` (in): Seek callback function
- `pUserData` (in): User data passed to callbacks
- `pConfig` (in): Optional decoder config
- `pDecoder` (out): Pointer to decoder structure

Returns: MA_SUCCESS on success

Decoder Operations

ma_decoder_uninit

```
ma_result ma_decoder_uninit(ma_decoder* pDecoder);
```

Description: Uninitializes a decoder.

Parameters:

- `pDecoder` (in): Pointer to decoder

Returns: MA_SUCCESS on success

ma_decoder_read_pcm_frames

```
ma_result ma_decoder_read_pcm_frames(ma_decoder* pDecoder,
                                     void* pFramesOut,
                                     ma_uint64 frameCount,
                                     ma_uint64* pFramesRead);
```

Description: Reads PCM frames from decoder.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `pFramesOut` (out): Buffer to receive PCM frames (can be NULL to skip)
- `frameCount` (in): Number of frames to read
- `pFramesRead` (out): Receives actual number of frames read (optional)

Returns: MA_SUCCESS on success, MA_AT_END when end reached

Notes:

- Returns fewer frames than requested at end of file
- Data format matches decoder config output format

ma_decoder_seek_to_pcm_frame

```

```

```
ma_result ma_decoder_seek_to_pcm_frame(ma_decoder* pDecoder,
                                       ma_uint64 frameIndex);
```

Description: Seeks to a specific PCM frame.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `frameIndex` (in): Frame index to seek to (0-based)

Returns: MA_SUCCESS on success

Notes: Not all formats support seeking (Vorbis limited)

Decoder Information

ma_decoder_get_data_format

```
ma_result ma_decoder_get_data_format(ma_decoder* pDecoder,
                                     ma_format* pFormat,
                                     ma_uint32* pChannels,
                                     ma_uint32* pSampleRate,
                                     ma_channel* pChannelMap,
                                     size_t channelMapCap);
```

Description: Gets the output data format of the decoder.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `pFormat` (out): Receives sample format
- `pChannels` (out): Receives channel count
- `pSampleRate` (out): Receives sample rate
- `pChannelMap` (out): Receives channel map (optional)
- `channelMapCap` (in): Size of channel map array

Returns: MA_SUCCESS on success

ma_decoder_get_length_in_pcm_frames

```
ma_result ma_decoder_get_length_in_pcm_frames(ma_decoder* pDecoder,
                                              ma_uint64* pLength);
```

Description: Gets the total length in PCM frames.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `pLength` (out): Receives length in frames

Returns: MA_SUCCESS on success

Notes: May return 0 for streaming formats that don't know length

ma_decoder_get_cursor_in_pcm_frames

```
ma_result ma_decoder_get_cursor_in_pcm_frames(ma_decoder* pDecoder,
                                              ma_uint64* pCursor);
```

Description: Gets the current cursor position in PCM frames.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `pCursor` (out): Receives cursor position

Returns: MA_SUCCESS on success

ma_decoder_get_available_frames

```
ma_result ma_decoder_get_available_frames(ma_decoder* pDecoder,
                                          ma_uint64* pAvailableFrames);
```

Description: Gets number of frames available to read without blocking.

Parameters:

- `pDecoder` (in): Pointer to decoder
- `pAvailableFrames` (out): Receives available frame count

Returns: MA_SUCCESS on success

Convenience Decoding Functions

ma_decode_file

```
ma_result ma_decode_file(const char* pFilePath,
                         ma_decoder_config* pConfig,
                         ma_uint64* pFrameCountOut,
                         void** ppPCMFramesOut);
```

Description: Decodes entire file into memory.

Parameters:

- `pFilePath` (in): Path to file
- `pConfig` (in/out): Decoder config (NULL = defaults)
- `pFrameCountOut` (out): Receives total frame count
- `ppPCMFramesOut` (out): Receives pointer to allocated PCM data

Returns: MA_SUCCESS on success

Notes:

- Allocates memory - caller must free with `ma_free()`
- Format info is updated in pConfig on output

ma_decode_memory

```
ma_result ma_decode_memory(const void* pData,
                           size_t dataSize,
                           ma_decoder_config* pConfig,
                           ma_uint64* pFrameCountOut,
                           void** ppPCMFramesOut);
```

Description: Decodes entire buffer into memory.

Parameters:

- `pData` (in): Encoded audio data
- `dataSize` (in): Size of data
- `pConfig` (in/out): Decoder config
- `pFrameCountOut` (out): Receives frame count
- `ppPCMFramesOut` (out): Receives PCM data pointer

Returns: MA_SUCCESS on success

Priority 4: Data Sources

Data Source Interface

Data sources provide a unified interface for reading audio data from various sources (files, decoders, generators, etc.).

ma_data_source_read_pcm_frames

```
ma_result ma_data_source_read_pcm_frames(ma_data_source* pDataSource,
                                         void* pFramesOut,
                                         ma_uint64 frameCount,
                                         ma_uint64* pFramesRead);
```

Description: Reads PCM frames from a data source.

Parameters:

- `pDataSource` (in): Pointer to data source
- `pFramesOut` (out): Buffer to receive frames
- `frameCount` (in): Number of frames to read
- `pFramesRead` (out): Actual frames read (optional)

Returns: MA_SUCCESS on success

ma_data_source_seek_to_pcm_frame

```
ma_result ma_data_source_seek_to_pcm_frame(ma_data_source* pDataSource,
                                           ma_uint64 frameIndex);
```

Description: Seeks to a PCM frame in the data source.

Returns: MA_SUCCESS on success

ma_data_source_get_data_format

```
ma_result ma_data_source_get_data_format(ma_data_source* pDataSource,
                                         ma_format* pFormat,
                                         ma_uint32* pChannels,
                                         ma_uint32* pSampleRate,
                                         ma_channel* pChannelMap,
                                         size_t channelMapCap);
```

Description: Gets data format of the data source.

ma_data_source_get_length_in_pcm_frames

```
ma_result ma_data_source_get_length_in_pcm_frames(ma_data_source*
pDataSource,
                                                  ma_uint64* pLength);
```

Description: Gets the total length in PCM frames.

ma_data_source_get_cursor_in_pcm_frames

```
ma_result ma_data_source_get_cursor_in_pcm_frames(ma_data_source*
pDataSource,
                                                  ma_uint64* pCursor);
```

Description: Gets current cursor position.

ma_data_source_set_looping

```
ma_result ma_data_source_set_looping(ma_data_source* pDataSource,
                                     ma_bool32 isLooping);
```

Description: Enables or disables looping.

ma_data_source_is_looping

```
ma_bool32 ma_data_source_is_looping(const ma_data_source* pDataSource);
```

Description: Checks if looping is enabled.

Audio Buffer Data Source

ma_audio_buffer_config

```
typedef struct {
    ma_format format;
    ma_uint32 channels;
    ma_uint32 sampleRate;
    ma_uint64 sizeInFrames;
    const void* pData;
    ma_allocation_callbacks allocationCallbacks;
} ma_audio_buffer_config;
```

ma_audio_buffer_init

```
ma_result ma_audio_buffer_init(const ma_audio_buffer_config* pConfig,
                               ma_audio_buffer* pAudioBuffer);
```

Description: Initializes an audio buffer data source from existing PCM data.

ma_audio_buffer_uninit

```
void ma_audio_buffer_uninit(ma_audio_buffer* pAudioBuffer);
```

Priority 5: Advanced Features

High-Level Engine API

The engine provides the easiest way to play sounds with 3D spatialization, mixing, and resource management.

ma_engine_config_init

```
ma_engine_config ma_engine_config_init(void);
```

Description: Initializes engine config with defaults.

Returns: Initialized engine config

ma_engine_init

```
ma_result ma_engine_init(const ma_engine_config* pConfig,
                         ma_engine* pEngine);
```

Description: Initializes the high-level engine.

Parameters:

- `pConfig` (in): Engine configuration (NULL = defaults)
- `pEngine` (out): Pointer to engine structure

Returns: MA_SUCCESS on success

Notes:

- Automatically initializes device, resource manager, and node graph
- Engine starts automatically

ma_engine_uninit

```
void ma_engine_uninit(ma_engine* pEngine);
```

Description: Uninitializes the engine.

Playing Sounds with Engine

ma_engine_play_sound

```
ma_result ma_engine_play_sound(ma_engine* pEngine,
                               const char* pFilePath,
                               ma_sound_group* pGroup);
```

Description: Plays a sound file ("fire and forget").

Parameters:

- `pEngine` (in): Pointer to engine
- `pFilePath` (in): Path to sound file
- `pGroup` (in): Sound group (NULL = no group)

Returns: MA_SUCCESS on success

Notes:

- Sound plays once and is automatically recycled
- No handle returned - cannot control after starting

Sound Objects

For more control, initialize ma_sound objects.

ma_sound_init_from_file

```
ma_result ma_sound_init_from_file(ma_engine* pEngine,
                                  const char* pFilePath,
                                  ma_uint32 flags,
                                  ma_sound_group* pGroup,
                                  ma_fence* pDoneFence,
                                  ma_sound* pSound);
```

Description: Initializes a sound from a file.

Parameters:

- `pEngine` (in): Pointer to engine
- `pFilePath` (in): Path to sound file
- `flags` (in): MA_SOUND_FLAG_* flags
- `pGroup` (in): Sound group (optional)
- `pDoneFence` (in): Fence for async loading (optional)
- `pSound` (out): Pointer to sound structure

Flags:

```
#define MA_SOUND_FLAG_STREAM              0x00000001  // Stream from disk
(don't load fully)
```

```
#define MA_SOUND_FLAG_DECODE              0x00000002  // Decode upfront
#define MA_SOUND_FLAG_ASYNC               0x00000004  // Load
asynchronously
#define MA_SOUND_FLAG_WAIT_INIT           0x00000008  // Wait for init
before returning
#define MA_SOUND_FLAG_NO_DEFAULT_ATTACHMENT 0x00001000  // Don't attach to
endpoint
#define MA_SOUND_FLAG_NO_PITCH            0x00002000  // Disable pitch
shifting
#define MA_SOUND_FLAG_NO_SPATIALIZATION   0x00004000  // Disable 3D
spatialization
```

Returns: MA_SUCCESS on success

ma_sound_uninit

```
void ma_sound_uninit(ma_sound* pSound);
```

Description: Uninitializes a sound.

Sound Playback Control

ma_sound_start

```
ma_result ma_sound_start(ma_sound* pSound);
```

Description: Starts playing the sound.

ma_sound_stop

```
ma_result ma_sound_stop(ma_sound* pSound);
```

Description: Stops playing the sound.

ma_sound_is_playing

```
ma_bool32 ma_sound_is_playing(const ma_sound* pSound);
```

Description: Checks if sound is currently playing.

ma_sound_at_end

```
ma_bool32 ma_sound_at_end(const ma_sound* pSound);
```

Description: Checks if sound has reached the end.

ma_sound_set_looping

```
void ma_sound_set_looping(ma_sound* pSound, ma_bool32 isLooping);
```

Description: Enables or disables looping.

ma_sound_is_looping

```
ma_bool32 ma_sound_is_looping(const ma_sound* pSound);
```

Sound Properties

ma_sound_set_volume

```
void ma_sound_set_volume(ma_sound* pSound, float volume);
```

Description: Sets the volume (1.0 = 100%).

ma_sound_get_volume

```
float ma_sound_get_volume(const ma_sound* pSound);
```

ma_sound_set_pan

```
void ma_sound_set_pan(ma_sound* pSound, float pan);
```

Description: Sets stereo panning (-1 = left, 0 = center, +1 = right).

ma_sound_get_pan

```
float ma_sound_get_pan(const ma_sound* pSound);
```

ma_sound_set_pitch

```
void ma_sound_set_pitch(ma_sound* pSound, float pitch);
```

Description: Sets pitch (1.0 = normal, 2.0 = double speed/octave up).

ma_sound_get_pitch

```
float ma_sound_get_pitch(const ma_sound* pSound);
```

3D Spatialization

ma_sound_set_position

```
void ma_sound_set_position(ma_sound* pSound, float x, float y, float z);
```

Description: Sets 3D position of sound.

ma_sound_get_position

```
ma_vec3f ma_sound_get_position(const ma_sound* pSound);
```

ma_sound_set_direction

```
void ma_sound_set_direction(ma_sound* pSound, float x, float y, float z);
```

Description: Sets direction vector for directional sounds.

ma_sound_get_direction

```
ma_vec3f ma_sound_get_direction(const ma_sound* pSound);
```

ma_sound_set_velocity

```
void ma_sound_set_velocity(ma_sound* pSound, float x, float y, float z);
```

Description: Sets velocity for doppler effect.

ma_sound_get_velocity

```
ma_vec3f ma_sound_get_velocity(const ma_sound* pSound);
```

ma_sound_set_attenuation_model

```
void ma_sound_set_attenuation_model(ma_sound* pSound,
                                    ma_attenuation_model
attenuationModel);
```

Description: Sets distance attenuation model.

Models:

```
typedef enum {
    ma_attenuation_model_none,
    ma_attenuation_model_inverse,
    ma_attenuation_model_linear,
    ma_attenuation_model_exponential
} ma_attenuation_model;
```

ma_sound_set_min_distance

```
void ma_sound_set_min_distance(ma_sound* pSound, float minDistance);
```

Description: Sets minimum distance (no attenuation within this).

ma_sound_set_max_distance

```
void ma_sound_set_max_distance(ma_sound* pSound, float maxDistance);
```

Description: Sets maximum distance (maximum attenuation beyond this).

ma_sound_set_cone

```
void ma_sound_set_cone(ma_sound* pSound,
                       float innerAngleInRadians,
```

```
                          float outerAngleInRadians,
                          float outerGain);
```

Description: Sets directional cone for the sound.

Engine Listener Control

ma_engine_listener_set_position

```
void ma_engine_listener_set_position(ma_engine* pEngine,
                                     ma_uint32 listenerIndex,
                                     float x, float y, float z);
```

Description: Sets 3D position of listener.

ma_engine_listener_get_position

```
ma_vec3f ma_engine_listener_get_position(const ma_engine* pEngine,
                                         ma_uint32 listenerIndex);
```

ma_engine_listener_set_direction

```
void ma_engine_listener_set_direction(ma_engine* pEngine,
                                      ma_uint32 listenerIndex,
                                      float x, float y, float z);
```

ma_engine_listener_set_velocity

```
void ma_engine_listener_set_velocity(ma_engine* pEngine,
                                     ma_uint32 listenerIndex,
                                     float x, float y, float z);
```

ma_engine_listener_set_cone

```
void ma_engine_listener_set_cone(ma_engine* pEngine,
                                 ma_uint32 listenerIndex,
                                 float innerAngleInRadians,
                                 float outerAngleInRadians,
                                 float outerGain);
```

Seeking and Timing

ma_sound_seek_to_pcm_frame

```
ma_result ma_sound_seek_to_pcm_frame(ma_sound* pSound, ma_uint64
frameIndex);
```

Description: Seeks to a specific frame.

ma_sound_get_time_in_pcm_frames

```
ma_uint64 ma_sound_get_time_in_pcm_frames(const ma_sound* pSound);
```

Description: Gets current playback position in frames.

ma_sound_get_length_in_pcm_frames

```
ma_result ma_sound_get_length_in_pcm_frames(const ma_sound* pSound,
                                            ma_uint64* pLength);
```

Description: Gets total length in frames.

ma_engine_get_time_in_pcm_frames

```
ma_uint64 ma_engine_get_time_in_pcm_frames(const ma_engine* pEngine);
```

Description: Gets engine's global time (for scheduling).

ma_engine_set_time_in_pcm_frames

```
ma_result ma_engine_set_time_in_pcm_frames(ma_engine* pEngine,
                                           ma_uint64 globalTime);
```

Description: Sets engine's global time.

Scheduled Start/Stop

ma_sound_set_start_time_in_pcm_frames

```
void ma_sound_set_start_time_in_pcm_frames(ma_sound* pSound,
                                           ma_uint64
absoluteGlobalTimeInFrames);
```

Description: Schedules sound to start at specific time.

ma_sound_set_stop_time_in_pcm_frames

```
void ma_sound_set_stop_time_in_pcm_frames(ma_sound* pSound,
                                          ma_uint64
absoluteGlobalTimeInFrames);
```

Description: Schedules sound to stop at specific time.

Priority 6: Resource Management

Resource Manager

The resource manager handles loading and caching of audio files, with support for streaming and reference counting.

ma_resource_manager_config_init

```
ma_resource_manager_config ma_resource_manager_config_init(void);
```

Description: Initializes resource manager config.

ma_resource_manager_init

```
ma_result ma_resource_manager_init(const ma_resource_manager_config*
pConfig,
                                   ma_resource_manager* pResourceManager);
```

Description: Initializes a resource manager.

Returns: MA_SUCCESS on success

ma_resource_manager_uninit

```
void ma_resource_manager_uninit(ma_resource_manager* pResourceManager);
```

Description: Uninitializes the resource manager.

Data Buffers (Loaded into Memory)

ma_resource_manager_data_buffer_init

```
ma_result ma_resource_manager_data_buffer_init(ma_resource_manager*
pResourceManager,
                                               const char* pFilePath,
                                               ma_uint32 flags,
                                               const
ma_resource_manager_pipeline_notifications* pNotifications,

ma_resource_manager_data_buffer* pDataBuffer);
```

Description: Loads an audio file fully into memory.

Parameters:

- `pResourceManager` (in): Resource manager
- `pFilePath` (in): Path to file
- `flags` (in): Loading flags
- `pNotifications` (in): Optional notifications (for async)
- `pDataBuffer` (out): Data buffer object

Flags:

```
#define MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_STREAM     0x00000001
#define MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_DECODE     0x00000002
#define MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_ASYNC      0x00000004
#define MA_RESOURCE_MANAGER_DATA_SOURCE_FLAG_WAIT_INIT  0x00000008
```

Returns: MA_SUCCESS on success

ma_resource_manager_data_buffer_uninit

```
ma_result
ma_resource_manager_data_buffer_uninit(ma_resource_manager_data_buffer*
pDataBuffer);
```

Data Streams (Streamed from Disk)

ma_resource_manager_data_stream_init

```
ma_result ma_resource_manager_data_stream_init(ma_resource_manager*
pResourceManager,
                                               const char* pFilePath,
                                               ma_uint32 flags,
                                               const
ma_resource_manager_pipeline_notifications* pNotifications,

ma_resource_manager_data_stream* pDataStream);
```

Description: Opens an audio file for streaming.

ma_resource_manager_data_stream_uninit

```
ma_result
ma_resource_manager_data_stream_uninit(ma_resource_manager_data_stream*
pDataStream);
```

Encoding (Writing Audio Files)

ma_encoder_config_init

```
ma_encoder_config ma_encoder_config_init(ma_encoding_format
encodingFormat,
                                         ma_format format,
                                         ma_uint32 channels,
                                         ma_uint32 sampleRate);
```

Description: Initializes encoder config.

Formats:

```
typedef enum {
    ma_encoding_format_unknown = 0,
    ma_encoding_format_wav,
    ma_encoding_format_flac,
    ma_encoding_format_mp3,
    ma_encoding_format_vorbis
} ma_encoding_format;
```

ma_encoder_init_file

```
ma_result ma_encoder_init_file(const char* pFilePath,
                               const ma_encoder_config* pConfig,
                               ma_encoder* pEncoder);
```

Description: Initializes an encoder to write to a file.

Returns: MA_SUCCESS on success

ma_encoder_uninit

```
void ma_encoder_uninit(ma_encoder* pEncoder);
```

ma_encoder_write_pcm_frames

```
ma_result ma_encoder_write_pcm_frames(ma_encoder* pEncoder,
                                      const void* pFramesIn,
                                      ma_uint64 frameCount,
                                      ma_uint64* pFramesWritten);
```

Description: Writes PCM frames to the encoder.

Additional Utility Functions

Format Conversion

ma_get_bytes_per_sample

```
ma_uint32 ma_get_bytes_per_sample(ma_format format);
```

Description: Returns bytes per sample for a format.

ma_get_bytes_per_frame

```
ma_uint32 ma_get_bytes_per_frame(ma_format format, ma_uint32 channels);
```

Description: Returns bytes per frame (sample_size * channels).

ma_volume_linear_to_db

```
```

```
float ma_volume_linear_to_db(float factor);
```

Description: Converts linear volume to decibels.

ma_volume_db_to_linear

```
float ma_volume_db_to_linear(float gain);
```

Description: Converts decibels to linear volume.

Channel Mapping

ma_channel_map_init_standard

```
void ma_channel_map_init_standard(ma_standard_channel_map
standardChannelMap,
                                  ma_channel* pChannelMap,
                                  size_t channelMapCap,
                                  ma_uint32 channels);
```

Description: Initializes a standard channel map.

ma_channel_map_is_valid

```
ma_bool32 ma_channel_map_is_valid(const ma_channel* pChannelMap,
                                  ma_uint32 channels);
```

ma_channel_map_is_equal

```
ma_bool32 ma_channel_map_is_equal(const ma_channel* pChannelMapA,
                                  const ma_channel* pChannelMapB,
                                  ma_uint32 channels);
```

ma_channel_map_is_blank

```
ma_bool32 ma_channel_map_is_blank(const ma_channel* pChannelMap,
                                  ma_uint32 channels);
```

Description: Checks if channel map is all MA_CHANNEL_NONE.

DSP Effects

Filters

miniaudio includes several built-in filters:

- **Biquad Filter:** Generic 2-pole/2-zero IIR filter
- **Low-pass Filter (LPF):** 1st, 2nd order, and higher order
- **High-pass Filter (HPF):** 1st, 2nd order, and higher order
- **Band-pass Filter (BPF):** 2nd order and higher order
- **Notch Filter:** 2nd order
- **Peaking EQ:** 2nd order
- **Low Shelf:** 2nd order
- **High Shelf:** 2nd order

All filters follow the same pattern:

1. Initialize config with `ma_XXX_config_init()`
2. Initialize filter with `ma_XXX_init()`
3. Process audio with `ma_XXX_process_pcm_frames()`
4. Uninitialize with `ma_XXX_uninit()`

Memory Management

Allocation Callbacks

```
typedef struct {
    void* pUserData;
    void* (*onMalloc)(size_t sz, void* pUserData);
    void* (*onRealloc)(void* p, size_t sz, void* pUserData);
    void (*onFree)(void* p, void* pUserData);
} ma_allocation_callbacks;
```

Most initialization functions accept optional allocation callbacks for custom memory management.

Thread Safety Notes

- **Context:** Thread-safe for enumeration and device info queries
- **Device:** Not thread-safe. Don't call init/uninit/start/stop from callback
- **Decoder:** Not thread-safe. Use one decoder per thread or add your own synchronization
- **Engine:** Thread-safe for most operations
- **Sound:** Thread-safe for property changes during playback

---

Backend Support

miniaudio supports multiple backends (automatically selected by priority):

- **Windows:** WASAPI, DirectSound, WinMM
- **macOS/iOS:** Core Audio
- **Linux:** ALSA, PulseAudio, JACK
- **BSD:** OSS, audio(4), sndio
- **Android:** AAudio, OpenSL|ES
- **Web:** Web Audio (Emscripten)
- **Custom:** Implement your own backend

Implementation Notes for Prolog Wrapper

Priority 1 (Essential - Implement First)

1. Version functions (`ma_version`, `ma_version_string`)
2. Context management (`ma_context_init`, `ma_context_uninit`)
3. Device enumeration (`ma_context_get_devices`, `ma_context_get_device_info`)

Priority 2 (Basic I/O - Core Functionality)

1. Device config and initialization (`ma_device_config_init`, `ma_device_init`)
2. Device control (`ma_device_start`, `ma_device_stop`, `ma_device_uninit`)
3. Data callback integration (critical - requires FFI callback support)
4. Volume control (`ma_device_set_master_volume`)

Priority 3 (File Decoding - Essential for File Playback)

1. Decoder initialization (`ma_decoder_init_file`, `ma_decoder_init_memory`)
2. Reading frames (`ma_decoder_read_pcm_frames`)
3. Seeking (`ma_decoder_seek_to_pcm_frame`)
4. Format queries (`ma_decoder_get_data_format`, `ma_decoder_get_length_in_pcm_frames`)

Priority 4 (Data Sources - Unified Interface)

1. Data source interface functions (work with decoders, buffers, etc.)
2. Audio buffer initialization
3. Looping support

Priority 5 (Advanced - High-Level API)

1. Engine initialization (`ma_engine_init`, `ma_engine_uninit`)
2. Simple sound playback (`ma_engine_play_sound`)
3. Sound objects (`ma_sound_init_from_file`, `ma_sound_start`, `ma_sound_stop`)
4. Sound properties (volume, pitch, pan)
5. 3D spatialization (positions, directions, attenuation)

Priority 6 (Resource Management - Optimization)

1. Resource manager (optional but useful for games)
2. Data buffers and streams
3. Encoding support (if writing audio files needed)

Example Usage Patterns

Basic Playback Device

```
ma_context context;
ma_context_init(NULL, 0, NULL, &context);

ma_device_config config = ma_device_config_init(ma_device_type_playback);
config.playback.format = ma_format_f32;
config.playback.channels = 2;
config.sampleRate = 48000;
config.dataCallback = data_callback;

ma_device device;
ma_device_init(&context, &config, &device);
ma_device_start(&device);

// ... let it play ...

ma_device_uninit(&device);
ma_context_uninit(&context);
```

Decode and Play File

```
ma_decoder decoder;
ma_decoder_init_file("sound.mp3", NULL, &decoder);

// Read frames in loop
ma_uint64 framesRead;
float pcmFrames[4096];
while (ma_decoder_read_pcm_frames(&decoder, pcmFrames, 2048, &framesRead)
== MA_SUCCESS && framesRead > 0) {
    // Output pcmFrames to device or process
}

ma_decoder_uninit(&decoder);
```

High-Level Engine

```
ma_engine engine;
ma_engine_init(NULL, &engine);
```

```c
// Simple fire-and-forget
ma_engine_play_sound(&engine, "explosion.wav", NULL);

// Or with control
ma_sound sound;
ma_sound_init_from_file(&engine, "music.mp3", 0, NULL, NULL, &sound);
ma_sound_set_volume(&sound, 0.5f);
ma_sound_set_looping(&sound, MA_TRUE);
ma_sound_start(&sound);

// ... later ...
ma_sound_uninit(&sound);
ma_engine_uninit(&engine);
```

References

- **Official Documentation:** https://miniaud.io/docs
- **GitHub Repository:** https://github.com/mackron/miniaudio
- **License:** Public Domain or MIT-0 (choose either)

This documentation covers the essential APIs needed for a Prolog wrapper implementation. For complete details on all 2300+ functions, refer to the inline documentation in miniaudio.h.