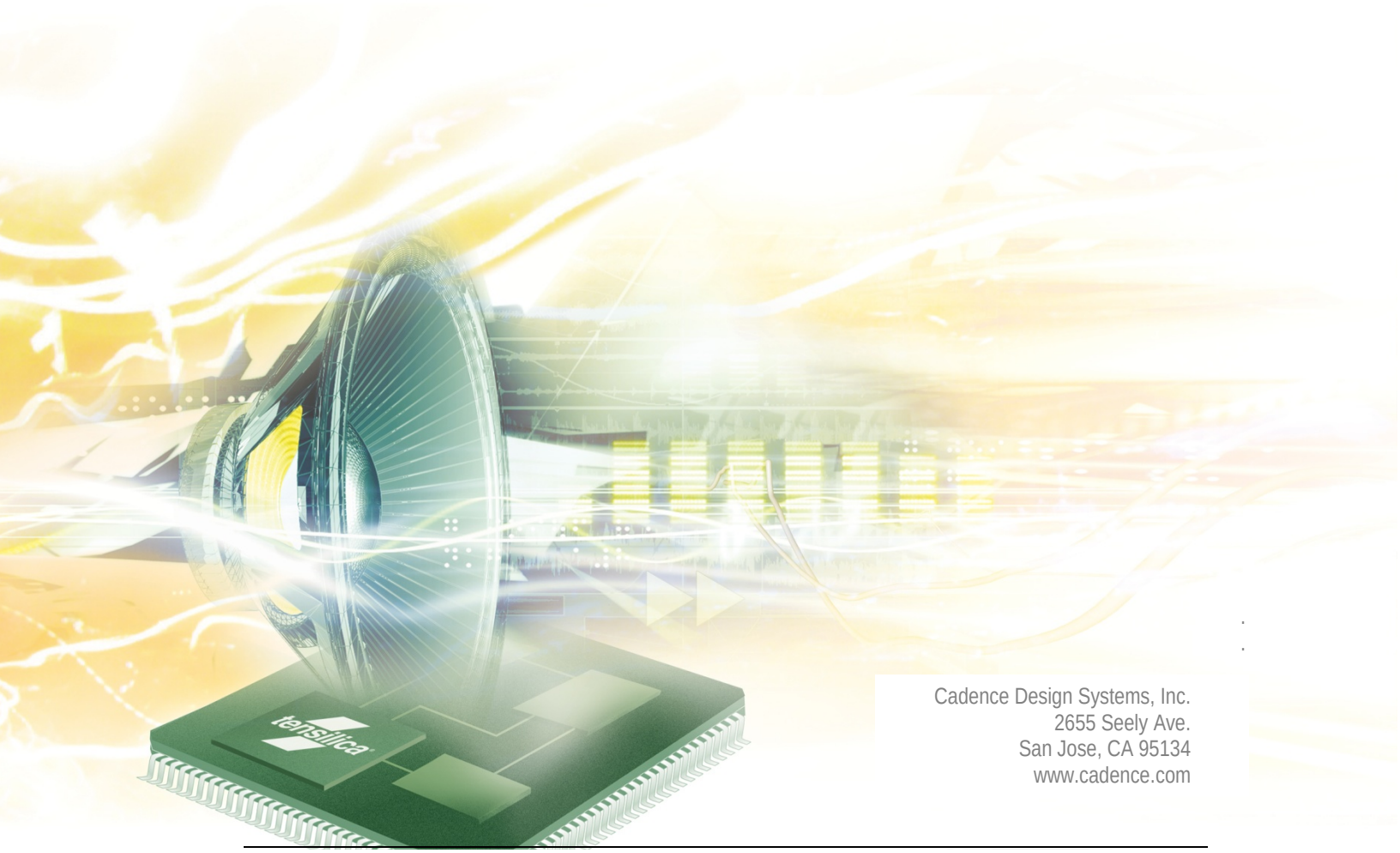
cadence®

# HiFi Speech Codec

## Application Programming Interface (API) Definition

For Xtensa HiFi Audio Engines




Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Version 1.0

May 2016

PD-16-8558-10-00

# Contents

# Figures



# Tables



# Document Change History

| Version | Changes |
|---|---|
| 1.0 | ■ Initial release. |

# 1. Introduction to the HiFi Speech Codec API

The HiFi Speech Codec Application Programming Interface (API) is a light-weight C-callable API that is exposed by all the HiFi-based Speech Codecs developed by Cadence. A “speech codec” is a generic term for any audio processing component and is not restricted to speech encoders and decoders. The speech codec is created using the Xtensa® Software Development Toolkit [1] and is targeted to a specific HiFi core [2].

A more complex version of the API called the “HiFi Audio Codec API” [3] is used for larger components (for example, complex audio codec) that require additional functionality.

The API has gone through several revisions; this document covers the latest revision, that is, Revision 1.1.

## *Document Overview*

The HiFi codec libraries implement a simple API to encapsulate the complexities of the coding operations and simplify the application and system implementation. Parts of the API are common to all the HiFi codecs, these are described in Section 2 after the introduction. Section 3 covers optional additional features that may be implemented by a particular HiFi codec.

# 2. Generic HiFi Speech Codec API

This section describes the API, which is common to all the HiFi speech processing libraries. The API facilitates any component that works in the overall method shown in the following diagram.

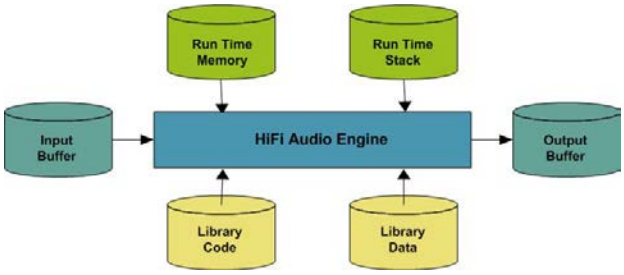



**Figure 1 HiFi Audio Processing Component Interfaces**

Section 2.1 discusses all the types of run-time memory required by the components. There is no state information held in static memory, therefore a single thread can perform time division processing of multiple components. Additionally, multiple threads can perform concurrent component processing.

## *Memory Management*

The HiFi audio processing API supports a flexible memory scheme and a simple interface that eases the integration into the final application. The API allows the components to request the required memory for their operations during run time.

The run time memory requirement consists primarily of the scratch and persistent memory. The components also require an input buffer and output buffer for the passing of data into and out of the component.

### 2.1.1 API Handle / Persistent Memory

The component API stores persistent state information in a structure that is referenced via an opaque handle. The handle is passed by the application for each API call. This object contains all state and history information that is maintained from one component frame invocation to the next within the same thread or instance. The components expect that the contents of the persistent memory be unchanged by the system apart from the component library itself for the complete lifetime of the component operation.

### 2.1.2 Scratch Memory

This is the temporary buffer used by the component during a single frame processing call. The contents of this memory region should not be changed if the actual component execution process is active, that is, if the thread running the component is inside any API call. This region can be used freely by the system between successive calls to the component.

### 2.1.3 Input Buffer

This is the buffer used by the algorithm for accepting input data. Before the call to the component, the input buffer needs to be completely filled with input data.

### 2.1.4 Output Buffer

This is the buffer in which the algorithm writes the output. This buffer needs to be made available for the component before its execution call. The output buffer pointer can be changed by the application between calls to the component. This allows the component to write directly to the required output area.

## *C Language API*

An overview of the component flow is shown in Figure 2. The audio processing component API consists of query, initialization, and execution functions.

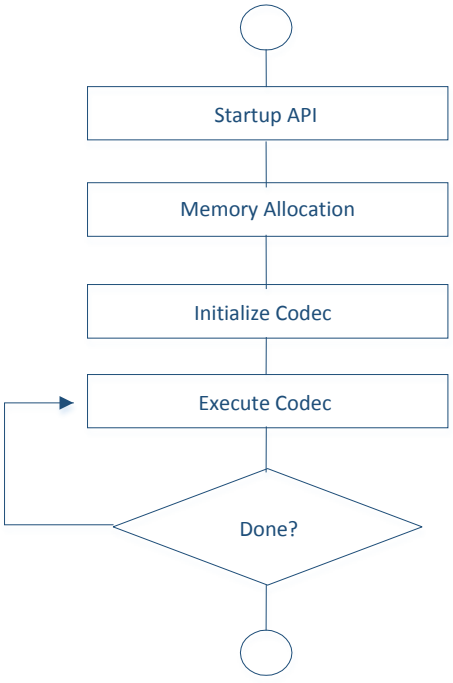



Figure 2 Audio Processing Component Flow Overview

### 2.1.5 Query Functions: xa_<codec>_get_<data>

The query functions are used in the startup and the memory allocation component stages to obtain information about the version and the memory requirements of the component library.

### 2.1.6 Initialization Functions: xa_<codec>_init

The initialization functions are used to reset the component to its initial state. Because the component library is fully reentrant, a process can initialize the component library multiple times and multiple processes can initialize the same component library as appropriate.

### 2.1.7 Execution Functions: xa_<codec>_process

The execution functions are used to process the audio frames.

The audio component sequence, as well as the functions associated with each stage, is described in detail in Section 3. Setting or querying component parameters is not shown in the figure below. This can happen any time after the component is initialized.

## *Generic API Errors*

Audio Processing API functions return an error code of type XA_ERRORCODE, which is of type `signed int`. The format of the error codes are defined in the following table.

| 31 | 30-15 | 14 – 11 | 10 - 6 | 5 - 0 |
|---|---|---|---|---|
| Fatal | Reserved | Class | Component | Sub code |

The errors that can be returned from the API are subdivided into those that are fatal, which require the restarting of the whole component and those that are nonfatal and are provided for information to the application.

The class of an error can be API, Config, or Execution. The API category errors are concerned with the incorrect use of the API. The Config errors are produced when the component parameters are incorrect or outside the supported usage. The Execution errors are returned after a call to the main encoding or decoding process and indicate situations that have arisen due to the input data.

# Common API Errors

All these errors are fatal and should not be encountered during a normal application operation. They signal that a serious error has occurred in the application that is calling the component.

- XA_API_FATAL_MEM_ALLOC
  - At least one of the pointers passed into the API function is NULL
- XA_API_FATAL_MEM_ALIGN
  - At least one of the pointers passed into the API function is not properly aligned

# Files Describing the API

The common include files (`include`) are:

- `xa_error_standards.h`
  - The macros and definitions for all the generic errors
- `xa_type_def.h`
  - All the types required for the API calls

# 3. HiFi Speech Codec API Specifics

A HiFi Speech Codec must conform to the generic codec API. However, it can have optional codec-specific additions.

Section 3.1 shows the files and details of API calls that may be specific to a particular codec. Section 3.2 describes codec-specific error codes. Configuration parameters, usage notes, and codec specific commands are described in Section 3.3.

## 3.1 Codec Specific Files

The codec API is required to be delivered in the form of a single header file (typically called `xa_<codec>_api.h`) and a single library file (typically called `xa_<codec>.a`). The library is built using a specific version of the Xtensa tools on a specific core.

## 3.2 Codec Specific Error Codes

Other than common error codes explained in Section 2, the codec may also report error codes specific to itself. These could be fatal or non-fatal errors.

## 3.3 API Functions

The codec API functions relevant to each stage in the component flow are specified in the following sections.

### 3.3.1 Startup Stage

The API startup functions described below get the various identification strings from the component library. They are for information only and their usage is optional. These functions do not take any input arguments and return `const char *`.

Table 3-1 Library Identification Functions

| Function | Description |
|---|---|
| `xa_<codec>_get_lib_name_string` | Gets the name of the library |
| `xa_<codec>_get_lib_version_string` | Gets the version of the library |
| `xa_<codec>_get_lib_api_version_string` | Gets the version of the API |

#### Example

For a hypothetical codec called SIMPLE_PROC:

```
const char *name = xa_simple_proc_get_lib_name_string();
const char *ver = xa_simple_proc_get_lib_version_string();
const char *aver = xa_simple_proc_get_lib_api_version_string();
```

#### Errors

- None

## 3.3.2 Memory Allocation Stage

During the memory allocation stage, the application needs to reserve the necessary memory for the HiFi codec library handles (persistent state) and scratch buffers. The required alignment of the handles and the scratch buffers is eight bytes. The application can use the functions listed in Table 3-2 to query the library for the required size of each buffer. The functions take a pointer of type `xa_<codec>_init_cfg_t` and return `WORD32`.

`xa_<codec>_init_cfg_t` is a structure that contains the initialization parameters for this instance of the library. The default initial configuration parameters are used if `NULL` is passed, instead of a valid pointer.

While input and output frame buffers are required for the operation of the component, they do not need to be reserved at this stage. Pointers to the frame buffers are passed in each invocation of the main component execution function.

The size and alignment requirements of the I/O buffers are specified in Section 3.3.4.

Table 3-2 Memory Management Functions

| Function | Description |
|---|---|
| `xa_<codec>_get_handle_byte_size` | Returns the size of the HiFi Codec API handle (persistent state) in bytes |
| `xa_<codec>_get_scratch_byte_size` | Returns the size of the HiFi Codec scratch memory |

### Example

For a hypothetical codec called SIMPLE_PROC, an example for default configuration parameters:

```
WORD32 handle_size;
WORD32 scratch_size;
handle_size = xa_simple_proc_get_handle_byte_size(NULL);
scratch_size = xa_simple_proc_get_scratch_byte_size(NULL);
```

### Errors

- Codec specific error, if the configuration parameters (passed in `p_cfg`) are not valid.

### 3.3.3 Initialization Stage

In the initialization stage, the application points the HiFi codec component to its API handle and scratch buffer. The application also specifies various other parameters related to the operation of the component and places the component in its initial state. The API functions for the HiFi codec component initialization are specified in Table 3-3.

Table 3-3 HiFi Codec Initialization Function Details

| | |
|---|---|
| Function | `xa_<codec>_init` |
| Syntax | `XA_ERRORCODE`<br>`xa_<codec>_init (`<br>`xa_codec_handle_t handle,`<br>`pWORD32 scratch,`<br>`xa_<codec>_init_cfg_t *p_cfg)` |
| Description | Resets the HiFi codec API handle into its initial state. Sets up the component to run using the supplied scratch buffer and the specified initial configuration parameters. |
| Parameters | **Input:** `handle`<br>Pointer to the component persistent memory. This is the opaque handle<br>Required size: See `xa_<codec>_get_handle_byte_size`<br>Required alignment: 8 bytes<br>**Input:** `scratch`<br>Pointer to the component scratch buffer<br>Required size: See `xa_<codec>_get_scratch_byte_size`<br>Required alignment: 8 bytes<br>**Input:** `p_cfg`<br>Initial configuration parameters (see Section 3.3.2). Note that the initial configuration parameters MUST be identical to those passed during the memory allocation stage. |

### Example

For hypothetical codec, SIMPLE_PROC, an example for default configuration parameters is:

```
xa_codec_handle_t handle =
      (xa_codec_handle_t)malloc(handle_size);
pWORD32 scratch = (pWORD32)malloc(scratch_size);
res = xa_simple_proc_init(handle, scratch, NULL);
```

### Errors

- XA_API_FATAL_MEM_ALLOC
  - `handle` or `scratch` is `NULL`
- XA_API_FATAL_MEM_ALIGN
  - `handle` or `scratch` is not aligned correctly
- Codec specific error, if the configuration parameters (passed in `p_cfg`) are not valid

## 3.3.4 Execution Stage

The codec processes the input stream and generates the output stream frame-by-frame. Each call to the component execution function requires one complete frame as input and produces one complete frame as output. If partial input frames are presented, they are buffered internally to the component.

The codec uses default parameters to perform the processing. These parameters can be changed or queried at any time after the initialization stage.

Table 3-4 Execution Stage Functions

| Function | Description |
|---|---|
| `xa_<codec>_process` | Processes one frame of audio data |
| `xa_<codec>_set_param` | Sets the value of a particular parameter |
| `xa_<codec>_get_param` | Returns the value of a particular parameter |

The syntax of the execution stage functions are specified in the following tables.

Table 3-5 HiFi Codec Process Function Details

| | |
|---|---|
| Function | `xa_<codec>_process` |
| Syntax | `XA_ERRORCODE`<br>`xa_<codec>_process (`<br>`xa_codec_handle_t handle,`<br>`pVOID p_in_data,`<br>`pVOID p_out_data,`<br>`pUWORD32 p_in_samples,`<br>`pUWORD32 p_out_samples)` |
| Description | Processes one frame of audio using the current component state. |
| Parameters | **Input**: `handle`<br>The opaque component handle<br><br>**Input**: `p_in_data`<br>A pointer to the input audio frame from which the input data will be read. This is the input buffer<br>Required alignment: 4 bytes<br><br>**Output**: `p_out_data`<br>A pointer to the output audio frame into which the output data will be written. This is the output buffer<br>Required alignment: 4 bytes<br><br>**Input/Output**: `p_in_samples`<br>Pointer to the number of input samples. This contains the amount of data to be processed. On return, `*p_in_samples` is set to the actual amount of data processed.<br><br>**Input/Output**: `p_out_samples`<br>Pointer to the number of output samples. This contains the space available in the output buffer, in terms of samples. On return, `*p_out_samples` is set to the actual number of output samples generated by the codec. |

## Example

Following is an example for a hypothetical codec called SIMPLE_PROC, which processes 32-bit PCM data and generates 32-bit PCM data. The filling of the data is not shown.

```
UWORD32 p_in_data[512];
UWORD32 p_out_data[512];
UWORD32 in_samples = 512;
UWORD32 out_samples = 512;
res = xa_simple_proc_process(handle, p_in_data, p_out_data,
                    &in_samples, &out_samples);
```

## Errors

- XA_API_FATAL_MEM_ALLOC
  - One of the pointers (`handle`, `p_in_data`, `p_out_data`, `p_in_samples` or `p_out_samples`) is `NULL`
- XA_API_FATAL_MEM_ALIGN
  - One of the pointers (`handle`, `p_in_data`, `p_out_data`, `p_in_samples` or `p_out_samples`) is not aligned correctly
- Codec specific errors, returned if
  - Initialization function was not called
  - One of the buffer sizes (passed in `p_in_samples` or `p_out_samples`) is not valid
  - Non-specific internal library error

Table 3-6 HiFi Codec Set Parameter Function Details

| | |
|---|---|
| Function | `xa_<codec>_set_param` |
| Syntax | `XA_ERRORCODE`<br>`xa_<codec>_set_param (`<br>`xa_codec_handle_t handle,`<br>`xa_<codec>_param_id_t param_id,`<br>`pVOID p_param_value)` |
| Description | Sets the parameter specified by `param_id` to the value passed in the buffer pointed to by `p_param_value` |
| Parameters | **Input:** `handle`<br>The opaque component handle.<br><br>**Input:** `param_id`<br>Identifies the parameter to be written. Refer to Section 3.3.5 for the list of parameters supported.<br><br>**Input:** `p_param_value`<br>A pointer to a buffer that contains the parameter value<br>Required alignment: 4 bytes |

## Example

For a hypothetical codec called SIMPLE_PROC, an example to set the sampling rate is:

```
WORD32 param_id = XA_SIMPLE_PROC_SAMPLE_RATE;
WORD32 sampling_rate = 48000;
res = xa_simple_proc_set_param(handle, param_id,
&sampling_rate);
```

## Errors

- XA_API_FATAL_MEM_ALLOC
  - One of the pointers (`handle` or `p_param_value`) is `NULL`
- XA_API_FATAL_MEM_ALIGN
  - One of the pointers (`handle` or `p_param_value`) is not aligned correctly
- Codec specific errors, returned if
  - Parameter identifier (`param_id`) is not valid
  - Parameter values (passed in `p_param_value`) are not valid

Table 3-7 HiFi Codec Get Parameter Function Details

| | |
|---|---|
| Function | `xa_<codec>_get_param` |
| Syntax | `XA_ERRORCODE`<br>`xa_<codec>_get_param (`<br>`xa_codec_handle_t handle,`<br>`xa_<codec>_param_id_t param_id,`<br>`pVOID p_param_value)` |
| Description | Gets the value of the parameter specified by `param_id` in the buffer pointed to by `p_param_value` |
| Parameters | **Input:** `handle`<br>The opaque component handle.<br><br>**Input:** `param_id`<br>Identifies the parameter to be read. Refer to Section 3.3.5 for the list of parameters supported<br><br>**Output:** `p_param_value`<br>A pointer to a buffer that is filled with the parameter value when the function returns<br>Required alignment: 4 bytes |

## Example

For a hypothetical codec called SIMPLE_PROC, an example to GET the sampling rate is:

```
WORD32 param_id = XA_SIMPLE_PROC_SAMPLE_RATE;
WORD32 sampling_rate;
res = xa_simple_proc_get_param(handle, param_id,
&sampling_rate);
```

## Errors

- XA_API_FATAL_MEM_ALLOC
  - One of the pointers (`handle` or `p_param_value`) is `NULL`
- XA_API_FATAL_MEM_ALIGN
  - One of the pointers (`handle` or `p_param_value`) is not aligned correctly
- Codec specific errors, returned if
  - Parameter identifier (`param_id`) is not valid

### 3.3.5 Codec Parameters

The Programmer's Guide for a specific codec describes the parameters that are supported by the `get_param` and `set_param` functions described in Section 3.3.4.

The following information is typically included:

- Parameter ID: Parameter identifier (`param_id`)
- Value type: A pointer (`p_param_value`) to a variable of this type is to be passed
- RW: Indicates whether the parameter can be read (`get`) and/or written (`set`)
- Range: Valid values of the parameter
- Default: Default value of the parameter
- Description: Brief description of the parameter

# 4. References

[1] Xtensa® Software Development Toolkit User's Guide.

*<TOOLS_PATH>\XtDevTools\downloads\<TOOLS_VERSION>\docs\sw_dev_toolkit_ug.pdf*

[2] HiFi Audio Engine User's Guide

*<TOOLS_PATH>\XtDevTools\downloads\<TOOLS_VERSION>\docs\HiFi*_ug.pdf*

[3] HiFi Audio Codec API Definition

*HiFi-Audio-Codec-API-Definition.docx, available in the same directory.*