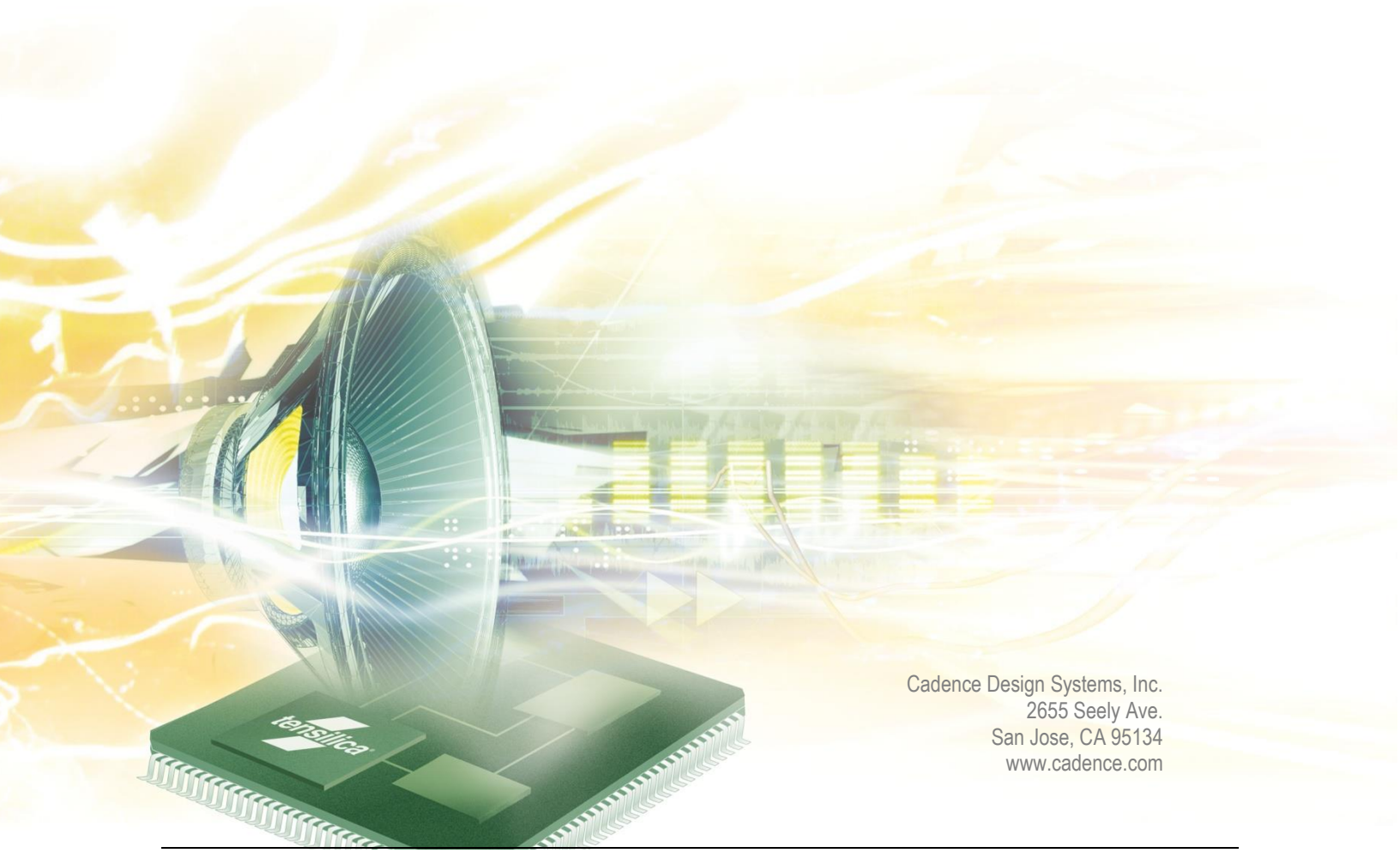
cadence®

Xtensa Audio Framework (Hostless)

Programmer's Guide

For HiFi DSPs



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

Version 1.5
March 2019

Contents

1. Introduction to Xtensa Audio Framework 1
1.1 Document Overview 1
1.2 Xtensa Audio Framework Specifications 2
1.2.1 Terminology 2
1.2.2 Feature Set 4
1.3 Xtensa Audio Framework Performance 5
1.3.1 Memory 6
1.3.2 Timings 7
2. Xtensa Audio Framework Architecture Overview 8
2.1 Xtensa Audio Framework Building Blocks 8
2.1.1 Applications 8
2.1.2 Xtensa Audio Framework 9
2.1.3 XOS (RTOS) 9
2.1.4 Audio Components 9
3. Xtensa Audio Framework Developer APIs 10
3.1 Files Specific to Developer APIs 12
3.2 Developer API-Specific Error Codes 12
3.2.1 Common API Errors 13
3.2.2 Specific Errors 13
3.3 Developer APIs 14
4. Xtensa Audio Framework Sample Applications 31
4.1 Build and Execute using makefile 35
4.1.1 Making the Executable 35
4.1.2 Usage 37
4.2 Build and Execute using Xplorer 37
5. Integration of New Audio Components with XAF 39
5.1 Component Modification 39
5.2 Component Integration 39
5.3 Component Integration – Examples 42
6. Known Issues 43
7. Appendix: Memory Guidelines 44
8. References 48

Figures

Figure 1-1 Terminology 3
Figure 2-1 Software Stack Diagram 8
Figure 3-1 Flowgraph Sequence for API Calls 11
Figure 4-1 Testbench 1 (pcm-gain) Block Diagram 31
Figure 4-2 Testbench 2 (dec) Block Diagram 31
Figure 4-3 Testbench 3 (dec-mix) Block Diagram 32
Figure 4-4 Testbench 4 (full-duplex) Block Diagram 32
Figure 4-5 Testbench 5 (amr-wb-dec) Block Diagram 32
Figure 4-6 Testbench 6 (src) Block Diagram 33
Figure 4-7 Testbench 7 (aac-dec) Block Diagram 33
Figure 4-8 Testbench 8 (mp3-dec-rend) Block Diagram 33
Figure 4-9 Testbench 9 (gain-renderer) Block Diagram 33
Figure 4-10 Testbench 10 (capturer-pcm-gain) Block Diagram 34
Figure 4-11 Testbench 11 (capturer-mp3-enc-test) Block Diagram 34

Tables

Table 1-1 Component types 4
Table 1-2 Library Memory 6
Table 1-3 Runtime Memory 6
Table 1-4 MCPS 7
Table 3-1 `xaf_adev_open` API 14
Table 3-2 `xaf_adev_close` API 16
Table 3-3 `xaf_comp_create` API 17
Table 3-4 `xaf_comp_delete` API 20
Table 3-5 `xaf_comp_set_config` API 21
Table 3-6 `xaf_comp_get_config` API 22
Table 3-7 `xaf_connect` API 23
Table 3-8 `xaf_comp_process` API 24
Table 3-9 `xaf_comp_get_status` API 27
Table 3-10 `xaf_get_verinfo` API 29
Table 5-1 Example components 42

Document Change History

Version	Changes
1.0	Initial release
1.1	Known issues (Section 6) in Release 1.0 fixed. Minor changes in API (Section 3). Mixer, audio encoder and speech decoder components with the corresponding testbenches added (Section 4).
1.2	Added support for the ALC5677 EVM. Real-time capturer and renderer components added. Xtensa tool chain v6.0.3 (RF-2015.3) supported only.
1.3	Removed references to a specific target (ALC5677 EVM). Updated Software Stack Diagram (Figure 2.1). Modified library inclusion step in Xtensa-Xplorer (section 4.2). Updated Memory Guidelines (Section 7, Appendix) and added examples.
1.4	Updated Feature Set (Section 1.2.2) and Known Issues (Section 6) about fast functional “TurboSim” ISS mode restriction with XAF. Sample Rate Convertor component wrapper is updated to work with Sample Rate Convertor v1.9 library.
1.5	Added support for Ogg-Vorbis component sample application. Added xaf_get_mem_info API support. Updated Memory and Timings tables for pcm_gain application on 7.0.5 tools.

1. Introduction to Xtensa Audio Framework

Xtensa Audio Framework (XAF) is a framework designed to accelerate the development of audio processing applications for the HiFi family of DSP cores. Application developers may choose components from the rich portfolio of audio and speech libraries already developed by Cadence and its ecosystem partners. In addition, customers may also package their proprietary algorithms and components and integrate them into the framework. Towards this goal, a simplified “Developer API” is defined, which enables application developers to rapidly create an end application and focus more on using the available components. XAF is designed to work on both the instruction set simulator as well as actual hardware.

The version of XAF described in this guide is designed to work on a single DSP (that is, a “hostless” solution).

XAF is part of the “HiFi Integrator Studio” suite of tools.

1.1 Document Overview

This guide covers all the information required to create, configure, and run audio processing chains using XAF developer APIs. Section 2 briefly describes the XAF architecture, and Section 3 provides details about developer APIs available for the application developer. Section 4 provides details about building and running a sample application, which illustrates usage of the developer APIs. Section 5 provides a “How To” guide for adding support for a new component in XAF.

1.2 Xtensa Audio Framework Specifications

This section provides XAF specifications, including the operating system.

1.2.1 Terminology

The following terms are used within this guide.

Audio Device: The software abstraction of a digital signal processor (DSP) core.

Component: A software module that conforms to a specified interface and runs on the audio device. It would implement some audio processing functionality.

Port: An interface through which a component can connect to other components and exchange data. Each port may be connected to only one port of another component. A component must have at least one port.

Input Port: A port through which a component can receive data from another component. A component may have 0 or more input ports.

Output Port: A port through which a component can send data to another component. A component may have 0 or more output ports.

Link: The connection between the output port of one component and the input port of another component.

Buffer: Memory block containing data that is transferred over a link between two ports.

Chain: A graph formed by connecting different components by links.

Framework: A software entity that enable the creation of an audio processing chain. It manages the transfer of buffers between ports as well as the scheduling of the different components in the chain.

Application: A software entity that uses the framework to create a chain. It is the responsibility of the application to provide input data to the chain and consume the output data generated by the chain.

Figure 1-1 shows the terms above in a diagrammatic form, with an example chain.

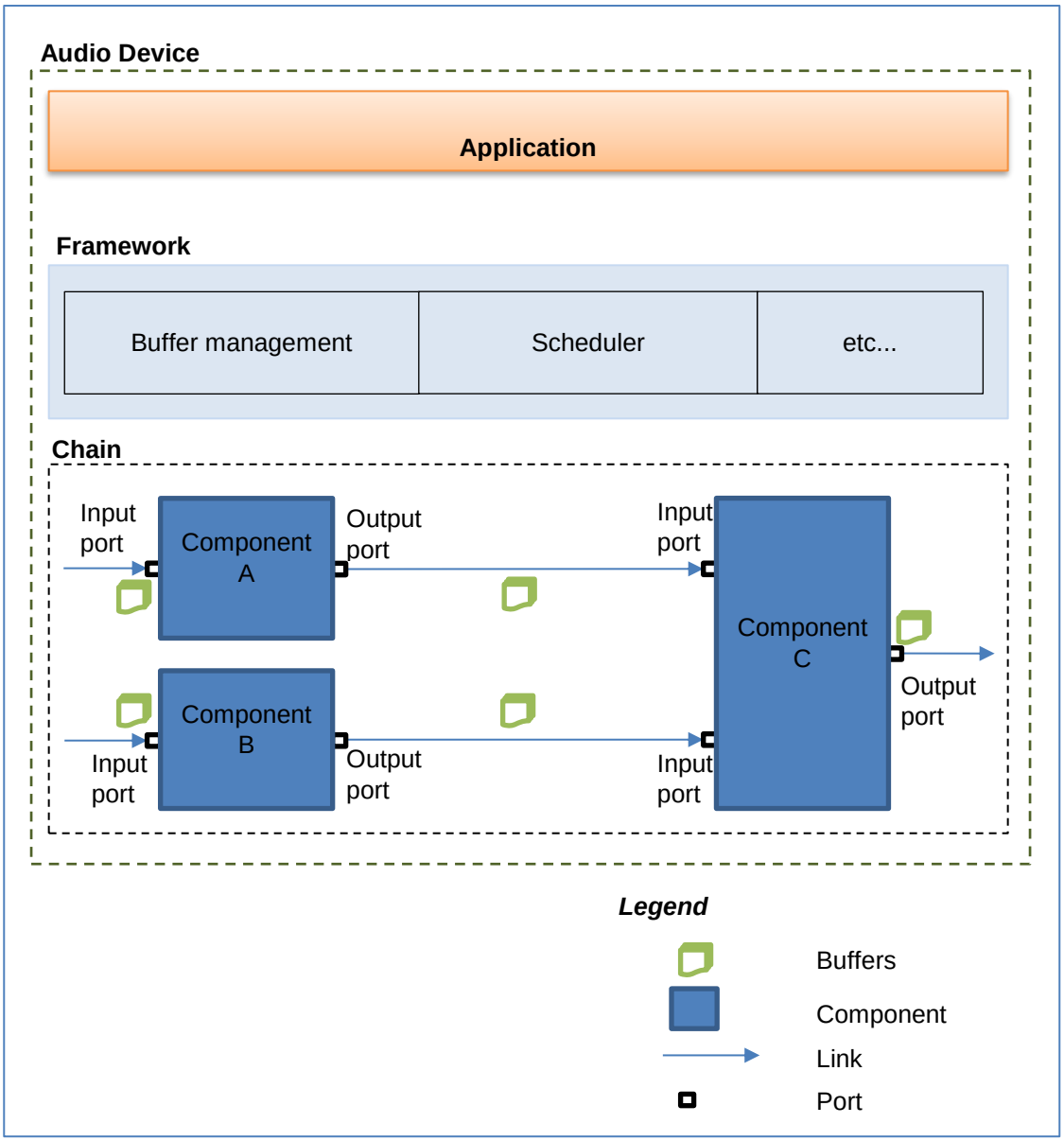


Figure 1-1 Terminology

1.2.2 Feature Set

API features:

- Ability to create components and connect them in a chain.
- Ability to read and write component configuration parameters.
- Ability to read component status and trigger component processing.

XAF features:

- Manages the scheduling of components in the chain. No explicit restriction on the complexity of the component chain, i.e., the number of components/links is restricted by the memory/MHz resources available and not by XAF.
- Manages the allocation of memory for data buffers. Number of data buffers is 0, 1, or 2 for input ports and 0 or 1 for output ports. The number of buffers on a link between two ports can be increased at component connection stage.
- Manages the allocation of memory for itself and created components. Dynamic memory allocation within XAF is done through an allocation function registered by the application. This allows the application to control the memory type/region for the allocation.
- Manages the data transfer between components. The buffering of data to match the different block sizes between two connected components is also managed by XAF. As XAF merely transfers the data between components, there is no restriction on the actual format of the data.
- Component types supported (See Table 1-1), depending on the number of ports and the type of data transferred across the ports (PCM or non-PCM).

Table 1-1 Component types

Component Type	Input		Output		Component Description
	Ports	PCM	Ports	PCM	
Decoder	1	N	1	Y	Decodes input compressed data to generate output PCM data.
Encoder	1	Y	1	N	Encodes input PCM data to generate output compressed data.
Mixer	4	Y	1	Y	Combines input PCM data from multiple ports to generate one output PCM data.
Pre-processing	1	Y	1	Y	Pre-processes input PCM data to generate output PCM data.
Post processing	1	Y	1	Y	Post-processes input PCM data to generate output PCM data.
Renderer	1	Y	0	NA	Plays input PCM data to a speaker/headphone.
Capturer	0	NA	1	Y	Captures output PCM data from a microphone.

Package features:

- Twelve example test applications provided to demonstrate various use-cases.
- Example code to demonstrate the integration of five Cadence audio libraries (MP3 decoder, MP3 encoder, AMR-WB decoder, Sample Rate Convertor, AAC decoder, Ogg-Vorbis decoder) into XAF is included in this package. Note that the actual libraries need to be licensed separately and are not part of this package.
- Compile time flags provided to support detailed analysis and debugging
  - Tracing: Enables printing of detailed component scheduling information (Set `TRACE = 1` while building, disabled by default).
  - Debugging: Disables optimization and enables debugger information (Set DEBUG = 1 while building, disabled by default).
  - Profiling: Measurement of component and XAF MHz. (Set XAF_PROFILE = 1 while building, enabled by default).

RTOS:

- XAF requires a real-time operating system (RTOS) for multithreaded execution, message queue interface, mutual exclusion for accessing shared resources, etc. This version of XAF supports Cadence XOS [1] (only) as the RTOS. Support for other RTOS variants will be added in future versions.

Limitations:

- Code for the components needs to be linked into the application. Dynamic loading of code is not supported.
- API enhancements to disconnect components from a graph and to pause/resume the graph are planned in future releases.
- Only one instance of XAF can run at a time.
- XAF does not support fast functional “TurboSim” mode of Instruction Set Simulator (ISS). ISS must be used in cycle accurate mode with XAF.

1.3 Xtensa Audio Framework Performance

The performance was characterized on the 5-stage HiFi DSP processor cores. The memory usage and performance figures are provided for design reference.

1.3.1 Memory

Table 1-2 Library Memory

Text (Kbytes)				Data (Kbytes)
HiFi Mini	HiFi 2	HiFi 3	HiFi 4	
20.0	20.6	21.7	22.7	0.6

Note Other than for Text and Data, XAF uses 8 bytes for bss. The measurements exclude the memory required by XOS and the standard C library. The measurements were done with Version 7.0.5 of the Xtensa tool chain.

The size of the total runtime memory allocated by XAF depends mainly on the two parameters `audio_frmwk_buf_size` and `audio_comp_buf_size` passed to the `xaf_adev_open()` function. Refer to Section 7 for guidelines on setting these parameters.

The total runtime memory allocated can be divided into two categories:

1. Memory used by XAF: This memory is allocated by XAF for its internal data structures.
2. Memory allocated by XAF for use by Audio Components: This is the memory that is allocated by XAF for usage by audio components.

Table 1-3 shows the runtime memory allocated by XAF for a simple processing chain.

Table 1-3 Runtime Memory

No	Memory breakup	RAM (Kbytes)			
		HiFi Mini	HiFi 2	HiFi 3	HiFi 4
1	Local Memory used by DSP Components	76.4	76.4	76.4	76.4
2	Shared Memory used by Audio Components and Framework	28	28	28	28
3	Local Memory used by Framework memory	15.2	15.1	15.2	15.2
	Total	119.6	119.5	119.6	119.6

Note The measurements were done with Version 7.0.5 of the Xtensa tool chain.

Note For this example, `audio_frmwk_buf_size` = 64 KB and `audio_comp_buf_size` = 128 KB, during `xaf_adev_open()` call. Note that some of the memory allocated during `xaf_adev_open` is for internal XAF structures and the remainder is allocated by XAF for use by the components. Refer to Section 7 for details.

1.3.2 Timings

Table 1-4 contains details for the MCPS usage for the processing function. The “Total” MCPS are the MHz consumed by the entire system. The “XAF” MCPS are the MCPS consumed by XAF. This is measured by subtracting the MCPS consumed by the application and the audio components from the total MCPS. Note that the XAF MCPS would depend on the complexity of the audio processing chain — this measurement was done on a simple chain consisting of a single PCM gain component.

Table 1-4 MCPS

Use Case		Average CPU Load (MHz)			
		HiFi Mini	HiFi 2	HiFi 3	HiFi 4
Simple PCM Gain (Mono, 48KHz, Buffer size = 4096 samples)	XAF	0.38	0.40	0.37	0.32
	Total	4.06	4.13	4.10	3.04

Note Performance specification measurements are carried out on a cycle-accurate simulator assuming an ideal memory system (that is, one with zero memory wait states) for HiFiMini/HiFi2/HiFi3/HiFi4 cores. This is equivalent to running with all code and data in local memories or using an infinite-size, pre-filled cache model. The measurements were done with Version 7.0.5 of the Xtensa tool chain.

2. Xtensa Audio Framework Architecture Overview

2.1 Xtensa Audio Framework Building Blocks

The following figure shows various building blocks of applications based on XAF. Note that in this figure the gray blocks are not part of XAF.

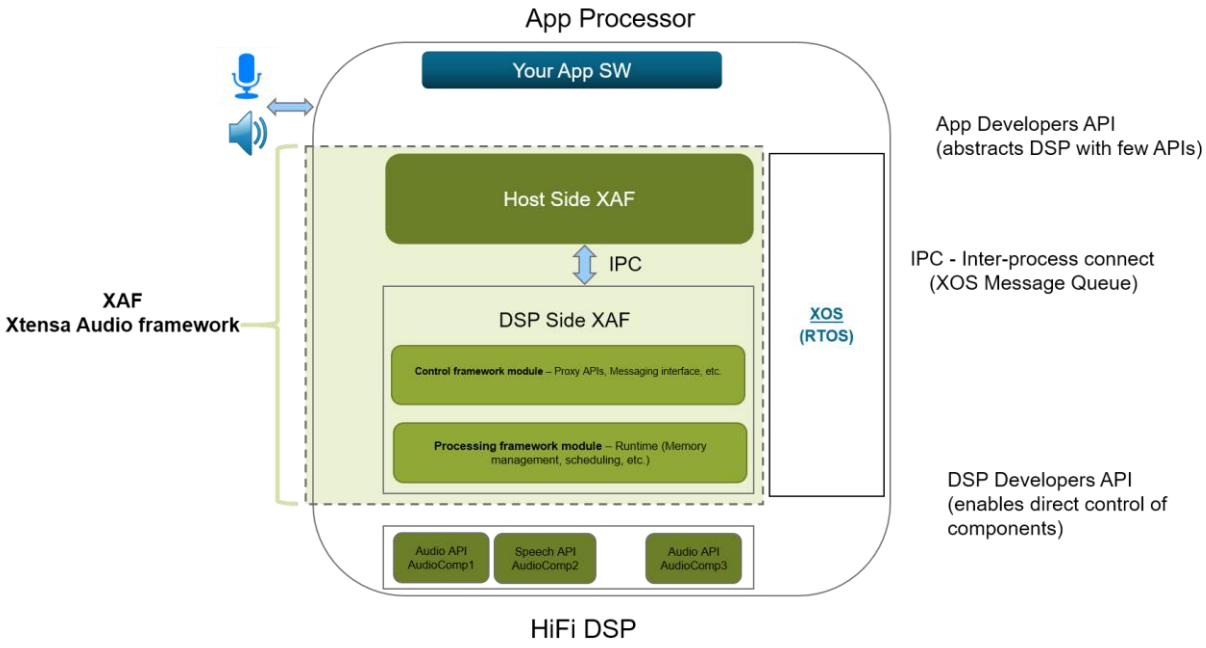


Figure 2-1 Software Stack Diagram

2.1.1 Applications

In this application space, an application developer will leverage the developer API to create a processing chain and XOS APIs to handle multiple threads. As such, XAF enables chains to be set up and configured.

The developer API is the interface between the Application space and XAF.

Note that XAF allows an unlimited number of components in the audio processing chain — the limitation is only from the system hardware. The system developer must ensure that there is enough memory and CPU bandwidth available on the hardware.

2.1.2 Xtensa Audio Framework

This block interfaces between the application framework and the audio processing components. It also performs all memory management for audio processing chains. It receives commands from applications and sends the responses back.

It also manages the data buffers between each individual audio component. Note that each audio component running in the audio processing chain may consume and produce a different amount of data. The decoder component from Cadence consumes one encoded frame and produces one output PCM frame in one execution call. For example, the MP3 decoder [4] from Cadence may consume 2048 bytes (largest encoded frame size) and produce one output PCM frame of 4608 bytes (1152 samples, stereo channel, 16-bit data). Post process components from Cadence generally consume and produce a variable block size (from a pre-defined set) in one execution call.

2.1.3 XOS (RTOS)

The applications use the XOS RTOS to create process threads for audio component chains. Also, XOS facilitates communication between the control and processing framework modules within XAF using message queues.

Note	XOS is released with the Xtensa tools SDK, and is not a part of the XAF release package.

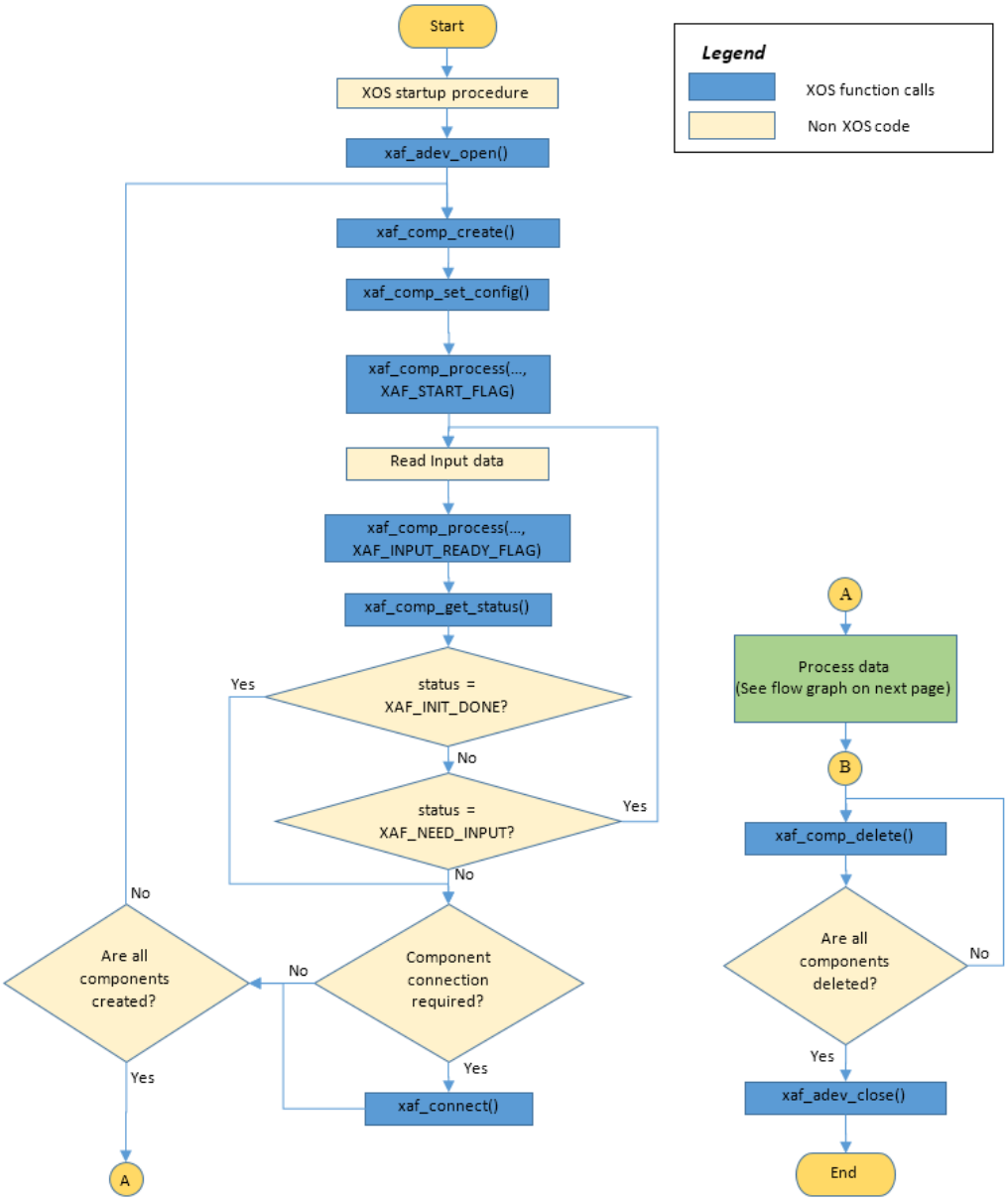
2.1.4 Audio Components

Audio components, used by the Application Framework, can be connected to form chains with a cascade or parallel interface. Except for the mixer, each audio component type can have one input and one output stream. The Mixer Component type can have up to four input streams and one output stream. Section 5 contains details on how to add a new component.

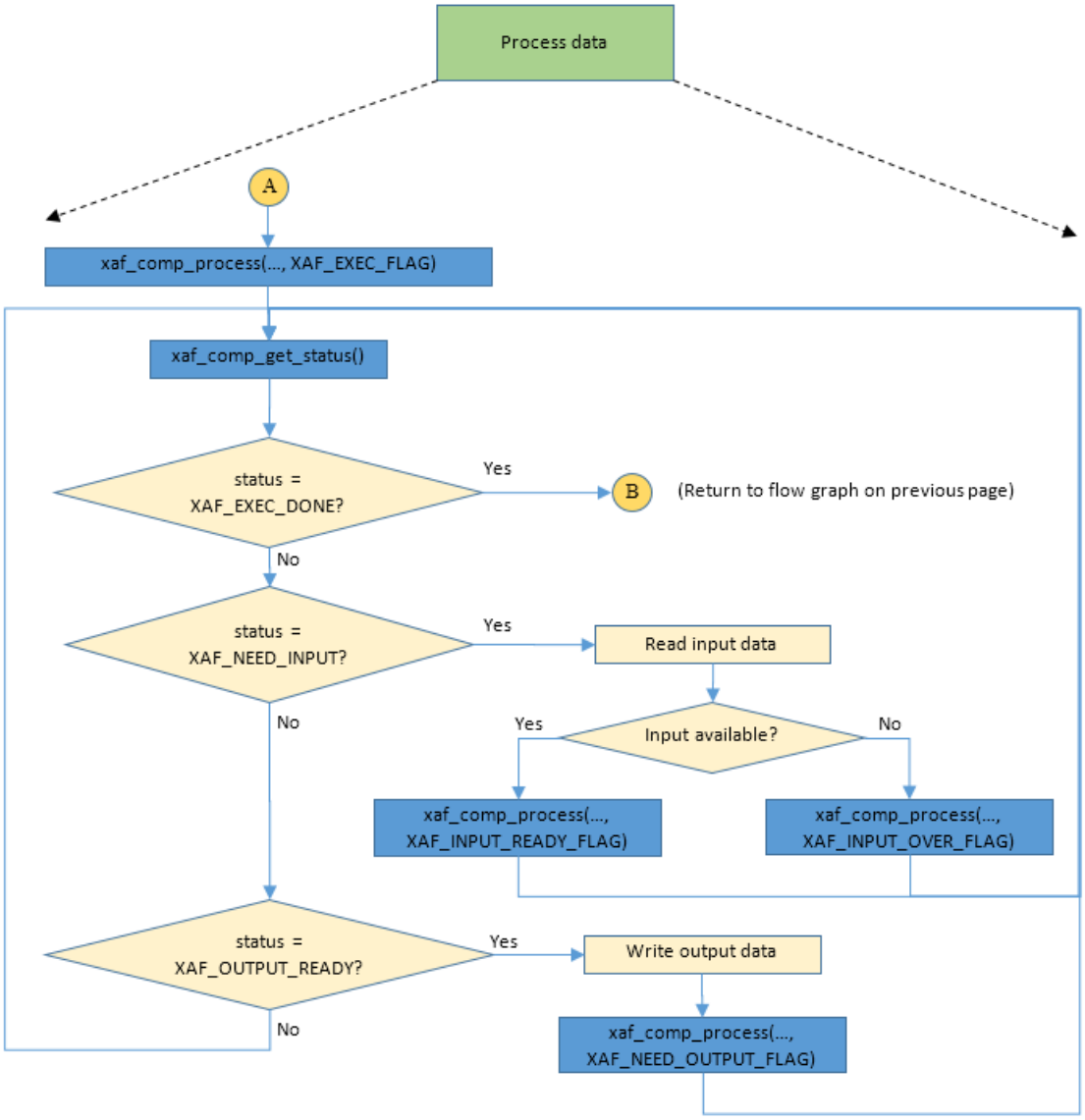
3. Xtensa Audio Framework Developer APIs

This section discusses XAF developer APIs that are available for the application programmer to create, configure and run audio processing chains.

Figure 3-1 shows the flow graph for a typical application.



(a) Flowgraph sequence for API calls of testbench



(b) Flowgraph sequence for API calls for each input and output component in the graph

Figure 3-1 Flowgraph Sequence for API Calls

Following is a brief description of the flowgraph sequence:

- Initialize XAF: The XAF is initialized by calling `xaf_adev_open`. The framework memory allocation is performed at this stage.
- Create Processing Chain: The various components in the chain are instantiated by calling `xaf_comp_create` for each component. Then, the component configuration parameters (if any) are set using `xaf_comp_set_config`. Finally, the components are connected together using `xaf_connect`.
- Process data: Input and output data is passed to the components using `xaf_comp_process`. This must be performed only for components that need to be supplied with input/output data (typically the edge components of the chain). The component status should be queried using `xaf_comp_status`. This stage continues until all the data has been processed.
- Delete Processing Chain: The various components of the chain are deleted by calling `xaf_comp_delete`.
- Terminate XAF: The XAF is terminated by calling `xaf_adev_close`. The memory allocated by the framework is freed at this stage.

3.1 Files Specific to Developer APIs

Developer API Header File (/include/)

- `xaf-api.h`

3.2 Developer API-Specific Error Codes

The errors in this section can result from the developer API layer of the Xtensa Audio Framework. All errors are fatal (unrecoverable) errors. In response to an error, the function `xaf_adev_close(p_adev, XAF_ADEV_FORCE_CLOSE)` may be called to close the device and release resources used by XAF.

3.2.1 Common API Errors

- `XAF_PTR_ERROR`

  This error indicates that a null pointer was passed to the developer API where a valid pointer was expected.

- XAF_INVALID_VALUE

  This error code indicates that an invalid value (out of valid range) was passed to the developer API.

- XAF_XOS_ERROR

  This error code indicates an internal error, typically caused when one of the XOS calls made within XAF returns an error.

- XAF_API_ERR

  This error code indicates a developer API call sequence error, for example, `xaf_comp_create()` is called before `xaf_adev_open()`.

3.2.2 Specific Errors

The following error is specific to some APIs.

- XAF_ROUTING_ERROR

  This error code indicates that the developer API was unable to connect `(xaf_connect() call)` the two requested components.

3.3 Developer APIs

This section contains tables describing the developer APIs.

Table 3-1 `xaf_adev_open` API

API	`XAF_ERR_CODE xaf_adev_open(pVOID *p_adev,`
	`WORD32 audio_frmwk_buf_size,`
	`WORD32 audio_comp_buf_size,`
	`xaf_mem_malloc_fxn_t mem_malloc,`
	`xaf_mem_free_fxn_t mem_free)`

Description	This API opens and initializes the audio device structure. It starts the processing thread that performs all audio processing. It also allocates persistent memory for Audio components and shared memory for the framework.

Actual Parameters	`p_adev`
	Pointer to audio device
	
	`audio_frmwk_buf_size`
	Size of memory allocated for the framework and for component data buffers.
	
	`audio_comp_buf_size`
	Size of memory allocated for the framework and for persistent/scratch/input/output memory allocated for audio components
	
	`mem_malloc`
	Function pointer to testbench implementation of memory allocation. The '`id`' indicates whether the memory is allocated for device (`DEV_ID`) or for component (`COMP_ID`).
	`pVOID mem_malloc(WORD32 size, WORD32 id);`
	
	Note: XAF expects that `mem_malloc` should return a 4-byte aligned address (this is required by XOS).
	
	`mem_free`
	Function Pointer to testbench implementation of memory free
	`VOID mem_free(pVOID ptr, WORD32 id);`

Restrictions	**Prerequisite:** The XOS startup procedure needs to be invoked before calling this function. This involves calling the following XOS functions. 1. `xos_set_clock_freq()` to set the core clock frequency. 2. `xos_start_main()` to start the scheduler. 3. `xos_start_system_timer()` to start the timer for scheduling. Refer to the function `start_xos()` in the file `test/src/xaf-utils-test.c` for an example. Only one instance of XAF can run at a time.

Example

```
ret = xaf_adev_open(&p_adev,
                    XAF_DSP_SHARED_POOL_SIZE,
                    XAF_DSP_SCR_PER_SIZE,
                    &mem_malloc,
                    &mem_free);
```

Errors

- Common API Errors

Table 3-2 `xaf_adev_close` API

API	`XAF_ERR_CODE xaf_adev_close(pVOID p_adev, xaf_comp_flag flag)`
Description	This API closes the Audio Device and frees up allocated memory. It also stops DSP thread execution.

Actual Parameters	`p_adev`
	Pointer to the audio device
	
	`flag`
	`XAF_ADEV_FORCE_CLOSE`: Closes the audio device, even when there are existing components. This option can be used to close the device following a fatal error.
	`XAF_ADEV_NORMAL_CLOSE`: Returns an error if there are active components in the chain. This option can be used to close the device in the normal sequence of operation.

Restrictions	Should not be called before `xaf_adev_open` API. All components must be deleted before closing the audio device. Only for a fatal error condition, the device should be force closed (i.e., with the `XAF_ADEV_FORCE_CLOSE` flag, even when all components are not deleted).

Example

```
ret = xaf_adev_close(p_adev, XAF_ADEV_NORMAL_CLOSE);
```

Errors

- Common API Errors

Table 3-3 `xaf_comp_create` API

API	`XAF_ERR_CODE xaf_comp_create(pVOID p_adev,`
	`pVOID *p_comp,`
	`xf_id_t comp_id,`
	`UWORD32 ninbuf,`
	`UWORD32 noutbuf,`
	`pVOID pp_inbuf[],`
	`xaf_comp_type comp_type)`

Description	This API creates the Audio Component. The component is identified by `comp_id` and `comp_type`. You can specify the number of input and output buffers for the component. The IO buffer requirement is dependent upon the position of the component in the audio processing chain – see the parameter description below for details.

Actual Parameters	`p_adev`
	Pointer to the audio device structure
	
	`p_comp`
	Pointer to the audio component structure (should be one of the available audio components)
	
	`comp_id`
	Component Identifier string. e.g. “mixer”, “audio-decoder/mp3”, etc. It should match with `class_id`’s defined under the constant definition of `xf_component_id` in `xa-factory.c` file. (Refer to Section 5.2, Step 6)
	
	`ninbuf`
	Unsigned integer containing the number of input buffers. This is the number of buffers that the testbench needs to pass to the component. For components connected in the chain where it receives input from other components, this must be configured as zero (0).
	Valid values: 0, 1, 2.
	
	`noutbuf`
	Unsigned integer containing the number of output buffers. This is the number of buffers that the component passes to the testbench as output. For components connected in the chain where the output is passed to another component, this must be configured as zero (0).
	Valid values: 0, 1.
	
	`pp_inbuf`
	Pointer to the array to hold `ninbuf` input buffer addresses that have been allocated within XAF. If the pointer is NULL, the input buffer addresses will not be returned.
	
	`type`
	Type of audio component
	Following are valid values:
	Type — **Description**
	`XAF_DECODER:` — Decoder component
	`XAF_ENCODER:` — Encoder component
	`XAF_MIXER:` — Mixer component
	`XAF_PRE_PROC:` — Preprocessing component
	`XAF_POST_PROC:` — Post processing component
	`XAF_RENDERER:` — Renderer component
	`XAF_CAPTURER:` — Capturer component

Restrictions	Should not be called before `xaf_adev_open`

Example

```
ret = xaf_comp_create(p_adev,
                      &p_audioComp,
                      "comp_id",
                      N_INP_BUFF,
                      N_OUT_BUFF,
                      & inbuf[0],
                      XAF_POST_PROC);
```

Errors

- Common API Errors

Table 3-4 `xaf_comp_delete` API

API	`XAF_ERR_CODE xaf_comp_delete(pVOID p_comp)`
Description	This API deletes the Audio Component and frees the memory associated with it.
Actual Parameters	`p_comp` Pointer to the audio component structure
Restrictions	Should not be called before `xaf_comp_create`. Should be used once all the threads have exited under normal execution conditions (after `xos_thread_join`). To force close the device, `xaf_dev_close` API with `XAF_ADEV_FORCE_CLOSE` flag should be used.

Example

```
ret = xaf_comp_delete(p_audioComp);
```

Errors

- Common API Errors

Table 3-5 `xaf_comp_set_config` API

API	`XAF_ERR_CODE xaf_comp_set_config(`
	`pVOID p_comp,`
	`WORD32 num_param,`
	`pWORD32 p_param)`

Description	This API sets (writes) configuration parameters to the Audio Component. `num_param` provides the number of configuration parameters to be set. `p_param` points to an array containing ID/value pairs for all `num_param` parameters. For example, for two parameters, `p_param` will contain ID1, VAL1, ID2, VAL2.

Actual Parameters	`p_comp`
	Pointer to the audio component structure
	
	`num_param`
	Integer containing the number of parameters to be set. The maximum limit is 16.
	
	`p_param`
	Pointer to an integer array containing ID/Value pairs – i.e., parameter ID followed by parameter value.

Restrictions	Should not be called before `xaf_comp_create`

Example

```
ret = xaf_comp_set_config(p_comp,
                          N_PARAMS,
                          &param[0]);
```

Errors

- Common API Errors

Table 3-6 `xaf_comp_get_config` API

API	`XAF_ERR_CODE xaf_comp_get_config(`
	`pVOID p_comp,`
	`WORD32 num_param,`
	`pWORD32 p_param)`

Description	This API gets (reads) configuration parameters from the Audio Component. `num_param` provides the number of configuration parameters to get. `p_param` points to an array containing ID/value pairs for all `num_param` parameters. For example, for two parameters, `p_param` will contain ID1, VAL1, ID2, VAL2. VAL1 and VAL2 can contain any arbitrary value, as they will be over-written when the function returns. Upon successful execution of this API, the value field of the ID/value pair will be set to the correct value.

Actual Parameters	`p_comp`
	Pointer to the audio component structure
	`num_param`
	Integer containing the number of parameters to get. The maximum limit is 16.
	`p_param`
	Pointer to an integer array containing ID/Value pairs – i.e., parameter ID followed by parameter value.

Restrictions	Should not be called before `xaf_comp_create`

Example

```
ret = xaf_comp_get_config(p_comp,
                          N_PARAMS,
                          &param[0]);
```

Errors

- Common API Errors

Table 3-7 `xaf_connect` API

<table>
<tr><td>API</td><td><pre>XAF_ERR_CODE xaf_connect(pVOID p_src,
                         pVOID p_dest,
                         WORD32 num_buf)</pre></td></tr>
<tr><td>Description</td><td>This API connects the output port of audio component <code>p_src</code> to the input port of audio component <code>p_dest</code> with <code>num_buf</code> buffers between them. The size of these buffers will be equal to the size of the output buffer of <code>p_src</code> .

This API will fail if there are no free input or output ports on Audio Components. Audio Components have input and output ports as follows:
<table>
<tr><th>Component Type</th><th>Input Ports</th><th>Output Ports</th></tr>
<tr><td><code>XAF_DECODER</code> or
<code>XAF_ENCODER</code> or
<code>XAF_PRE_PROC</code> or
<code>XAF_POST_PROC</code></td><td>1</td><td>1</td></tr>
<tr><td><code>XAF_MIXER</code></td><td>4</td><td>1</td></tr>
<tr><td><code>XAF_RENDERER</code></td><td>1</td><td>0</td></tr>
<tr><td><code>XAF_CAPTURER</code></td><td>0</td><td>1</td></tr>
</table></td></tr>
<tr><td>Actual Parameters</td><td><code>p_src</code>
Pointer to the source audio component structure

<code>p_dest</code>
Pointer to the destination audio component structure

<code>num_buf</code>
Number of buffers to be added between components
Valid values: 2, 3, 4</td></tr>
<tr><td>Restrictions</td><td>Should not be called before at least two audio components are created using <code>xaf_comp_create</code>.</td></tr>
</table>

Example

```
ret = xaf_connect(p_audioComp1,
                  p_audioComp2,
                  N_BUFFS);
```

Errors

- Common API Errors
- XAF_ROUTING_ERROR
  - Indicates that the API was unable to connect the two requested components

Table 3-8 `xaf_comp_process` API

API	`XAF_ERR_CODE xaf_comp_process(pVOID p_adev,`
	`pVOID p_comp,`
	`pVOID p_buf,`
	`UWORD32 length,`
	`xaf_comp_flag flag)`

Description	This API is the main process function for the audio component; it will do Audio Component start, initialization, execution, and wrap-up based on the process `flag` provided to it. This API needs to be called only for components that need to be supplied with input/output data, typically the edge components of the chain. After processing has started, this API should be called until end of stream, alternatively along with `xaf_comp_get_status` API. The value to be set for the parameter '`flag`' depends on the status returned by the `xaf_comp_get_status` API. **Note**: This API is asynchronous, i.e., it delivers the process command to the audio component and returns. The audio component will process this request when all required resources (IO buffers, CPU, etc.) from the processing chain are available. The status of this process command can be probed by the API described in Table 3-9. **Note**: The pointer to an audio device (`p_adev`) is not required and can be passed as NULL during the execution phase of the audio component (after the component is initialized).

<table>
<tr><td>Actual Parameters</td><td>
<code>p_adev</code>
Pointer to the audio device structure

<code>p_comp</code>
Pointer to the audio component structure

<code>p_buf</code>
Pointer to the input buffer with the input data or output buffer to be filled

<code>length</code>
Unsigned integer containing the length of buffer in bytes

<code>process_flag</code> – Process flag
Following are valid values:
<table>
<tr><th>Flag</th><th>Description</th></tr>
<tr><td><code>XAF_START_FLAG</code></td><td>Initiates processing, to be called only once for each component, during initialization.</td></tr>
<tr><td><code>XAF_EXEC_FLAG</code></td><td>Executes, to be called only once for each component to start processing.</td></tr>
<tr><td><code>XAF_INPUT_OVER_FLAG</code></td><td>Indicates input is complete, when xaf_comp_get_status returns <code>XAF_NEED_INPUT</code>, and input stream is exhausted.</td></tr>
<tr><td><code>XAF_INPUT_READY_FLAG</code></td><td>Indicates input buffer availability, when xaf_comp_get_status returns <code>XAF_NEED_INPUT</code>, and input data is available.</td></tr>
<tr><td><code>XAF_NEED_OUTPUT_FLAG</code></td><td>Request for output, when xaf_comp_get_status returns <code>XAF_OUTPUT_READY</code>, and size returned in p_info[1] is non-zero.</td></tr>
</table>
</td></tr>
<tr><td>Restrictions</td><td>Should not be called before <code>xaf_comp_create</code></td></tr>
</table>

Example

```
ret = xaf_comp_process( p_adev,
                        p_audioComp,
                        &Buff,
                        length,
                        compFlag);
```

Errors

- Common API Errors

Table 3-9 `xaf_comp_get_status` API

<table>
<tr><td>API</td><td>

```
XAF_ERR_CODE xaf_comp_get_status(
                          pVOID p_adev,
                          pVOID p_comp,
                          xaf_comp_status *p_status,
                          pVOID p_info)
```

</td></tr>
<tr><td>Description</td><td>This API returns the status of the audio component and associated information. <code>p_adev</code> and <code>p_comp</code> should point to the valid audio device and audio component structures respectively. This API will return one of following status and associated information.
Note: This API is a blocking API, i.e., it may block for status from the DSP thread for a previously issued process command.</td></tr>
<tr><td>Actual Parameters</td><td><code>p_adev</code>
Pointer to the audio device structure

<code>p_comp</code>
Pointer to the audio component structure

<code>p_status</code>
Pointer to get the audio component status
Valid values are:
<table>
<tr><th>Flag</th><th>Description</th><th><code>p_status</code></th></tr>
<tr><td><code>XAF_STARTING</code></td><td>Started</td><td></td></tr>
<tr><td><code>XAF_INIT_DONE</code></td><td>Initialization complete</td><td></td></tr>
<tr><td><code>XAF_NEED_INPUT</code></td><td>Component needs data</td><td>Buffer pointer, size in bytes</td></tr>
<tr><td><code>XAF_OUTPUT_READY</code></td><td>Component has generated output</td><td>Buffer pointer, size in bytes</td></tr>
<tr><td><code>XAF_EXEC_DONE</code></td><td>Execution done</td><td></td></tr>
</table>

<code>p_info</code>
Pointer to array of size 2 (pointer, size) to get information from the audio component associated with its status. When the p_status returned is <code>XAF_STARTING</code> or <code>XAF_INIT_DONE</code>, this buffer is not updated.</td></tr>
<tr><td>Restrictions</td><td>Should not be called before <code>xaf_comp_create</code></td></tr>
</table>

Example

```
ret = xaf_comp_get_status(p_adev,
                          p_audioComp,
                          &compStatus,
                          &Info[0]);
```

Errors

- Common API Errors

Table 3-10 `xaf_get_verinfo` API

API	`XAF_ERR_CODE xaf_get_verinfo (pUWORD8 ver_info[3])`
Description	This API gets the version information from the XAF library. It returns an array of the following three strings. `ver_info[0]` Library name `ver_info[1]` Library version `ver_info[2]` API version
Actual Parameters	`ver_info` Pointer to array of three strings
Restrictions	None

Example

```
ret = xaf_get_verinfo(&versionInfo[0]);
```

Errors

- Common API Errors

Table 3-11 `xaf_get_mem_stats` API

API	`XAF_ERR_CODE xaf_get_mem_stats(pVOID p_adev, WORD32 *p_mem_stats)`
Description	This API returns the information about the memory usage statistics of the audio components, framework and XAF. `p_adev` should point to the valid audio device structure. This API will update the pointer contents with memory usage statistics.

Actual Parameters	`p_adev`
	Pointer to the audio device structure
	
	`p_mem_stats`
	Pointer to an array of three WORD32 data types to get information from the API about the memory usage statistics in bytes.
	1. Local Memory used by DSP Components (`p_mem_stats[0]`),
	2. Shared Memory used by Components and Framework (`p_mem_stats[1]`) and
	3. Local Memory used by Framework (`p_mem_stats[2]`)

Restrictions	The API is recommended to be used at the very end of application execution and before closing the device (using xaf_dev_close API call) for the memory statistics to be reliable.

Example

```
WORD32 mem_stats[3];
ret = xaf_get_mem_stats(p_adev,
                        &mem_stats[0]);
```

Errors

- Common API Errors

4. Xtensa Audio Framework Sample Applications

Eleven sample applications (testbenches) are provided, which implement eleven different audio processing chains as described below. Audio components and links are shown in blue in the diagrams below.

Note	All of the audio component libraries used in this document's example testbenches are not included in the XAF release package. They need to be separately licensed.

Testbench 1 (`xa_af_hostless_test`) applies gain to PCM streams.

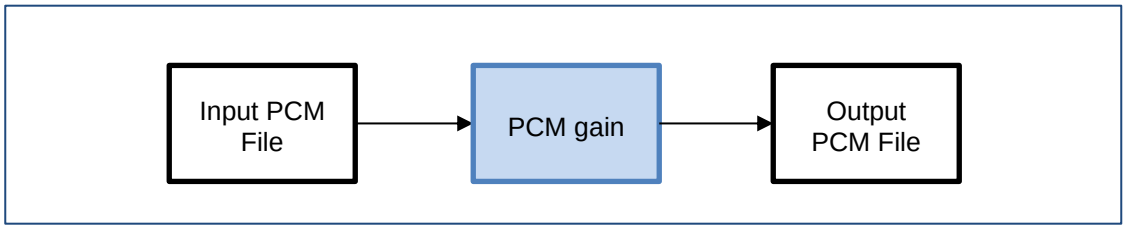


Figure 4-1 Testbench 1 (pcm-gain) Block Diagram

Testbench 2 (`xaf-dec-test`) decodes MP3 streams.

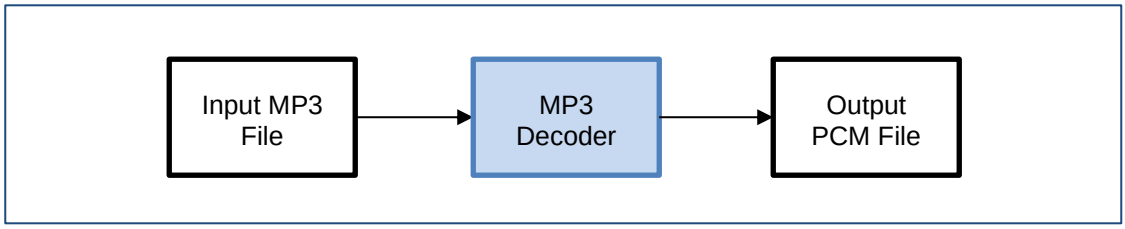


Figure 4-2 Testbench 2 (dec) Block Diagram

Testbench 3 (`xaf-dec-mix-test`) decodes two MP3 streams and mixes the output.

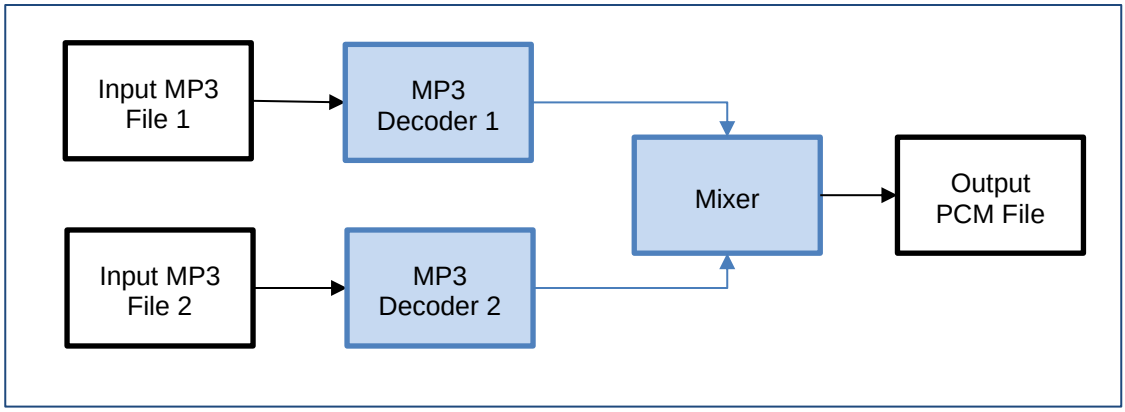


Figure 4-3 Testbench 3 (dec-mix) Block Diagram

Testbench 4 (`xaf-full-duplex-test`) decodes an MP3 stream and simultaneously encodes an MP3 stream.

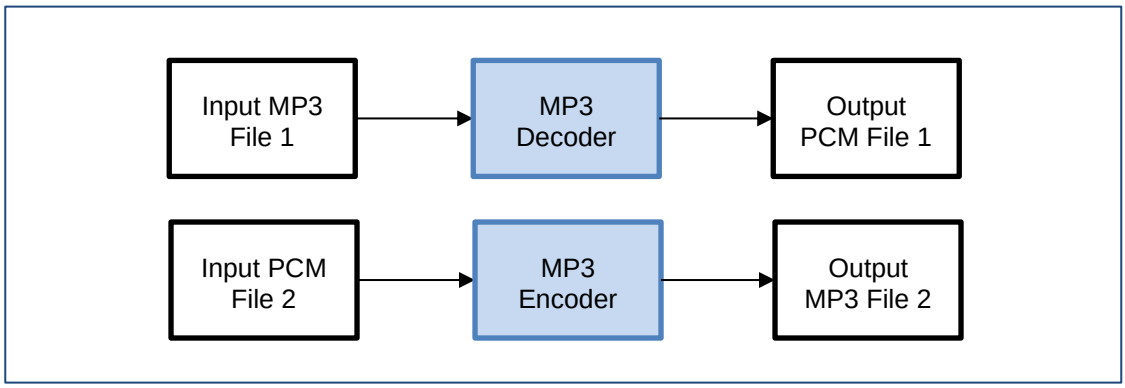


Figure 4-4 Testbench 4 (full-duplex) Block Diagram

Testbench 5 (`xaf-amr-wb-dec-test`) decodes AMR-WB streams.

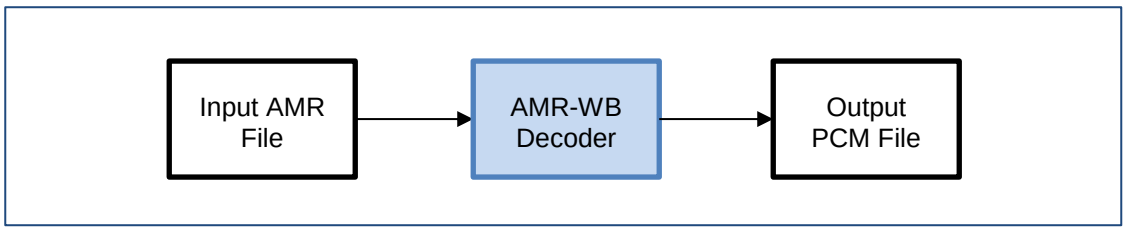


Figure 4-5 Testbench 5 (amr-wb-dec) Block Diagram

Testbench 6 (`xaf-src-test`) does a sample rate conversion.

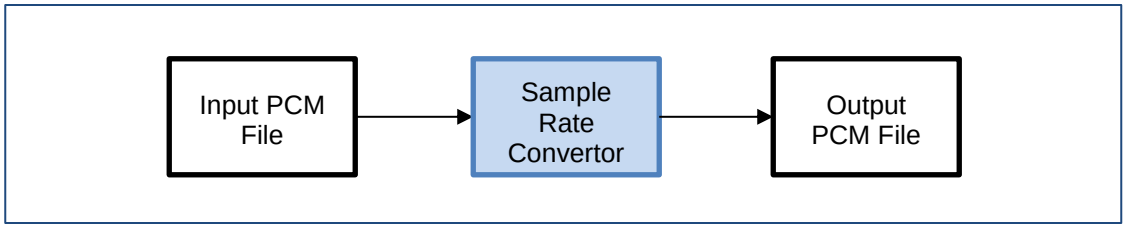


Figure 4-6 Testbench 6 (src) Block Diagram

Testbench 7 (`xaf-aac-dec-test`) decodes AAC streams.

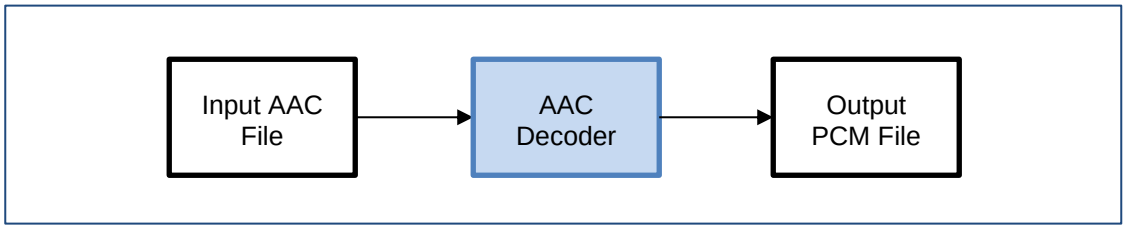


Figure 4-7 Testbench 7 (aac-dec) Block Diagram

Testbench 8 (`xaf-mp3-dec-rend-test`) decodes MP3 streams and renders it on the audio output device (hardware case). For the simulator case, the output is written to a file.

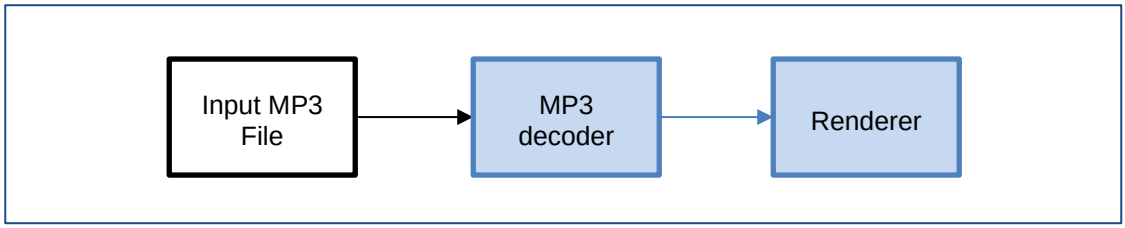


Figure 4-8 Testbench 8 (mp3-dec-rend) Block Diagram

Testbench 9 (`xaf-gain-renderer-test`) applies gain to PCM streams and renders it on the audio output device (hardware case). For the simulator case, the output is written to a file.

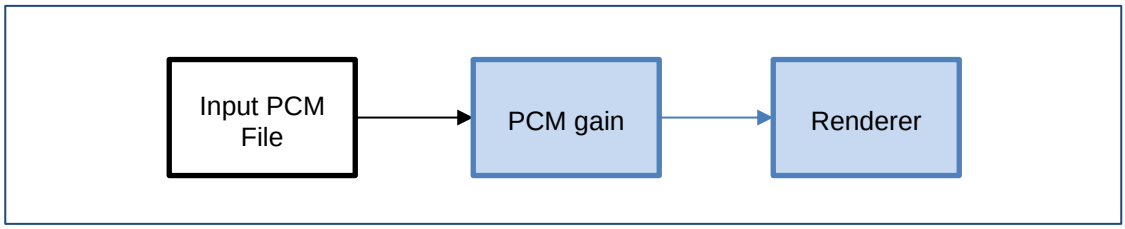


Figure 4-9 Testbench 9 (gain-renderer) Block Diagram

Testbench 10 (`xaf-capturer-pcm-gain-test`) captures a PCM stream from the audio input device (hardware case) and applies a gain to it. For the simulator case, the input is read from a file.

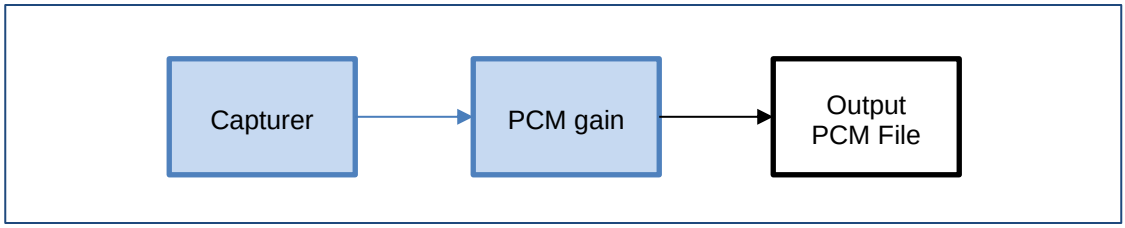


Figure 4-10 Testbench 10 (capturer-pcm-gain) Block Diagram

Testbench 11 (`xaf-capturer-mp3-enc-test`) captures data from the audio input device (hardware case) and encodes it to an MP3 stream. For the simulator case, the input is read from a file.

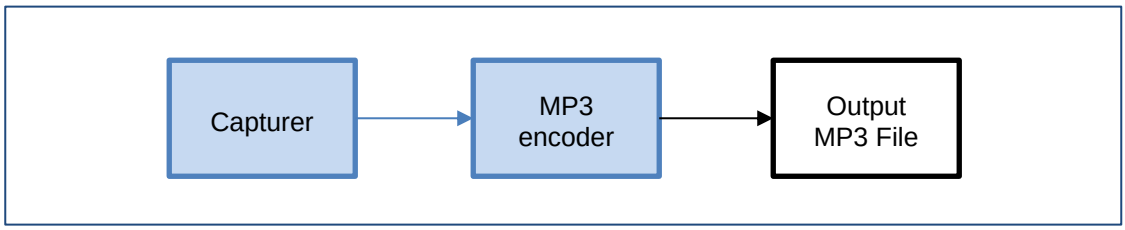


Figure 4-11 Testbench 11 (capturer-mp3-enc-test) Block Diagram

Testbench 12 (`xaf-vorbis-dec-test`) decodes Ogg streams.

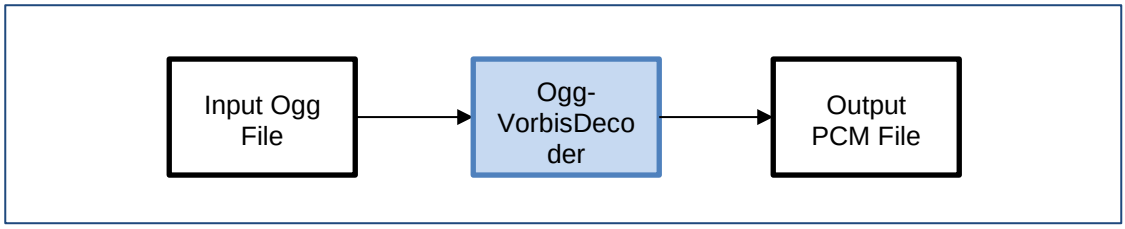


Figure 4-12 Testbench 12 (vorbis-dec) Block Diagram

Testbench specific source files (`/test/src/`)

- `xaf-pcm-gain-test.c`
- `xaf-dec-test.c`
- `xaf-dec-mix-test.c`
- `xaf-full-duplex-test.c`
- `xaf- amr-wb-dec-test.c`
- `xaf-src-test.c`
- `xaf-aac-dec-test.c`
- `xaf-mp3-dec-rend-test.c`

- xaf-gain-renderer-test.c
- xaf-capturer-pcm-gain-test.c
- xaf-capturer-mp3-enc-test.c
- xaf-vorbis-dec-test.c

Note: For the testbench xaf-src-test.c, execution is repeated 32 times with the same parameters, demonstrating consistency of the framework.

Common testbench source files (/test/src/)

- xaf-clk-test.c – Clock functions used for MCPS measurements.
- xaf-mem-test.c – Memory allocation functions.
- xaf-utils-test.c – Other shared utility functions.
  - xaf-fio-test.c – File read and write support

Other directories (in /test/)

- plugins/ – Wrappers for the different audio components.
- test_inp/ – Input data for the test execution.
- test_out/ – Output data from test execution will be written here.
- test_ref/– Reference data against which the generated output can be compared.

The XAF library and all testbenches can be built and executed using either of two ways, described in the following sections.

1. Using makefile
2. Using Xplorer workspace

4.1 Build and Execute using makefile

4.1.1 Making the Executable

Before building the executable, ensure the environment variable $XTENSA_CORE is set correctly.

Testbench 1 Only:

To build the pcm-gain testbench application, follow these steps:

3. Go to /test/build.
4. At the prompt type:

```
$ xt-make –f makefile_testbench_sample clean all
```

This will build the example test application `xa_af_hostless_test`.

All Testbenches:

To build the other testbenches, the Cadence MP3 decoder [4], MP3 encoder [5], AMR-WB decoder [6,7], Sample rate convertor [8] , AAC decoder [9] and Ogg-Vorbis[10] libraries are required. Copy these libraries to the following directories.

```
/test/plugins/cadence/mp3_dec/lib/xa_mp3_dec.a
/test/plugins/cadence/mp3_enc/lib/ xa_mp3_enc.a
/test/plugins/cadence/amr_wb/lib/xa_amr_wb_codec.a
/test/plugins/cadence/src-pp/lib/xa_src_pp.a
/test/plugins/cadence/aac_dec/lib/xa_aac_dec.a
/test/plugins/cadence/vorbis_dec/lib/xa_vorbis_dec.a
```

5. Go to /test/build.
6. At the prompt, type:

```
$ xt-make –f makefile_testbench_sample clean all-dec
```

This will build all the testbench applications.

Note: If source code distribution is available, the library needs to be built before building the testbench application. To build the XAF library, follow these steps:

1. Go to `/build/`.
2. At the prompt, type

```
$ xt-make clean all install
```

This command will build the XAF library and copy it to the `/lib/` folder.

Special Build Settings

- To build in the debug mode, add "`DEBUG=1`" to the command lines described above.
- To build with trace prints, add "`TRACE=1`" to the command lines described above.

4.1.2 Usage

The sample application executables can be run as described below using the cycle-accurate mode of the Instruction Set Simulator (ISS). The input files for the applications are stored in the `test/test_inp` folder. The generated output files are available in the `test/test_out` folder. These can be compared against the reference output files in the `test/test_ref` folder.

Testbench 1 only:

To run only the pcm-gain test application, at the prompt (in `test/build`), type:

```
$ xt-make –f makefile_testbench_sample run
```

All Testbenches:

To run all the five testbenches, at the prompt (in `test/build`), type:

```
$ xt-make –f makefile_testbench_sample run-dec
```

Note In Instruction Set Simulator (ISS) mode, the renderer testbench output is stored to the output file in the `test/test_out/` path. Similarly, the input for capturer testbench is read from the input file stored in `test/test_inp/` path.

4.2 Build and Execute using Xplorer

Note The above testbenches require Xtensa Xplorer version 6.0.3 or greater.

Following are the steps for importing to Xplorer and building testbenches:

1. To import the HiFi Audio Framework Xtensa Workspace file (extension xws) into Xplorer, click on “File” and select “Import…”. The “Import” wizard opens. Select “Import Xtensa Xplorer Workspace”. Click “Next >”. Browse for the Xtensa workspace file, and click “Next >”. Select the available project checkboxes and click “Finish”.
2. Select “testxa_af_hostless” as the active project and any of the compatible HiFi core as the configuration.
3. Build by clicking on the “Build Active” button.
4. To run Testbench 1 (PCM gain), from the “Run configurations” menu, under “Arguments” tab in “Program Arguments” text box add the following text and click on “Run” button.

   ```
   -infile:<input PCM file> -outfile:<output PCM file>
   ```

1. To build and run other testbenches, follow below steps:
   a. Copy the library binary of the component (if required) to the location test/plugins/cadence/<component>/lib/.

b. Ensure that following symbols are defined in “Build Properties” under “Symbols” tab.

```
-DXA_MP3_DECODER=1
-DXA_MP3_ENCODER=1
-DXA_SRC_PP_FX=1
-DXA_AAC_DECODER=1
-DXA_MIXER=1
-DXA_AMR_WB_DEC=1
-DXA_RENDERER=1
-DXA_CAPTURER=1
-DXA_VORBIS_DECODER=1
```

c. In the “Build Properties” wizard, under “Addl Linker” tab, in the “Additional linker options” add the component library name and the path of the library required by the testbench. The path can either be absolute path or relative path (eg. ${*workspace_loc*:testxa_af_hostless/test/plugins/cadence/aac_dec/lib}/xa_aac_dec.a).

d. Exclude testbench file and component wrapper source files of testbench other than one you want to run, by right clicking on those files and selecting “Build\Exclude”.

e. Include testbench file and component wrapper source files of the required testbench by right clicking on those files and selecting “Build\Include”.

f. Follow steps 3 and 4 as given above, with appropriate command-line arguments.

5. Integration of New Audio Components with XAF

This section describes how to create an application with a new audio component in addition to the existing example audio components.

5.1 Component Modification

The new component must be modified as follows:

1. Change the component interface to conform to the HiFi Audio/Speech Codec Application Programming Interface [2][3]. The interface (API) is a C-callable API that is exposed by all the HiFi based Audio/Speech Codecs developed by Cadence. An “audio codec” is a generic term for any audio processing component and is not restricted to encoders and decoders.
2. XAF requires all components to support the following configuration parameters for the PCM data ports.
3. XA_CODEC_CONFIG_PARAM_CHANNELS: Number of channels.
4. XA_CODEC_CONFIG_PARAM_SAMPLE_RATE: Sampling rate.
5. XA_CODEC_CONFIG_PARAM_PCM_WIDTH: PCM width.
6. Build the audio component using the Xtensa tools to create a library targeted at the appropriate HiFi core.

5.2 Component Integration

The following steps need to be followed to integrate the library into the component. For each step, the corresponding step for the MP3 decoder library is also provided as an example, marked by **MP3_DEC_EG**.

Integration Step 1: Add component files

Three files have to be added to the XAF library to enable support for a new component.

- Header file containing the library API definition.
- Library file implementing the library.
- Wrapper file that “glues” the library to the XAF.

The detailed steps are as follows:

1. Create a separate folder under `/test/plugins/` for the new component.

   MP3_DEC_EG: `test/plugins/cadence/mp3_dec`

2. Copy the component library for the appropriate core(s) to that folder

   MP3_DEC_EG: `test/plugins/cadence/mp3_dec/lib/xa_mp3_dec.a`

3. Copy the API header file for the audio component to the `test/include/audio` folder. This header file must contain the library entry point declaration and all associated structures and constants.

   MP3_DEC_EG: `test/include/audio/xa_mp3_dec_api.h`

4. Create a wrapper file for the new component in the `/test/plugins/` folder. The wrapper file connects the library to XAF.

   MP3_DEC_EG: `test/plugins/cadence/mp3_dec/xa-mp3-decoder.c`

Integration Step 2: Update the application to include the component

The application must be updated to include references to the new component. The detailed steps are as follows:

5. In the `test/plugins/xa-factory.c` file, add the audio component entry point API function extern declaration.

   MP3_DEC_EG: The line below in `xa_factory.c`

   ```
   extern XA_ERRORCODE xa_mp3_decoder(xa_codec_handle_t, WORD32, WORD32, pVOID);
   ```

6. In the constant definition of `xf_component_id` (in `xa_factory.c`), add the registration information for the new audio component.

   MP3_DEC_EG: The line below in `xa_factory.c`

   ```
   {"audio-decoder/mp3", xa_audio_codec_factory, xa_mp3_decoder},
   ```

   The required fields are:

   a. `class_id` (string identifier): This defines the class name and the component name. The different class names are defined in the `comp_id` array.

      MP3_DEC_EG: "audio-decoder/mp3"

   b. `class_constructor` - predefined by XAF and can be either of:
      - `xa_audio_codec_factory` (for components with a single input buffer and a single output buffer), or
      - `xa_mixer_factory` (for components with multiple input buffers and a single output buffer),
      - `xa_renderer_factory` (for components with a single input buffer and no output buffer)
      - `xa_capturer_factory` (for components with a single output buffer and no output buffer)

      MP3_DEC_EG: xa_audio_codec_factory

c. The function name for the audio component entry point, as defined in the API header file for the component.

MP3_DEC_EG: xa_mp3_decoder

Integration Step 3: Create the application to use the component

7. Create a new audio application source file in the `test/src/` folder. The audio application would use the XAF calls to create and run an audio processing chain with the new component.

   MP3_DEC_EG: `test/src/xaf-dec-test.c`. In this file, the audio processing chain consists of the MP3 decoder alone. Data is read from a file and provided to the MP3 decoder. The output from the MP3 decoder is written to a file. For more complicated processing chains involving the MP3 decoder, please refer to `test/src/xaf-dec-mix-test.c` (MP3 decoder and mixer) `and xaf-mp3-dec-rend-test.c` (MP3 decoder and renderer).

8. Update the `build/makefile_testbench` file to compile the wrapper file

   MP3_DEC_EG: `PLUGINOBJS_MP3_DEC += xa-mp3-decoder.o`.

9. Update the `build/makefile_testbench` file to include the library file

   MP3_DEC_EG: Refer lines containing `xa_mp3_dec.a`.

10. Update the `build/makefile_testbench` file to compile the application source file

    MP3_DEC_EG: `APP2OBJS = xaf-dec-test.o`.

11. Update the `build/makefile_testbench` file to create and run the final executable

    MP3_DEC_EG: Refer lines to the target `dec`, `dec-mix`, `full-duplex` and `run-dec`.

12. Build and test the application. Refer to the procedure in Section 4.1.

5.3 Component Integration – Examples

Several example components are provided that can be used as starting points for the development of new components. These are described in Table 5-1. The table does not include the mixer, renderer, and capturer which are considered to be part of XAF. The component folders are under `test/plugins/cadence` and the applications are in the `test/src` folder.

Table 5-1 Example components

Component Name	API	Description	References
Cadence MP3 decoder[4]	Audio[2]	Decodes MP3 data	Folder: `mp3_dec` Application: `xaf-dec-test.c`, `xaf-dec-mix-test.c`, `xaf-full-duplex-test.c`, `xaf-mp3-dec-rend-test.c`
Cadence MP3 encoder[5]	Audio[2]	Encodes MP3 data	Folder: `mp3_enc` Application: `xaf-full-duplex-test.c`, `xaf-capturer-mp3-enc-test.c`
Cadence AMR-WB decoder[6]	Speech[3]	Decodes AMR-WB data	Folder: `amr_wb` Application: `xaf-amr-wb-dec-test.c`
Cadence Sample rate convertor[8]	Audio[2]	Converts sampling rate	Folder: `src-pp` Application: `xaf-src-test.c`
Cadence AAC decoder[9]	Audio[2]	Decodes AAC data	Folder: `aac_dec` Application: `xaf-aac-dec-test.c`
Cadence Ogg-Vorbis decoder[109]	Audio[2]	Decodes Ogg data	Folder: vorbis_dec Application: `xaf-vorbis-dec-test.c`

6. Known Issues

The XAF has only been tested with Version 6.0.3 of the Xtensa tool chain. The Instruction Set Simulator (ISS) has been used in the cycle-accurate simulation mode. The fast functional “TurboSim” mode has not been tested.

XAF is a multi-threaded execution with time dependent conditions and checks (wait timeouts etc.) and it is not expected to work in fast functional “TurboSim” simulation mode seamlessly. So XAF must not be used with fast functional “TurboSim” simulation mode.

7. Appendix: Memory Guidelines

XAF manages the allocation of memory for all created components. Most of the memory is allocated within the `xaf_adev_open` function and depends on the two parameters `audio_comp_buf_size` and `audio_frmwk_buf_size` passed to this function. This section provides guidelines to the application developer to compute these parameters.

Notation: Consider a chain of N components, where the n^{th} component has A_n input ports and B_n output ports and requires P_n, S_n, I_n, and O_n KB for persistent, scratch, input, and output buffers respectively. Assume that the n^{th} component is created (`xaf_comp_create`) with X_n input buffers and Y_n output buffers. Note that X_n would be zero except for the components that need to receive data from the application and Y_n would be zero except for the components that need to send data to the application. Furthermore, assume that the n^{th} component is connected (`xaf_comp_connect`) to another component with Z_n buffers (to be counted only if the n^{th} component is connected to another component).

XAF allocates two memory buffers within the `xaf_adev_open()` function.

- Audio component buffer of size `audio_comp_buf_size`: All memory required by the components is allocated from this buffer – this includes persistent, scratch, input, and output buffers required by the component. The persistent, scratch, input, and output buffer sizes for a component are typically mentioned in the programmer's guide for that particular component.

  Then the total memory required by all components in the chain would be given by the formula:

$$T = T_1 + T_2, \qquad T_1 = \sum_{n=1}^{N} (P_n + A_n I_n + B_n O_n Z_n), \qquad T_2 = \max_n S_n$$

  T_1 is the sum of the persistent, input and output sizes required by the components. T_2 is the maximum scratch memory required by the components, as the scratch memory is shared across components. In this version of XAF, T_2 is fixed at 56 KB, via the compile time constant `XF_CFG_CODEC_SCRATCHMEM_SIZE`. Furthermore, some memory is required by XAF itself. The size of the memory required by XAF is (N + 16) KB, where N is the number of components. Note that, this 1 KB per component includes each component's API-structure, memory table and miscellaneous audio-framework data structures for the component.

  Thus, `audio_comp_buf_size` should be set to a value greater than (T1 + 56 + N + 16) KB.

- XAF buffer of size `audio_frmwk_buf_size`: All buffers exchanged between components and the application are allocated from this buffer. The number of buffers exchanged are defined in the `xaf_comp_create` call.

Then the total memory required by all components in the chain would be given by the formula:

$$S = \sum_{n=1}^{N} (4A_n X_n + O_n B_n Y_n),$$

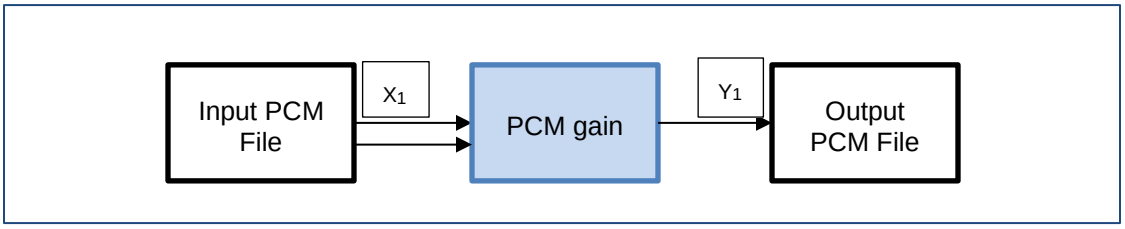
In this version of XAF, the inter-component input buffer size is fixed at 4 KB, via the compile time constant `XAF_INBUF_SIZE`. Furthermore, some memory is also required by XAF itself. The size of the memory required by XAF is 16 KB, independent of the number of components.

Thus, `audio_frmwk_buf_size` should be set to a value greater than (S + 16) KB.

Following examples illustrate the memory size computations described above for two example testbenches. Note that, memory numbers provided in these examples are for AE_HiFi3_LE5 core.

- Example 1: "PCM_Gain"(`xa_af_hostless_test`)

  Number of components, N =1 (PCM Gain)



n = 1 (PCM-gain):

A_1 = 1, B_1 = 1, X_1 = 2, Y_1 = 1, Z_1 = 0, S_1 (Scratch Memory) = 4 KB, P_1 (Persistent Memory) = 0, I_1 (Input buffer) = 4 KB, O_1 (Output buffer) = 4 KB

`audio_comp_buf_size` Computation:

$T_1 = 0(P_1) + 1(A_1) * 4(I_1) + 1(B_1) * 4(O_1) * 0 (Z_1)$ = 4 KB

T = 4 (T_1) + 56 + 1 + 16 = 77 KB is the required `audio_comp_buf_size`.

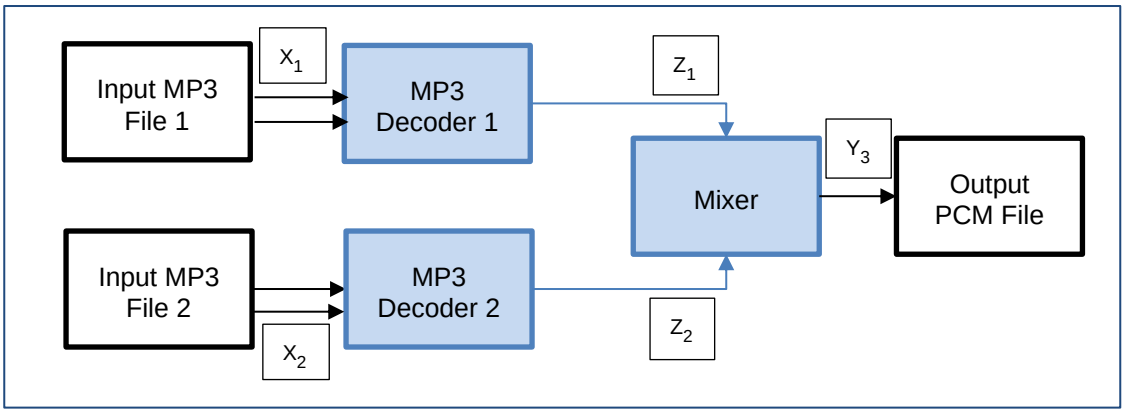
`audio_frmwk_buf_size` Computation:

$S = 4 * 1(A_1) * 2(X_1) + 4(O_1) * 1(B_1) * 1(Y_1)$ = 12 KB

S + 16 = 28 KB is the required `audio_frmwk_buf_size`.

- Example 2: "2 MP3 Decoder + Mixer" (`xaf-dec-mix-test`)

  Number of components, N =3 (MP3 Decoder1, MP3 Decoder2, Mixer)



  n = 1 (MP3 Decoder1):

  A_1 = 1, B_1 = 1, X_1 = 2, Y_1 = 0, Z_1 = 4, S_1 (Scratch Memory) = 7 KB, P_1 (Persistent Memory) = 12.125 KB, I_1 (Input buffer) = 2 KB, O_1 (Output buffer) = 4.5 KB

  n = 2 (MP3 Decoder2):

  A_2 = 1, B_2 = 1, X_2 = 2, Y_2 = 0, Z_2 = 4, S_2 (Scratch Memory) = 7 KB, P_2 (Persistent Memory) = 12.125 KB, I_2 (Input buffer) = 2 KB, O_2 (Output buffer) = 4.5 KB

  n = 3 (Mixer):

  A_3 = 4, B_1 = 1, X_3 = 0, Y_3 = 1, Z_3 = 0, S_3 (Scratch Memory) = 2 KB, P_3 (Persistent Memory) = 0, I_3 (Input buffer) = 2 KB, O_3 (Output buffer) = 2 KB.

  `audio_comp_buf_size` Computation:

  sum1 = 12.125 (P_1) + 1 (A_1) * 2 (I_1) + 1 (B_1) * 4.5 (O_1) * 4 (Z_1) = 32.125 KB

  sum2 = 12.125 (P_2) + 1 (A_2) * 2 (I_2) + 1 (B_2) * 4.5 (O_2) * 4 (Z_2) = 32.125 KB

  sum3 = 0 (P_3) + 4 (A_3) * 2 (I_3) + 1 (B_3) * 2 (O_3) * 0 (Z_3) = 8 KB

  T_1 = 32.125 + 32.125 + 8 = 72.25 KB

  T = 72.25 (T_1) + 56 (T_2) + 3 + 16 = 147.25 KB is the required `audio_comp_buf_size`.

  `audio_frmwk_buf_size` Computation:

  sum1 = 4 * 1 (A_1) * 2 (X_1) + 4.5 (O_1) * 1 (B_1) * 0 (Y_1) = 8 KB

  sum2 = 4 * 1 (A_2) * 2 (X_2) + 4.5 (O_2) * 1 (B_2) * 0 (Y_2) = 8 KB

sum3 = 4 * 4 (A_3) * 0 (X_3) + 2 (O_3) * 1 (B_3) * 1 (Y_3) = 2 KB

S = 8 + 8 + 2 = 18 KB

S + 16 = 34 KB is the required `audio_frmwk_buf_size`.

8. References

[1] *Xtensa XOS Reference Manual* – For Xtensa Tools Version 11.0.3. This is provided as part of the Xtensa tool chain, <TOOLS_INSTALL_PATH>/XtDevTools/downloads/RF-2015.3/docs/xos_rm.pdf.

[2] *HiFi Audio Codec Application Programming Interface (API) Definition*, Ver 1.0. This document is provided as part of this package.

[3] *HiFi Speech Codec Application Programming Interface (API) Definition*, Ver 1.0. This document is provided as part of this package.

[4] *Cadence MP3 Decoder* – Library version 3.18, API version 1.16 (HiFi Mini, HiFi2, HiFi 3 and HiFi 4).

[5] *Cadence MP3 Encoder* – Library version 1.6, API version 1.12 (HiFi Mini, HiFi 2 and HiFi 3). The library needs to be rebuilt from sources for HiFi 4.

[6] *Cadence AMR-WB Decoder* – Library version2.7, API version 1.0 (For HiFi 3 and HiFi 4)

[7] *Cadence AMR-WB Decoder* – Library version 2.3, API version 1.0 (For HiFi Mini and HiFi 2)

[8] *Cadence Sample Rate Convertor* – Library version 1.9, API version 1.12 (HiFi Mini, HiFi 2, HiFi 3 and HiFi 4).

[9] *Cadence AAC Decoder* – Library version3.7, API version 1.16 (HiFi Mini, HiFi 2, HiFi 3 and HiFi 4)

[10] *Cadence Ogg-Vorbis Decoder* – Library version 1.12, API version 1.16 (HiFi Mini, HiFi 2, HiFi 3 and HiFi 4)