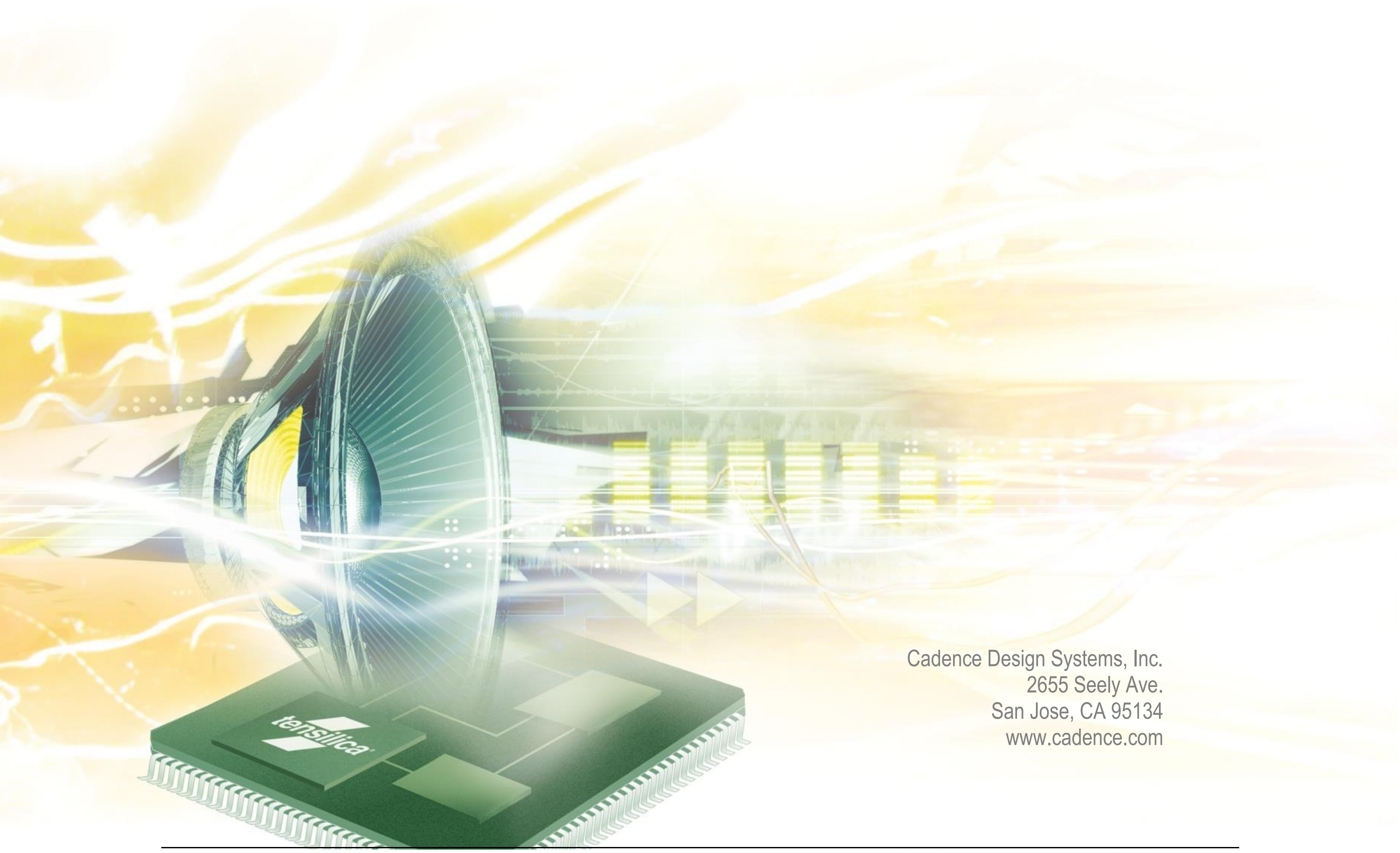
cadence®

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 2.0
January 2020

Contents

Figures

Tables

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	Real-time capturer and renderer components added. Xtensa tool chain v6.0.3 (RF-2015.3) supported only.
1.3	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 1.3.1) 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.
2.0	Added new XAF Developer APIs: xaf_pause, xaf_resume, xaf_disconnect, xaf_probe_start and xaf_probe_stop. Updated prototype for XAF Developer API: xaf_connect. Added support for FreeRTOS in XAF. Added support for pre-emptive scheduling of components in XAF. Added support for Multi-Input, Multi-Output (MIMO) processing class in XAF. Added three samples applications to demonstrate use of new XAF Developer APIs. Updated XAF Architecture details in Section 2. Updated Memory and Timings tables on Xtensa tools chain version RI-2019.2. Added support for Opus encoder plugin component.

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).

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 XAF Developer APIs available for the application developer. Section 4 provides details about building and running a sample application, which illustrates usage of the XAF Developer APIs. Section 5 provides a "How To" guide for adding support for a new component in XAF.

1.2 Xtensa Audio Framework Terminology

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.

Probe: Probe is the XAF mechanism for exporting to application, the processed data of specified ports on each process or execution call of the component.

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 enables the creation of an audio processing chain. It manages the transfer of buffers between components as well as the scheduling of 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.

OSAL APIs: Operating System Abstraction Layer (OSAL) APIs defined to abstract RTOS dependency of XAF through common interfaces.

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

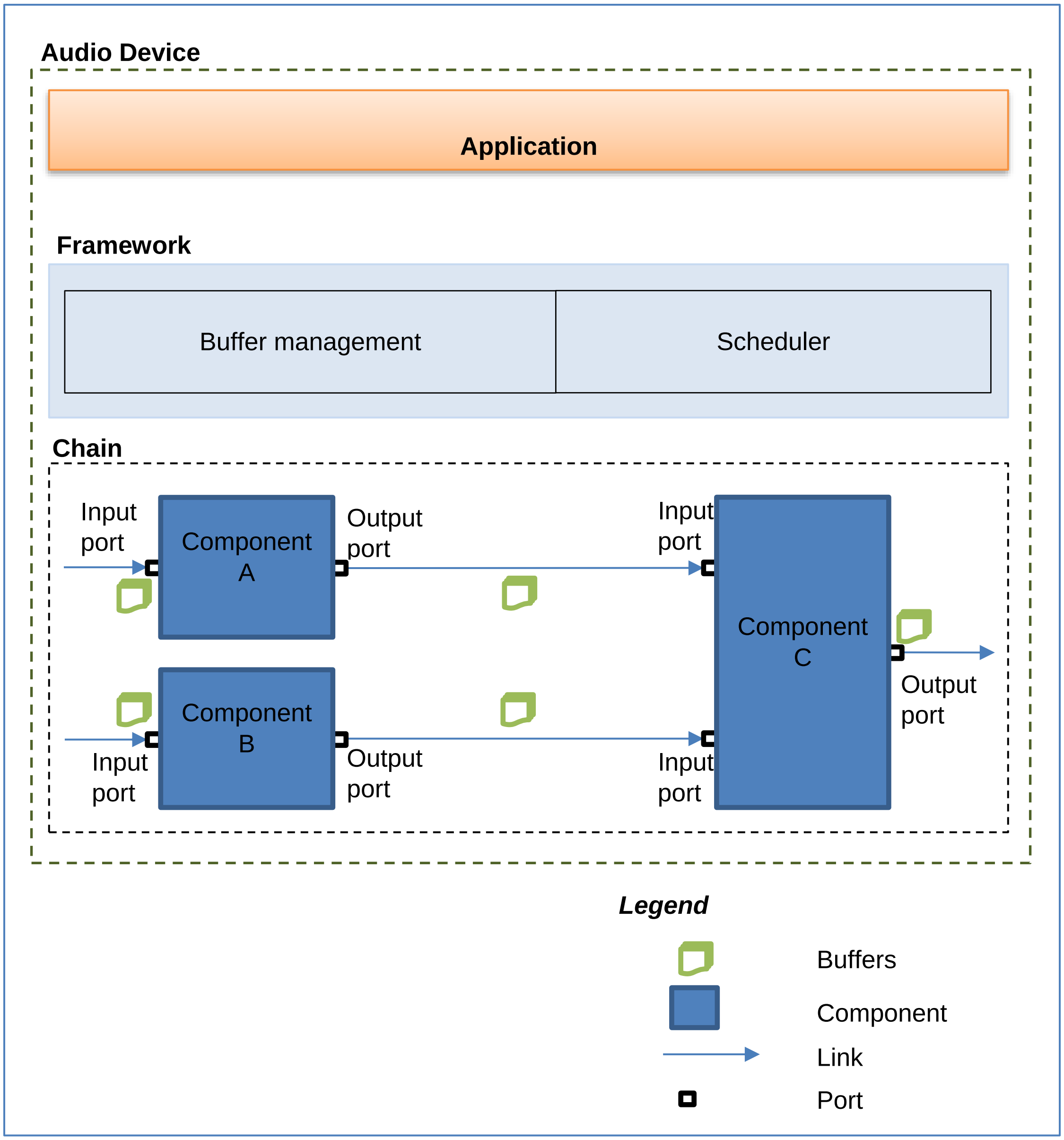


Figure 1-1 XAF Terminology

1.2.2 Port Numbering of Components in XAF

In XAF, port numbering of an audio component starts with 0 for first input port and is incremented for consecutive input ports, followed by output ports.

A component with **n** input ports and **m** output ports has port numbering as shown in Figure 1-2.

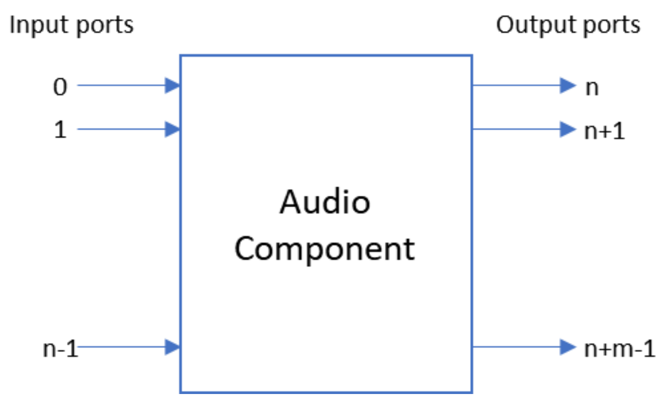


Figure 1-2 Port Numbered Audio Component

1.3 Xtensa Audio Framework Specifications

This section provides XAF specifications.

1.3.1 Feature Set

API features:

- Ability to create components and connect them in a processing chain.
- Ability to read and write component configuration parameters.
- Ability to read component status and trigger component processing.
- Ability to pause and resume ports of components in a chain at runtime.
- Ability to disconnect and delete or re-connect components in a chain at runtime.
- Ability to probe components at runtime.
- Ability to prioritize components for execution.

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 hardware resources such as available memory/MHz and not by XAF.
- Manages the allocation of memory for data buffers for sharing data between application and audio components as well as between any two connected audio components.
- Manages the allocation and deallocation 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. Note, as XAF merely transfers the data between components, application programmer should ensure data format compatibility (sample rate, number of channels, PCM width) between connected components.
- Allows for prioritization of components for execution. At runtime, component instances with higher priority will preempt processing performed by components with lower priority. This feature is useful to ensure timely execution of components with real-time behavior (for e.g. microphone capture or speaker playback).
- Various component types supported (see Table 2-1), depending on the number of ports and the type of data transferred across the ports (PCM or non-PCM).

Example applications in XAF package:

- Fifteen test applications are provided to demonstrate various use-cases.
- Example code to demonstrate the integration of seven Cadence audio libraries (MP3 decoder, MP3 encoder, AMR-WB decoder, Sample Rate Convertor, AAC decoder, Ogg-Vorbis decoder, Opus encoder) into XAF is included in this package. Note that the actual audio libraries must be licensed separately and are not part of this package.
- Optional support for trace prints and cycles profiling is provided for detailed analysis of XAF execution.

Supported configurations:

- HiFi Cores: HiFi 3, HiFi 4, HiFi 5
- Xtensa Tools Chain: Version RI-2019.2
- RTOS: Cadence XOS [1] or FreeRTOS [12] (see details in Section 2.1.3)

Note	XAF is only tested with supported configurations mentioned above and it must be used with one of the supported configuration combinations.

Limitations:

- Only one instance of XAF can run at a time.
- XAF does not support fast functional “TurboXim” mode of Instruction Set Simulator (ISS). ISS must be used in cycle accurate mode with XAF.
- In current version of XAF, only one (first) input port can receive input data from application and only one (first) output port can send output data to application; that is, edge component cannot have multiple input ports or output ports connected to application.

1.4 Xtensa Audio Framework Performance

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

1.4.1 Memory

Table 1-1 Library Memory

Text (Kbytes)			Data (Kbytes)
HiFi 3	HiFi 4	HiFi 5	
35.7	38.3	43.3	0.7

Note Other than for Text and Data, XAF uses 3.1 Kbytes for bss. The measurements exclude the memory required by RTOS and the standard C library. The measurements are done with Version RI-2019.2 of the Xtensa tool chain with XOS.

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 three categories:

1. Local memory allocated by XAF for use by audio components: This is the memory that is allocated by XAF for usage by audio components and it is controlled by `audio_comp_buf_size` parameter passed to the `xaf_adev_open()` function.
2. Shared memory allocated by XAF for communication between application and audio components: This is the memory allocated by XAF to transfer data and messages between application and audio components and it is controlled by `audio_frmwk_buf_size` parameter passed to the `xaf_adev_open()` function.
3. Memory used by XAF structures: This memory is allocated by XAF for its internal data structures.

Table 1-2 shows the runtime memory allocated by XAF for Testbench 1 as shown in Figure 4-1 (a simple processing chain consisting of single PCM gain component).

Table 1-2 Runtime Memory

No	Memory breakup	RAM (Kbytes)		
		HiFi 3	HiFi 4	HiFi 5
1	Local memory allocated by XAF for use by audio components	76.4	76.4	76.4
2	Shared memory allocated by XAF for communication between application and audio components	28.0	28.0	28.0
3	Memory used by XAF structures	22.8	22.8	22.8
	Total	127.2	127.2	127.2

Note The measurements are done with Version RI-2019.2 of the Xtensa tool chain.

Note For Testbench 1, `audio_frmwk_buf_size` = 64 KB and `audio_comp_buf_size` = 128 KB are passed during `xaf_adev_open()` call. The actual memory used by XAF for Testbench 1 processing chain is mentioned above.

1.4.2 Timings

Table 1-3 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 is done for Testbench 1 as shown in Figure 4-1 (a simple processing chain consisting of single PCM gain component) with XOS.

Table 1-3 MCPS

Use Case		Average CPU Load (MHz)		
		HiFi 3	HiFi 4	HiFi 5
Testbench 1 – PCM Gain (Mono, 44.1KHz, Buffer size = 4096 samples)	XAF	0.8	0.6	0.6
	Total	4.1	2.9	3.0

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 HiFi 3/HiFi 4/HiFi 5 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 are done with Version RI-2019.2 of the Xtensa tool chain with XOS.

2. Xtensa Audio Framework Architecture Overview

2.1 Application Software Architecture with Xtensa Audio Framework

Figure 2-1 shows various building blocks of application software based on XAF. Note that in this figure the application, RTOS, and audio components are not part of XAF. These building blocks are briefly described in the following sections.

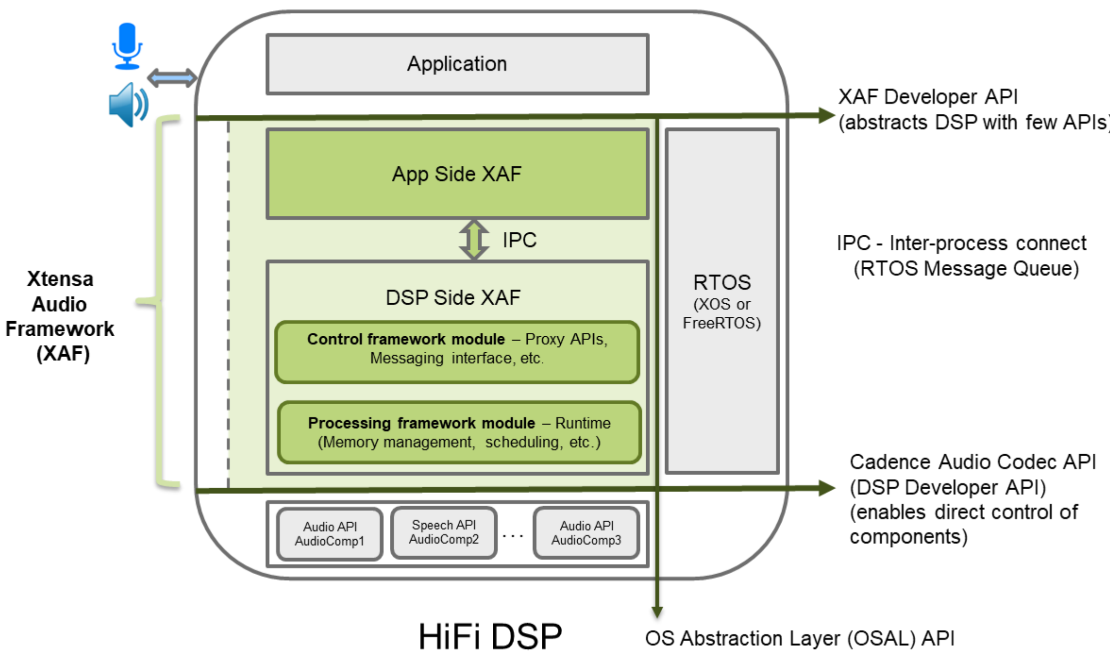


Figure 2-1 Application Software Stack Diagram

2.1.1 Application

In the application, an application developer will leverage the XAF Developer APIs to create a processing chain. The XAF Developer API is the interface between the application and XAF, and it enables chains to be set up, configured, and run. XAF Developer APIs also can be used to control and modify the processing chains at runtime.

Note that XAF allows an unlimited number of components in the audio processing chain — the limitation is only from the system hardware. The application developer must ensure that there is enough memory and CPU bandwidth available on the hardware. Figure 2-2 shows an example music playback processing chain that can be created using XAF. Fifteen sample applications (testbenches) are provided with XAF package, which implement fifteen different audio processing chains. Details of these sample applications are described in Section 4.

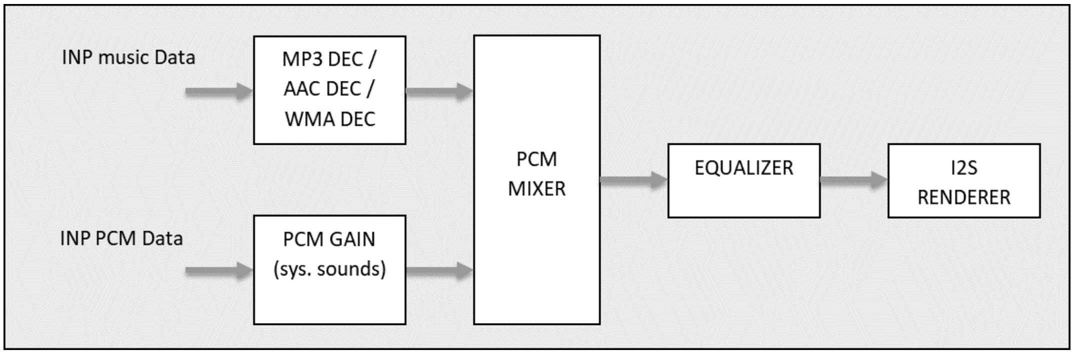


Figure 2-2 Example Music Playback Processing Chain

2.1.2 Xtensa Audio Framework

Xtensa Audio Framework (XAF) is responsible for creating, configuring, and running the processing chains through XAF Developer API. Memory management of components, data movement between components, and scheduling of components is all done by XAF internally and is completely abstracted from the application.

As shown in Figure 2-1, XAF architecture includes three major building blocks: App Side XAF, Inter-Process Connect (IPC), and DSP Side XAF.

App Side XAF:

App Side XAF is responsible for building and maintaining audio processing chains as per application need. There is no actual audio processing done at this side. Instead, it is a control code that runs in application thread context at highest priority with regard to the other two building blocks. App Side XAF manages the operation of underlying DSP Side XAF by sending commands and receiving responses from it.

IPC:

Inter-Process Connect (IPC) is the communication link between App Side XAF and DSP Side XAF. It passes commands and responses between two sides and it has no knowledge about information being passed. It runs in a separate thread context at higher priority than DSP Side XAF.

DSP Side XAF:

DSP Side XAF does the actual audio processing based on commands received from App Side XAF, and sends responses back to App Side XAF after command completion. Based on commands received from App Side XAF, it creates, configures, and connects components to create processing chain and executes the components to perform audio processing. DSP Side

XAF runs in a separate thread context at lowest priority with regard to the other two building blocks. By default, in DSP Side XAF, all components execute in the single thread context at same priority and there is no pre-emption of one component execution by another. For advanced applications, some components may be required to execute at higher priority than others and it is supported in XAF by a separate developer API (see Table 3-16 for details). Note, in this case multiple DSP Side worker threads will be created based on number of different priority components. An example application for pre-emption could be where capturer and renderer components are configured with higher priority so that processing of captured microphone data or playback of output PCM data is done in timely fashion without any gaps.

2.1.3 RTOS

XAF uses RTOS to create multiple threads required for its functioning as described in section 2.1.2. The application may also require threads to feed input and/or consume output data for components connected to it. Also, Inter-Process Connect is implemented using RTOS message queues and mutex. Cadence XOS [1] and Xtensa port of FreeRTOS V10.2.0 [12] are supported with XAF. Operating System Abstraction Layer (OSAL) is defined for all RTOS functionality requirements in XAF. The OSAL APIs are described in Section 8.

Note	XOS is released with the Xtensa tools SDK and is not a part of the XAF release package.
Note	Xtensa port of FreeRTOS is not a part of the XAF release package. See Section 4.5 for details of downloading and building FreeRTOS for XAF.

2.1.4 Audio Components

Audio components are the actual data processing modules. XAF interacts with audio components using Cadence Audio Codec API (DSP Developer API). Cadence Audio Codec API are described in detail in [2]. Section 5 contains details on how to add a new audio component in XAF. Table 2-1 lists various audio component types supported by XAF in the current release. Component types are defined by data processing functionality and number of input and output ports.

Table 2-1 Audio 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.

Component Type	Input		Output		Component Description
	Ports	PCM	Ports	PCM	
Post processing	1	Y	1	Y	Post-processes input PCM data to generate output PCM data.
Renderer	1	Y	1[1]	NA	Plays input PCM data to a speaker/headphone.
Capturer	0	NA	1	Y	Captures output PCM data from a microphone.
MIMO	4[2]	Y	4[3]	Y	Multi-Input Multi-Output (MIMO) component process input PCM data to generate output PCM data.

2.2 Internal Architecture Details of Xtensa Audio Framework

This section deep dives into internal architecture and implementation details of XAF.

2.2.1 Control and Data Flow in XAF

As briefly discussed in section 2.1.2, XAF architecture includes three major building blocks: App Side XAF, Inter-Process Connect (IPC), and DSP Side XAF. App Side XAF and DSP Side XAF pass control and data using commands and responses through Inter-Process Connect as shown in Figure 2-3.

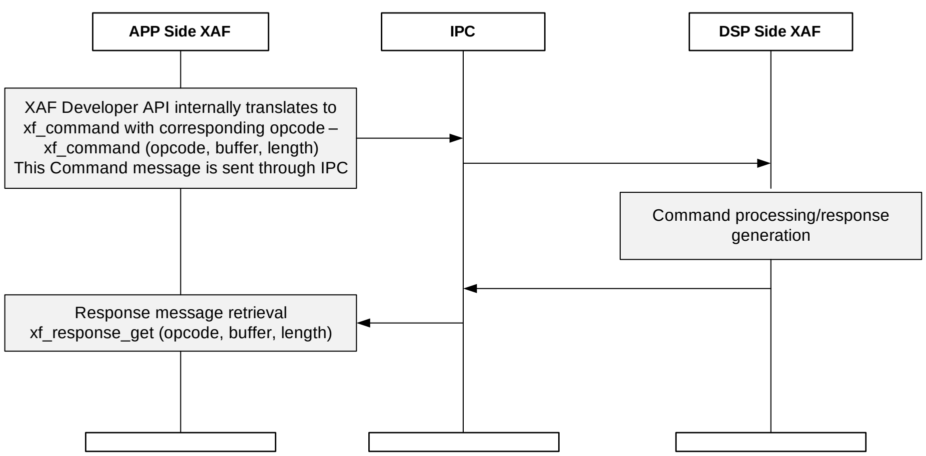


Figure 2-3 XAF Command and Response Flow

[1] Renderer component has one optional output port (can be used as feedback path for echo cancellation).

[2] Maximum number of input ports for MIMO components is 4.

[3] Maximum number of output ports for MIMO component is 4.

All of the XAF Developer API calls except `xaf_comp_process` and `xaf_probe_start` API calls are blocking or synchronous i.e. the API call would wait for response from DSP Side XAF for command completion. Example of synchronous XAF Developer API is `xaf_comp_set_config` API (see Table 3-6 for details of API). Figure 2-4 shows control flow sequence for the same.

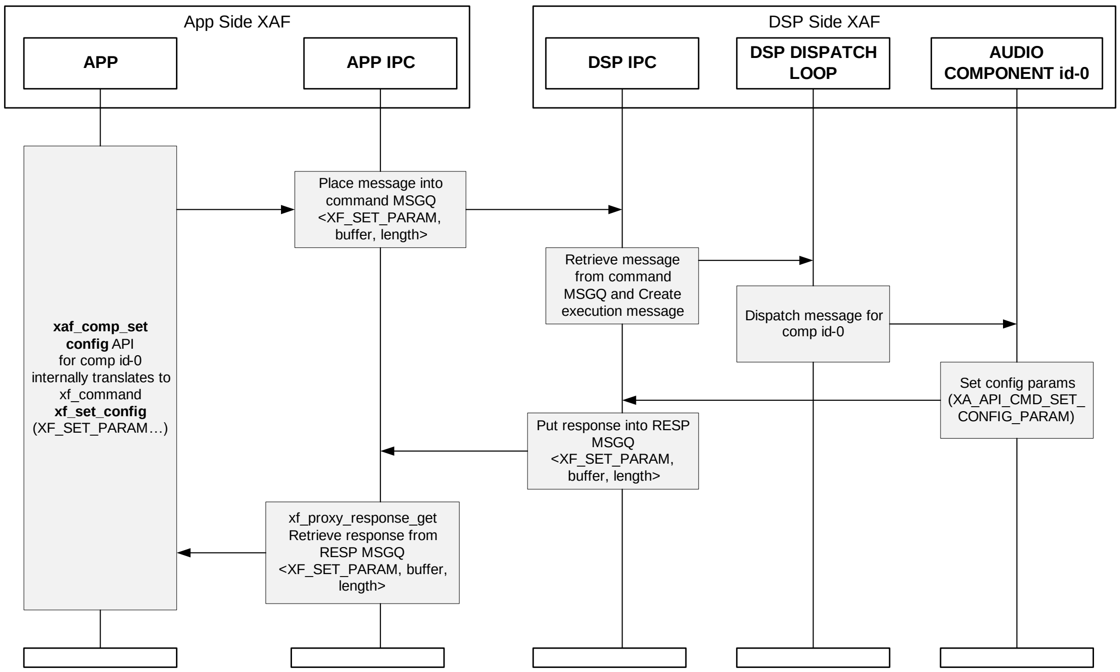


Figure 2-4 XAF Developer API `xaf_comp_set_config` Control Flow

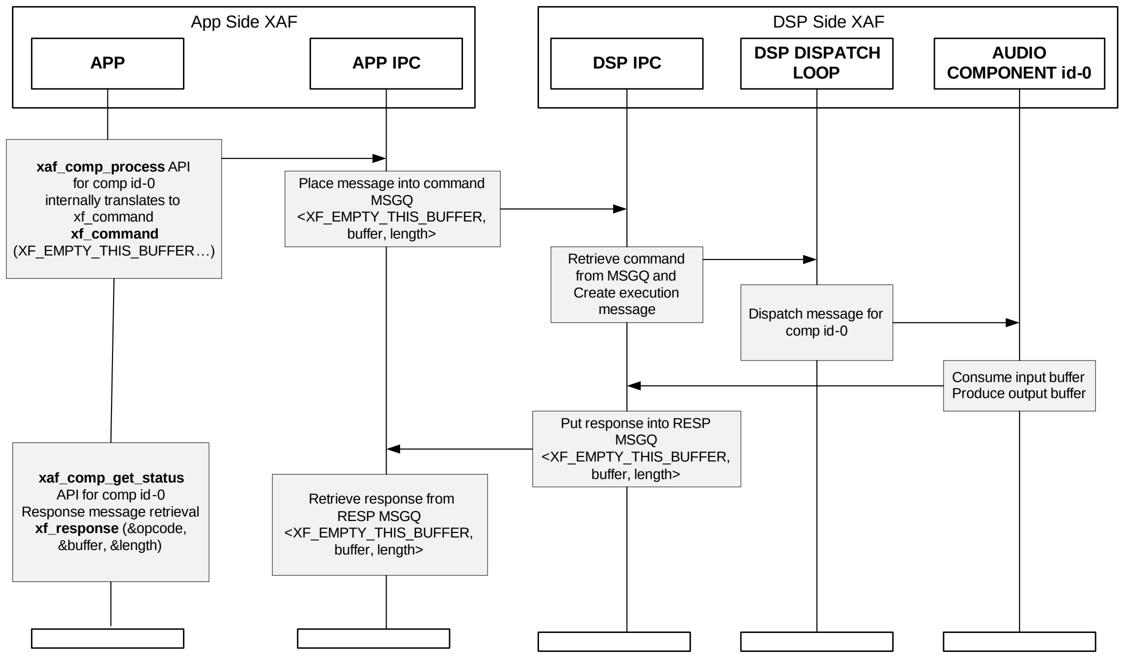


Figure 2-5 XAF Developer API `xaf_comp_process` Control Flow

XAF Developer APIs `xaf_comp_process` (see Table 3-10 for details of API) and `xaf_probe_start` (see Table 3-14 for details of API) are non-blocking or asynchronous. Specifically, the API call would not wait for response from DSP Side XAF for command completion, rather response from DSP Side XAF can be queried for by `xaf_comp_get_status` API (see Table 3-11 for details of API) at any later point of time. Figure 2-5 shows control flow sequence for these API calls where application feeds input data to audio component id-0. When audio component id-0 consumes the input data, it sends the response to application. Note, `xaf_comp_get_status` API call would block if there is any pending response on the component.

Audio components connected with each other on DSP Side XAF also use commands and responses to share data with each other through local message queue. Note, this local message queue is internal and different from the one between App Side XAF and DSP Side XAF. The audio component communication is shown in Figure 2-6 where application feeds input data to audio component id-0, which is then connected to audio component id-1 and output of audio component id-1 is sent back to application.

Note, Figure 2-5 and Figure 2-6 do not show all transactions for simplification and ease of understanding.

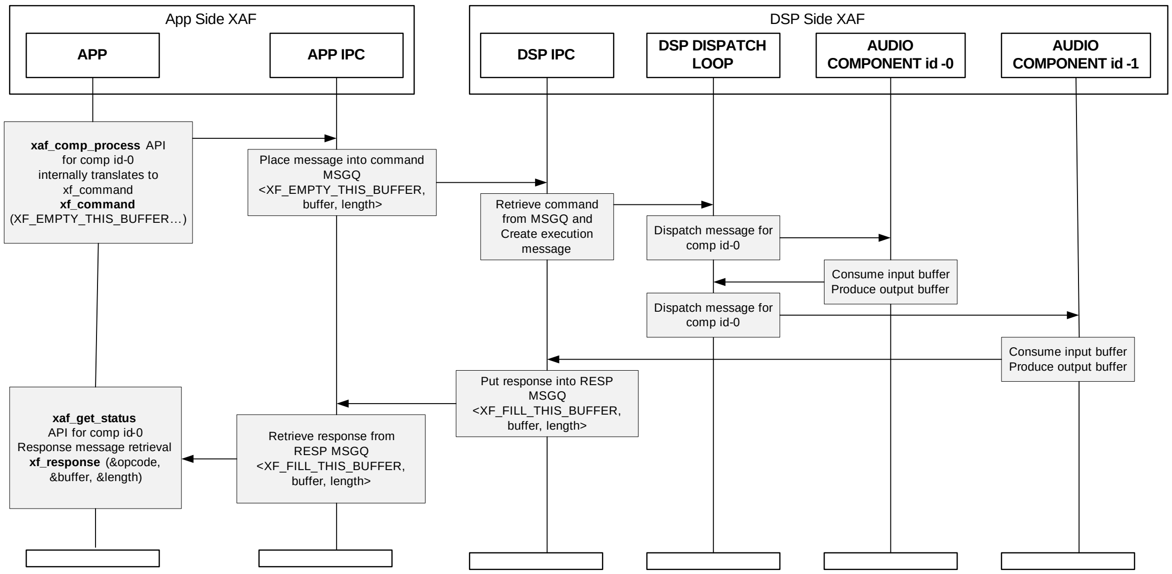


Figure 2-6 XAF Control Flow Between Audio Components

2.2.2 DSP Side XAF Details

DSP Side XAF uses object-oriented class like architecture for managing, scheduling, and executing various audio components as shown in Figure 2-7. Generic base class provides the functionality common to all components (for example, memory allocations or deallocations). Various derived classes that inherit the base class are defined based on input-output ports and data processing pattern of components. Each derived class implementation defines handling of input and output data on its IO ports. It also defines pause, resume, connect, and disconnect functionality for the class. The following derived classes are defined in current version of XAF.

- Audio Codec Class – Supports components with one input port and one output port. Suitable for audio decoders, encoders and pre/post-processing modules.
- Mixer Class – Supports components with maximum four input ports and one output port. Defined for mixer components.
- Multi-Input Multi-Output (MIMO) Class – Supports components with multiple input ports and multiple output ports. Suitable for PCM processing modules with multiple input, output ports such as PCM Splitter, Acoustic Echo Canceler, etc. Maximum number of input or output ports is defined to four in current version of XAF.
- Capturer Class – Supports components with zero input port and one output port. Defined for microphone capture modules.
- Renderer Class – Supports components with one input port and zero or one optional output port. Defined for speaker playback modules. Optional output port is defined for feedback or reference data which can be used for echo cancellation.

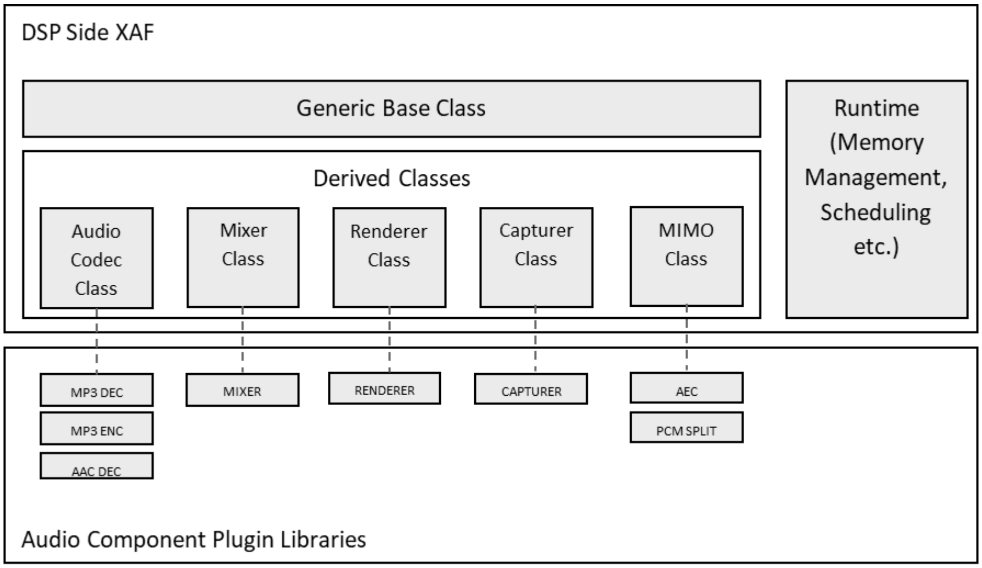


Figure 2-7 DSP Side XAF Audio Component Architecture

The generic base class and derived class use Cadence Audio Codec API to interact with audio component plugins, hence it is required that any audio component for XAF must support Cadence Audio Codec API. Note, actual component plugin libraries are not part of XAF and must be provided to the application at link time.

Each derived class implements process or execution function for its components with a three-step function:

- First step is pre-process, which prepares input and output ports for execution
- Second step is actual processing of data by the component plugin library
- Third step is post-process, which manages input and output data after execution

Figure 2-8 shows process function for Audio Codec Class with highlighting calls made to audio component plugin library using Cadence Audio Codec API. Note, pre-process also passes input-over message to component plugin library when input is over, and post-process also flushes output ports when execution-complete message is received from component plugin library. EDF scheduling policy used in post-process for rescheduling of the component is described in Section 2.2.3.

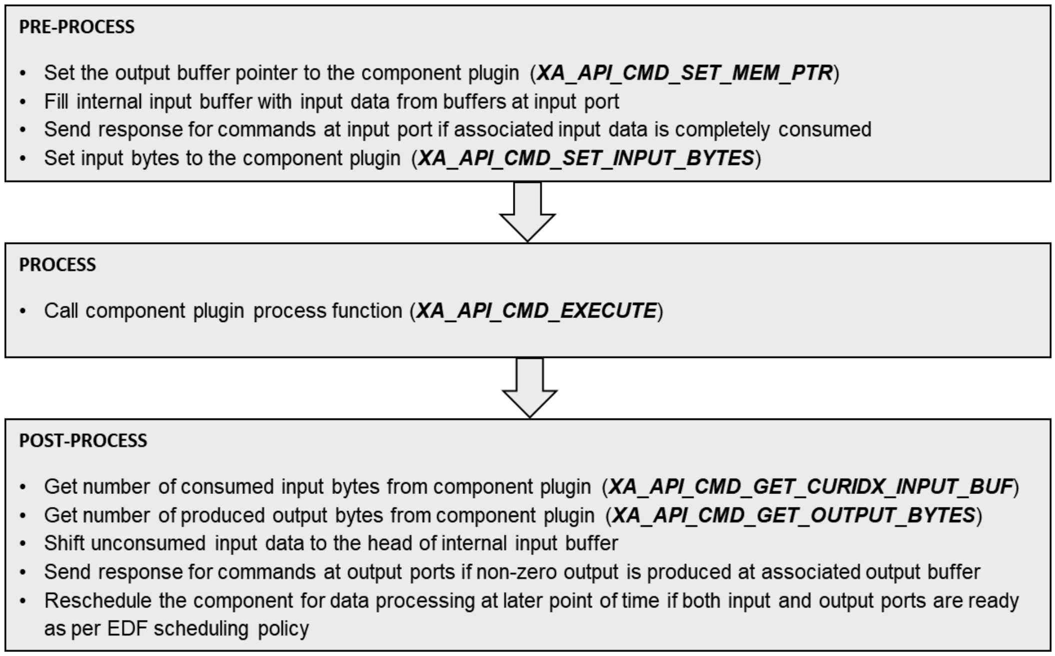


Figure 2-8 XAF Audio Codec Class Process Sequence

2.2.3 Audio Component Management

To explain XAF audio component IO buffer management, scheduling, etc., this section uses a simple audio processing pipeline where PCM Gain component (applies gain on input PCM data) receives input data from application and is connected to MP3 Encoder, and output of MP3 decoder is sent back to application. When PCM Gain component is created with two input buffers to receive data from application and MP3 Encoder is created with one output buffer to send data back to application, various buffers will be allocated in XAF as shown in Figure 2-9.

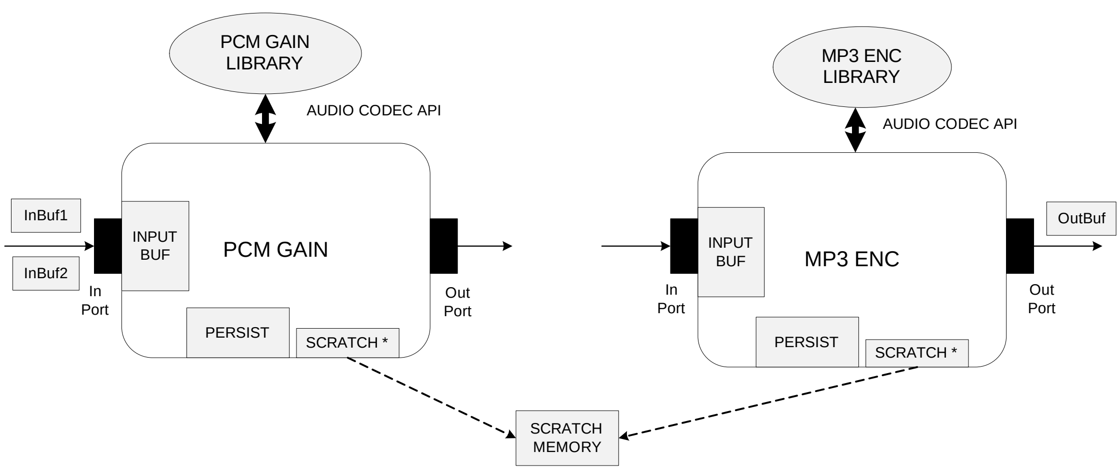


Figure 2-9 XAF Audio Components at Creation

Both PCM Gain and MP3 Encoder components have one input port and one output port, and are created as Audio Codec Class components. One internal input buffer and one internal persistent buffer is always allocated for each component. In this example, it is assumed that both components are at same priority, hence they run in the same thread context and share the scratch buffer. Note, XAF requires scratch memory size to be biggest of scratch memory requirement of all components running in the same thread context (i.e. same priority). The sizes of input, output, persistent and scratch buffers are queried from component library by XAF using Cadence Audio Codec API. Note, no output buffer is allocated for PCM Gain component yet.

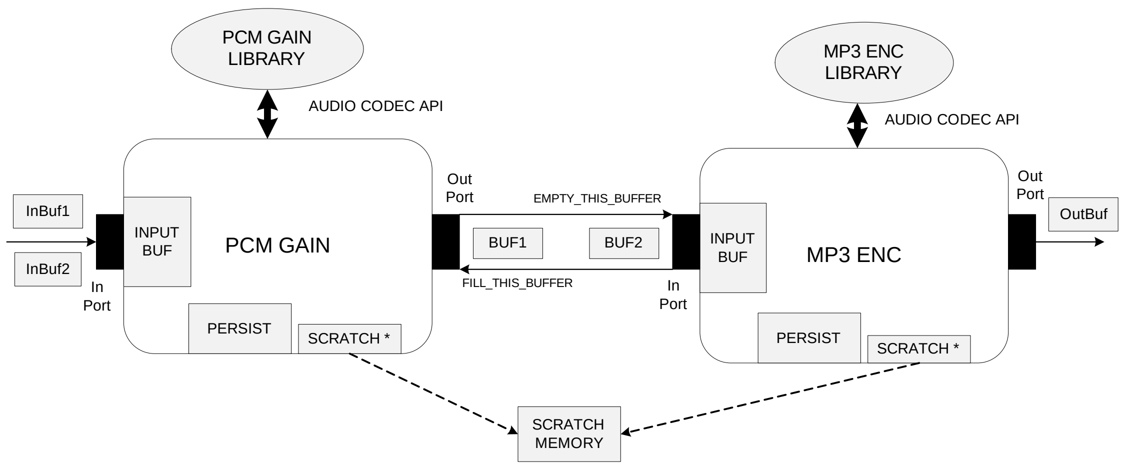


Figure 2-10 XAF Connected Audio Components

When PCM Gain component output port is connected to MP3 Encoder input port using `xaf_connect` API with two buffers (see Table 3-8 for API details), connect buffers are allocated by XAF (BUF1 and BUF2) as shown in Figure 2-10. The size of these two buffers would be equal to output buffer size requirement of PCM Gain component.

Note in XAF, when buffer arrives at input port of a component either from preceding component or application, input data is always copied into component's internal input buffer and during processing, output data is always produced in the received output buffer at output port either from succeeding component or application. Buffer arrived at input port is sent back only after all input data is consumed and buffer received at output port is sent back whenever output data of non-zero size is produced in it.

XAF uses 'Earliest Deadline First (EDF)' scheduler to manage scheduling of various audio components in the processing chain. When input port is ready (input data is available at input port) and when output port is ready (output buffer is available at output port), the component is scheduled for data processing or execution. Each execution of component would consume some input data and produce some output data. After one execution if input and output ports are still ready, the component would be scheduled for next execution at later point of time based on its next deadline. The timestamp computed using output PCM samples produced or input PCM samples consumed and sample rate of data is used as measure of deadline by EDF scheduler in XAF.

With XAF, audio components with different frame sizes can be seamlessly connected with each other at application level. XAF internal design with EDF scheduler manages audio components operating with different frame sizes. For example, if PCM Gain component processes 1024 PCM samples in one execution and MP3 encoder processes 4096 samples in one execution in Figure 2-10 above, PCM Gain would get scheduled and executed four times for each execution of MP3 Encoder automatically in XAF.

3. Xtensa Audio Framework Developer APIs

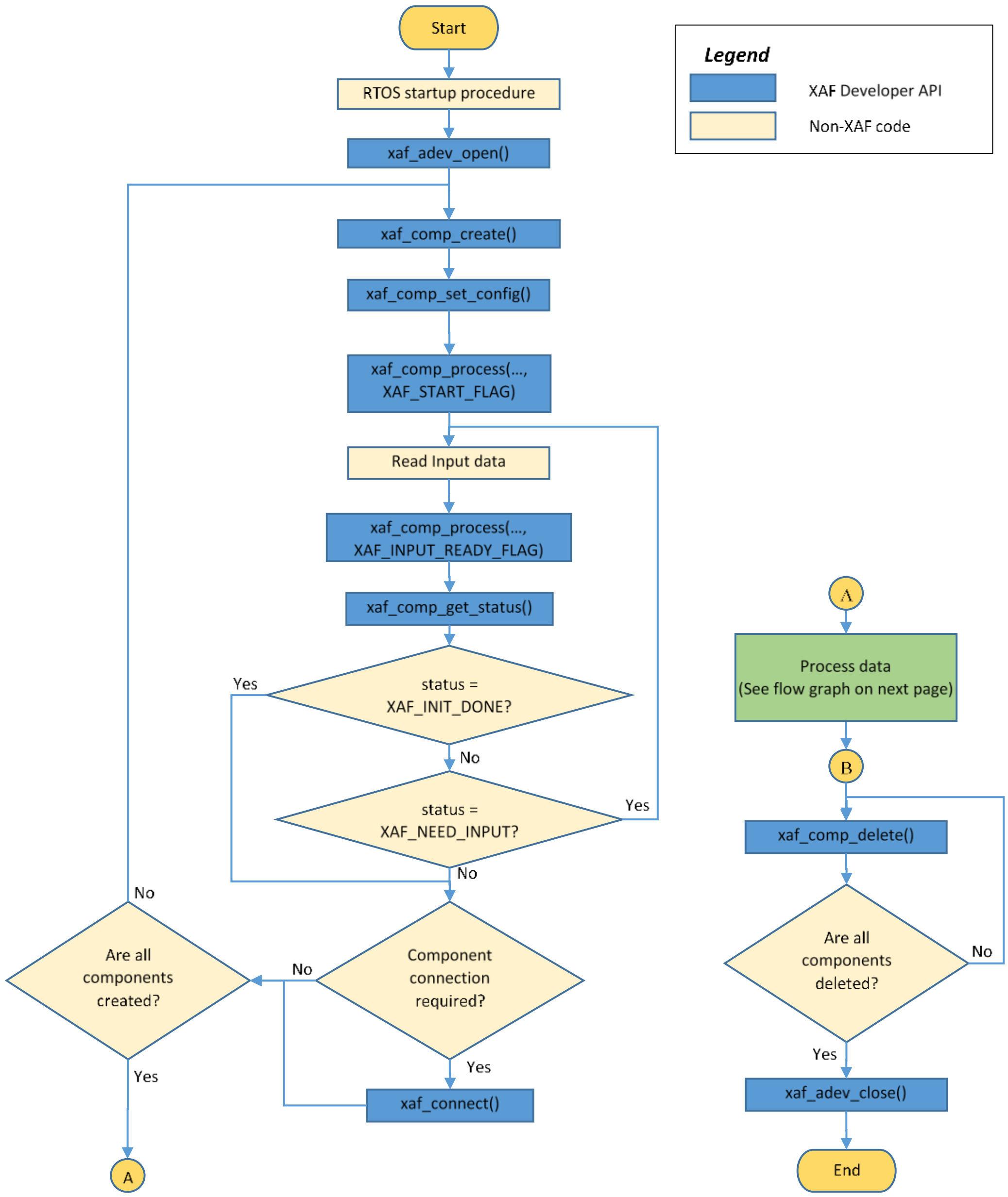
This section discusses XAF Developer APIs that are available for the application programmer to create, configure, and run audio processing chains.

XAF Developer APIs are summarized in Table 3-1.

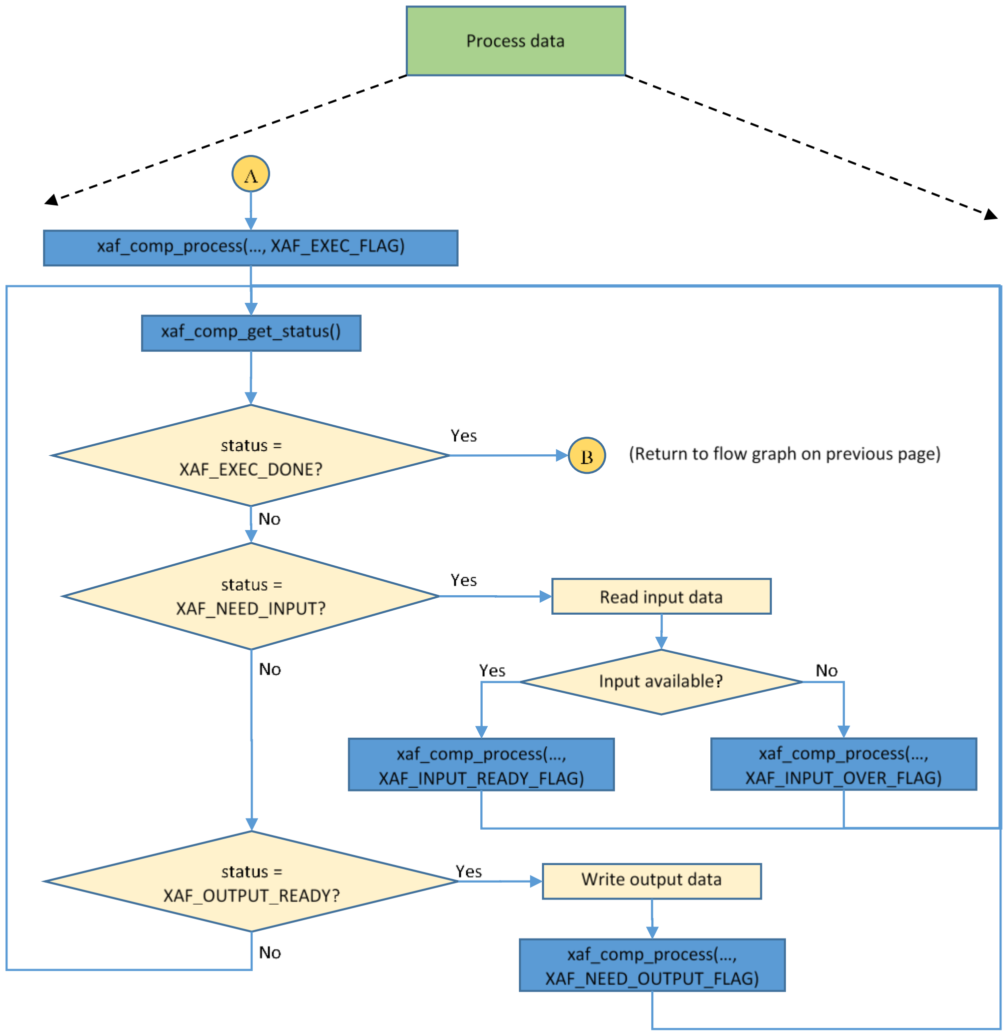
Table 3-1 XAF Developer APIs

API Type	XAF Developer API	Can be called at runtime?
Startup API	xaf_adev_open	No
	xaf_comp_create	Yes
Configuration API	xaf_comp_set_config	Yes
	xaf_comp_get_config	Yes
	xaf_adev_set_priorities	No
Connect API	xaf_connect	Yes
	xaf_disconnect	Yes
Process API	xaf_comp_process	Yes
	xaf_comp_get_status	Yes
Control API	xaf_pause	Yes
	xaf_resume	Yes
Probe API	xaf_probe_start	Yes
	xaf_probe_stop	Yes
Closure API	xaf_adev_close	No
	xaf_comp_delete	Yes
Information API	xaf_get_verinfo	Yes
	xaf_get_mem_stats	Yes

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`. The components are initialized using `xaf_comp_process` with `XAF_START_FLAG` flag and connected using `xaf_connect`.

  Note: Audio decoder components require input data during initialization to find out input stream parameters such as sample rate, number of channels. So the initialization loop shown in Figure 3-1 (a) that feeds input data to component during initialization is required only for audio decoder components, and such loop is not required for encoder or PCM data processing components.
- 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.
- The following features are available in XAF at runtime:
  - Pause or resume ports: Consumption or production of data on a port can be paused by using `xaf_pause` API. A paused port can be resumed by using `xaf_resume` API.
  - Probe components: Probing of data on input and/or output ports of a component can be started by using `xaf_probe_start` API and probing can be stopped by using `xaf_probe_stop` API.
  - Disconnect and reconnect components: Any connected output ports of a component can be disconnected dynamically by using `xaf_disconnect` API. Components also can be connected or reconnected dynamically by using `xaf_connect` API.

3.1 Files Specific to XAF Developer APIs

XAF Developer API Header File (/include/)

- `xaf-api.h`

3.2 XAF Developer API-Specific Error Codes

The errors in this section can result from the XAF Developer API call 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 XAF 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 XAF Developer API.

- XAF_RTOS_ERROR

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

- XAF_API_ERR

  This error code generally indicates that the XAF Developer API is called out of order, for example, `xaf_comp_create()` is called before `xaf_adev_open()`. Note, this error is also returned if incorrect response is received from DSP Side XAF for command sent by the XAF Developer API.

3.2.2 Specific Errors

The following errors are specific to some APIs.

- XAF_ROUTING_ERROR

  This error code indicates that the XAF Developer API `xaf_connect()` or `xaf_disconnect()` was unsuccessful to connect or disconnect the two requested components.

- XAF_STATUS_TIMEOUT

  This error code is returned if XAF Developer API `xaf_comp_get_status()` does not receive pending response from DSP Side XAF within defined wait time limit. The maximum wait time is defined by `MAXIMUM_TIMEOUT` (`10000 ms`) in current version of XAF.

3.3 XAF Developer APIs

This section contains tables describing the XAF Developer APIs.

Table 3-2 `xaf_adev_open` API

API	`XAF_ERR_CODE xaf_adev_open(pVOID *pp_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 which is a parent structure for all XAF operations. It starts the processing thread that performs all audio processing on DSP Side XAF and starts the IPC thread. It also allocates local memory to be used by the audio components on DSP Side XAF and shared memory for communication between Host Side XAF and DSP Side XAF.

Actual Parameters	`pp_adev`
	Address of pointer to audio device. This API call would allocate memory for audio device and update this pointer with it.
	
	`audio_frmwk_buf_size`
	Size of memory to be allocated for shared buffers and structures between Host Side XAF and DSP Side XAF. This size must be aligned to 32 bytes and greater than or equal to 16 kB (for XAF structures). Refer to Section 7 for more details on memory guidelines.
	
	`audio_comp_buf_size`
	Size of memory to be allocated for various audio component buffers and structures required locally on DSP Side XAF. This size must be aligned to 32 bytes and greater than or equal to 73 kB (includes 56 kB for scratch and 17 kB for XAF structures). Refer to Section 7 for more details on memory guidelines.
	
	`mem_malloc`
	Function pointer to the memory allocation routine to be used by XAF. This routine must have prototype as shown below where the ‘`id`’ indicates whether the memory is allocated for audio device (`DEV_ID`) or for audio components (`COMP_ID`).
	`pVOID mem_malloc(WORD32 size, WORD32 id);`
	
	Note: XAF expects that `mem_malloc` should return a 4-byte aligned address.
	
	`mem_free`
	Function Pointer to the memory free routine to be used by XAF. This routine must have prototype as shown below.
	`VOID mem_free(pVOID ptr, WORD32 id);`

Restrictions	**Prerequisite:** The RTOS startup procedure needs to be invoked before calling this function. Procedures for XOS and FreeRTOS are as following. For XOS: 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_rtos()under #if defined (HAVE_XOS) in the file test/src/xaf-utils-test.c for an example. For FreeRTOS: The start-up procedure for FreeRTOS involves starting the main thread and starting the scheduler by calling the function `vTaskStartScheduler()`. Refer to the function `init_rtos()` under `#ifdef HAVE_FREERTOS` 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 3-3 `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 and IPC thread execution.

Actual Parameters	`p_adev`
	Pointer to the audio device
	
	`flag`
	`XAF_ADEV_FORCE_CLOSE`: Forces close of 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 3-4 `xaf_comp_create` API

API	`XAF_ERR_CODE xaf_comp_create(pVOID p_adev,`
	`pVOID *pp_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 audio 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 for details.

Actual Parameters	`p_adev`
	Pointer to the audio device structure
	
	`p_comp`
	Address of pointer to the audio component structure
	
	`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 for details on how to add a new audio component in XAF).
	
	`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.
	
	`comp_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
	`XAF_MIMO_PROC_12:` — MIMO component with 1 input and 2 output ports
	`XAF_MIMO_PROC_21:` — MIMO component with 2 input and 1 output ports
	`XAF_MIMO_PROC_22:` — MIMO component with 2 input and 2 output ports
	`XAF_MIMO_PROC_23:` — MIMO component with 2 input and 3 output ports
	`XAF_MIMO_PROC_10:` — MIMO component with 1 input and 0 output ports
	`XAF_MIMO_PROC_11:` — MIMO component with 1 input and 1 output ports

Restrictions	Should not be called before `xaf_adev_open` API

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 3-5 `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` API. Should not be called while application has thread waiting for pending responses from the component. Should be called once all the application threads have exited under normal execution conditions (after `__xf_thread_join` API). To force close the device, `xaf_adev_close` API with `XAF_ADEV_FORCE_CLOSE` flag should be used.

Example

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

Errors

- Common API Errors

Table 3-6 `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. Note, this API can also set (write) three configuration parameters to the XAF. These three parameters are discussed in detail in Section 3.4.

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` API. Each parameter value must be of size 4 bytes.

Example

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

Errors

- Common API Errors

Table 3-7 `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 value received from audio component.

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` API. Each parameter value is of size 4 bytes.

Example

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

Errors

- Common API Errors

Table 3-8 `xaf_connect` API

API	`XAF_ERR_CODE xaf_connect(pVOID p_src,`
	`WORD32 src_out_port,`
	`pVOID p_dest,`
	`WORD32 dest_in_port,`
	`WORD32 num_buf)`

Description

This API connects the output port `src_out_port` of audio component `p_src` to the input port `dest_in_port` of audio component `p_dest` with `num_buf` connect buffers between them. The size of each connect buffer will be equal to the size of the output buffer of `p_src`.

For port numbering convention, refer to Section 1.2.2.

For MIMO Class components, `xaf_connect` API call passes the output port connect information to component plugin through `XA_MIMO_PROC_CONFIG_PARAM_PORT_CONNECT` configuration parameter.

This API will fail if it is called for an invalid port or already connected port. Audio components have input and output ports as follows. Note, renderer component has one optional output port (can be used as feedback path for echo cancellation).

Component Type	Input Ports	Output Ports
`XAF_DECODER` or `XAF_ENCODER` or `XAF_PRE_PROC` or `XAF_POST_PROC`	1	1
`XAF_MIXER`	4	1
`XAF_RENDERER`	1	1 (optional)
`XAF_CAPTURER`	0	1
`XAF_MIMO_PROC_12`	1	2
`XAF_MIMO_PROC_21`	2	1
`XAF_MIMO_PROC_22`	2	2
`XAF_MIMO_PROC_23`	2	3
`XAF_MIMO_PROC_10`	1	0

Processing frame sizes of connecting components should be considered for choosing number of connect buffers. For example, higher number of connect buffers between source component of very small frame size and destination component of higher frame size would reduce framework overhead cycles. If pre-emptive scheduling is enabled, priority of source component should also be considered for choosing number of connect buffers. For example, if capturer source component at higher priority is producing output data at every 1 millisecond and processing time of destination AEC component is 3 milliseconds, the connect buffers should be at least 3 in this case.

Actual Parameters	`p_src`
	Pointer to the source audio component structure
	
	`src_out_port`
	Output port number of `p_src` audio component
	
	`p_dest`
	Pointer to the destination audio component structure
	
	`dest_in_port`
	Input port number of `p_dest` audio component
	
	`num_buf`
	Number of connect buffers to be added between components
	Valid values: 2 to 1024

Restrictions	Should not be called before at least two audio components are created using `xaf_comp_create` API and source component has been initialized.

Example

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

Errors

- Common API Errors
- XAF_ROUTING_ERROR

  Indicates that the API failed to connect the two requested components (due to invalid port numbers, already connected ports, or uninitialized source audio component, etc.)

Table 3-9 `xaf_disconnect` API

API	`XAF_ERR_CODE xaf_disconnect(pVOID p_src, WORD32 src_out_port, pVOID p_dest, WORD32 dest_in_port)`
Description	This API destroys the data link between output port `src_out_port` of audio component `p_src` and input port `dest_in_port` of audio component `p_dest` by deallocating data buffers and message pool created during `xaf_connect` API call. Any unprocessed data between the ports would be dropped during disconnect. This API has Class specific implementation as described below. **Audio Codec Class:** **Mixer Class:** **Capturer Class:** Audio Codec Class or Mixer Class or Capturer Class component has only one output port. `xaf_disconnect` API call on its output port would cancel any pending processing of the component, flush the output port (drop unprocessed data between ports) and free buffers and message pool between ports. **MIMO Class:** MIMO Class component has multiple output ports. If MIMO Class component has only one output port, `xaf_disconnect` API behavior is same as Audio Codec Class. If MIMO Class component has multiple output ports, `xaf_disconnect` API call would flush the output port and free buffers and message pool between ports but it would not cancel any pending processing of the component. Further, it would pass the output port disconnect information to component plugin through `XA_MIMO_PROC_CONFIG_PARAM_PORT_DISCONNECT` configuration parameter. Component plugin implementation should manage processing or execution with disconnected output port as they see fit. **Renderer Class:** Renderer Class component also has one optional output port (used as feedback path for echo cancellation etc.). `xaf_disconnect` API behavior on its output port is same as Audio Codec Class.

Actual Parameters	`p_src`
	Pointer to the source audio component structure
	
	`src_out_port`
	Output port number of source component (to be disconnected)
	
	`p_dest`
	Pointer to the destination audio component structure
	
	`dest_in_port`
	Input port number of destination component (to be disconnected from output port of source component)

Restrictions	Should not be called before ports (to be disconnected) are connected using `xaf_connect` API Application must properly handle disconnected components and pipeline, otherwise the processing pipeline may get stalled.

Example

```
ret = xaf_disconnect (p_audioComp1,
                      SRC_OUT_PORT_NUM,
                      p_audioComp2,
                     DEST_INP_PORT_NUM);
```

Errors

- Common API Errors
- XAF_ROUTING_ERROR
  Indicates that the API failed to disconnect the two requested ports (due to invalid port numbers, invalid components, or uninitialized source component, etc.)

Table 3-10 `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 initialization, execution, and wrap-up based on the process `flag` provided to it. During pipeline execution, 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 and also for the components which are being probed. 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 queried by the `xaf_comp_get_status` API described in Table 3-11. **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>Use this flag to initialize processing, to be called only once for each component, during initialization. After this API call, initialization status must be queried using <code>xaf_comp_get_status</code> API.</td></tr>
<tr><td><code>XAF_EXEC_FLAG</code></td><td>Use this flag to start execution, to be called only once for each component to start processing.</td></tr>
<tr><td><code>XAF_INPUT_OVER_FLAG</code></td><td>Use this flag to indicate input is complete when <code>xaf_comp_get_status</code> API returns <code>XAF_NEED_INPUT</code>, and input stream is exhausted.</td></tr>
<tr><td><code>XAF_INPUT_READY_FLAG</code></td><td>Use this flag to indicate input buffer availability when <code>xaf_comp_get_status</code> API returns <code>XAF_NEED_INPUT</code>, and input data is available.</td></tr>
<tr><td><code>XAF_NEED_OUTPUT_FLAG</code></td><td>Use this flag to request for output when <code>xaf_comp_get_status</code> API returns <code>XAF_OUTPUT_READY</code>, and size returned in <code>p_info[1]</code> is non-zero (zero indicates execution completion for the component).</td></tr>
<tr><td><code>XAF_NEED_PROBE_FLAG</code></td><td>Use this flag to request for probe output when <code>xaf_comp_get_status</code> API returns <code>XAF_PROBE_READY</code>, and size returned in <code>p_info[1]</code> is non-zero (zero indicates execution completion for the component).</td></tr>
</table>
</td></tr>
</table>

Restrictions	Should not be called before `xaf_comp_create` API

Example

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

Errors

- Common API Errors

Table 3-11 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. p_adev and p_comp 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; that is, it may block for status from the DSP Side XAF for a previously issued process command.</td></tr>
<tr><td>Actual Parameters</td><td>p_adev
Pointer to the audio device structure

p_comp
Pointer to the audio component structure

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

p_info
Pointer to array of size two WORD32 data types (pointer, size) to get information from the audio component associated with its status. When the p_status returned is XAF_STARTING or XAF_INIT_DONE, this buffer is not updated.</td></tr>
<tr><td>Restrictions</td><td>Should not be called before xaf_comp_create API</td></tr>
</table>

Example

```
WORD32 Info[2];
ret = xaf_comp_get_status(p_adev,
                    p_audioComp,
                    &compStatus,
                    &Info[0]);
```

Errors

- Common API Errors

Table 3-12 `xaf_pause` API

API	`XAF_ERR_CODE xaf_pause(pVOID p_comp, WORD32 port)`
Description	This API pauses the processing of data on specified port `port` of audio component `p_comp` i.e. if input port is paused, input data consumption would be paused on that port and if output port is paused, output data production would be paused on that port. This API has Class specific implementation as described below. **Audio Codec Class:** Audio Codec Class component has one input port and one output port, so `xaf_pause` API call on any port would simply pause the processing or execution of the component. Note, this may in turn pause the preceding and / or following pipeline processing. **Mixer Class:** Mixer Class component has four input ports and one output port. `xaf_pause` API call on any input port would not pause the component processing if there is at least one active input port with data. `xaf_pause` API call on output port would pause the component processing, and this may in turn pause the preceding and / or following pipeline processing. **MIMO Class:** MIMO Class component has multiple input ports and multiple output ports. `xaf_pause` API call on any port would only pass paused port information to the component plugin using `XA_MIMO_PROC_CONFIG_PARAM_PORT_PAUSE` configuration parameter and component plugin implementation should manage processing or execution with paused port as it sees fit. Note, this may in turn pause the preceding and / or following pipeline processing. **Capturer Class:** **Renderer Class:** Being hardware specific, Capturer or Renderer Class do not support `xaf_pause` API. The pause feature can be implemented by component plugin through configuration parameter.

Actual Parameters	`p_comp`
	Pointer to the audio component structure
	
	`port`
	Port number of the input or output port to be paused

Restrictions	Should not be called before `xaf_comp_create` API

Example

```
ret = xaf_pause (p_audioComp, port_num);
```

Errors

- Common API Errors

Table 3-13 `xaf_resume` API

API	`XAF_ERR_CODE xaf_resume(pVOID p_comp, WORD32 port)`
Description	This API resumes processing of data on specified port `port` of audio component `p_comp` i.e. if input port is resumed, input data consumption would be resumed on that port and if output port is resumed, output data production would be resumed on that port. For MIMO Class components, `xaf_resume` API call passes the port resume information to component plugin through `XA_MIMO_PROC_CONFIG_PARAM_PORT_RESUME` configuration parameter.

Actual Parameters	`p_comp`
	Pointer to the audio component structure
	
	`port`
	Port number of the input or output port to be resumed

Restrictions	Should not be called before `xaf_comp_create` API

Example

```
ret = xaf_resume(p_audioComp, port_num);
```

Errors

- Common API Errors

Table 3-14 `xaf_probe_start` API

API	`XAF_ERR_CODE xaf_probe_start(pVOID p_comp)`
Description	This API starts probe operation on audio component `p_comp`. Probe operation enables exporting of processed data for specified ports to application on each process or execution call of the audio component. Ports to be probed for an audio component must be configured using the configuration parameter `XAF_COMP_CONFIG_PARAM_PROBE_ENABLE` during audio component initialization. Note, application may require creating a separate thread to query status and consume data exported through probe operation if it does not already have one for feeding input to and / or consuming output from the probed audio component.
Actual Parameters	`p_comp` Pointer to the audio component structure
Restrictions	Should not be called before `xaf_comp_create` API

Example

```
        param[0] = XAF_COMP_CONFIG_PARAM_PROBE_ENABLE ;
        param[1] = 0x3; // for probing both input and output port for
audio                           codec class
        xaf_comp_set_config(p_audioComp, 1, param);
        ret = xaf_probe_start (p_audioComp);
```

Errors

- Common API Errors

Table 3-15 `xaf_probe_stop` API

API	`XAF_ERR_CODE xaf_probe_stop(pVOID p_comp)`
Description	This API stops probe operation on audio component `p_comp`. Note, if application has created a separate thread to consume data exported through probe operation, it should be deleted by application after `xaf_probe_stop` API call.
Actual Parameters	`p_comp` Pointer to the audio component structure
Restrictions	Should not be called before `xaf_comp_create` API

Example

```
ret = xaf_probe_stop (p_audioComp);
```

Errors

- Common API Errors

Table 3-16 `xaf_adev_set_priorities` API

API	`XAF_ERR_CODE xaf_adev_set_priorities(pVOID p_adev, WORD32 n_rt_priorities, WORD32 rt_priority_base, WORD32 bg_priority)`
Description	This API enables preemptive scheduling of audio components on the DSP side XAF. By default, DSP Side XAF creates only one DSP worker thread for processing or execution of all audio components and preemption of one audio component processing by another is not supported. With `xaf_adev_set_priorities` API, preemptive scheduling is enabled, and a higher priority audio component processing request can preempt lower priority audio component processing. This is achieved using different priority RTOS threads for different priority audio components. These RTOS threads are created with `xaf_adev_set_priorities` API as described below. XAF priority for an audio component is set using `XAF_COMP_CONFIG_PARAM_PRIORITY` configuration parameter and it can be changed at runtime. `xaf_adev_set_priorities` API call sets up audio device `p_adev` for preemptive scheduling and creates (`n_rt_priorities + 1`) DSP worker threads. One DSP worker thread is dedicated to processing or execution of unprioritized audio components and it is assigned RTOS priority specified by `bg_priority`. Remaining `n_rt_priorities` threads are dedicated to processing or execution of audio components with XAF priorities from `0` to (`n_rt_priorities – 1`) and are assigned RTOS priorities from `rt_priority_base` to (`rt_priority_base + n_rt_priorities – 1`) respectively. Note, higher number indicates higher priority and vice versa.

Actual Parameters	`p_adev`
	pointer to the audio device structure
	
	`n_rt_priorities`
	number of real time priority levels
	
	`rt_priority_base`
	lowest real time priority level
	
	`bg_priority`
	back-ground priority level

Restrictions	Should not be called before `xaf_adev_open` API. Should be called only once after `xaf_adev_open` API.

Example

```
/* following call creates two DSP worker threads with priorities 3 and
 * 4 respectively for processing of prioritized components, and creates
 * one DSP worker thread with priority 1 for unprioritized components
 */
ret = xaf_adev_set_priorities(p_adev, 2, 3, 1);
```

Errors

- Common API errors

Table 3-17 `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 3-18 `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 Audio Components (`p_mem_stats[0]`),
	2. Shared Memory used by Audio Components and Framework (`p_mem_stats[1]`) and
	3. Local Memory used by Framework structures (`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_adev_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

3.4 XAF Configuration Parameters

This section describes configuration parameters that are supported by XAF. These parameters should be used with `xaf_comp_set_config` API described in Table 3-6.

Table 3-19 `XAF_COMP_CONFIG_PARAM_PROBE_ENABLE` Config Parameter

Configuration Parameter	`XAF_COMP_CONFIG_PARAM_PROBE_ENABLE`
Description	Probe operation enables exporting of processed data for specified ports to the application on each process or execution call of the audio component. This configuration parameter is used to specify ports for probe operation using a port mask value. Port mask is a 32-bit unsigned integer where bit 0 (LSB) corresponds to port number 0, bit 1 corresponds to port number 1 and so on. If a bit is set, the corresponding port is enabled for probe operation.
Values	Value Type: `UWORD32` Default Value: `0` (All ports disabled) Example value: `0x3` (port 0 and port 1 are enabled for probe operation)
Restrictions	This API is only supported during audio component initialization (as it results in one-time probe buffer allocation during initialization) i.e. probe specification cannot be changed at runtime.

Table 3-20 `XAF_COMP_CONFIG_PARAM_RELAX_SCHED` Config Parameter

<table>
<tr><td>Configuration Parameter</td><td><code>XAF_COMP_CONFIG_PARAM_RELAX_SCHED</code></td></tr>
<tr><td>Description</td><td>By default, each processing or execution call of MIMO Class component requires that all active ports are ready i.e. all active input ports have data and all active output ports have buffer available.
This configuration parameter is used to specify ports on which this readiness check should be relaxed using a port mask value. Port mask is a 32-bit unsigned integer where bit 0 (LSB) corresponds to port number 0, bit 1 corresponds to port number 1 and so on. If a bit is set, the corresponding port readiness check should be relaxed during MIMO Class component processing.
Note, if this configuration parameter is used, it is responsibility of respective component plugin implementation to manage execution without readiness of specified ports.</td></tr>
<tr><td>Values</td><td><table>
<tr><td>Value Type</td><td><code>UWORD32</code></td></tr>
<tr><td>Default Value</td><td><code>0</code> (All ports disabled)</td></tr>
<tr><td>Example value</td><td><code>0x3</code> (port 0 and port 1 readiness checks are relaxed)</td></tr>
</table></td></tr>
<tr><td>Restrictions</td><td>This API is only supported for MIMO Class components and it can be used at component initialization as well as at runtime.</td></tr>
</table>

Table 3-21 XAF_COMP_CONFIG_PARAM_PRIORITY Config Parameter

Configuration Parameter	XAF_COMP_CONFIG_PARAM_PRIORITY
Description	By default, DSP Side XAF creates only one DSP worker thread for processing or execution of all audio components and preemption of one audio component processing by another is not supported. With xaf_adev_set_priorities API, preemptive scheduling is enabled, and a higher priority audio component processing request can preempt lower priority audio component processing. This configuration parameter is used to specify priority of audio component. It accepts values from 0 to (max(UWORD32)-1). Note, higher number indicates higher priority and vice versa.
Values	Value Type: UWORD32 Default Value: 0 (runs with default lowest priority) Example value: 0x3 (audio component runs at priority 3)
Restrictions	This API is supported at component initialization as well as at runtime. For this configuration parameter to have effect, xaf_adev_set_priorities API must be used to create different priority RTOS threads during audio device creation, else this parameter would be ignored.

4. Xtensa Audio Framework Package

The XAF package is released in following two forms. The contents of XAF release package and steps to build and execute in both forms are described in the following sections.

1. .tgz package for linux / makefile based usage
2. .xws package for Xtensa Xplorer based usage

4.1 XAF Sample Applications

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

Note All 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.

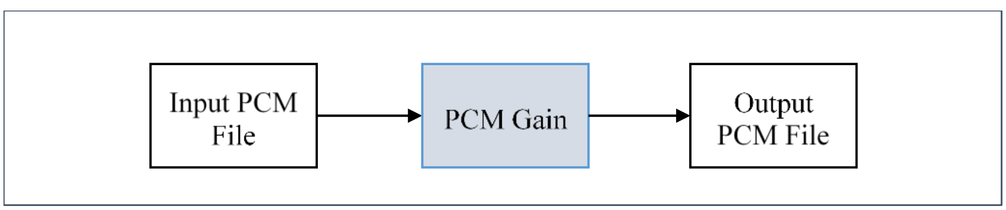


Figure 4-1 Testbench 1 (pcm-gain) Block Diagram

Testbench 2 (`xa_af_dec_test`) decodes MP3 streams.

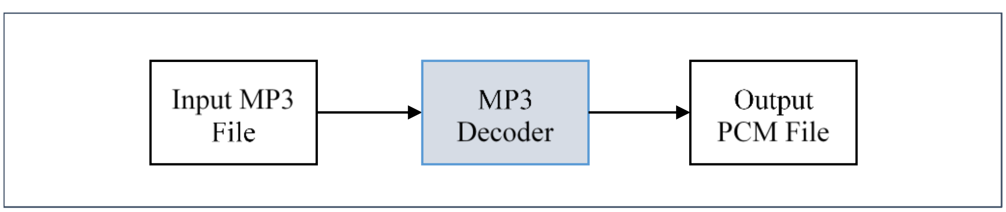


Figure 4-2 Testbench 2 (mp3-dec) Block Diagram

Testbench 3 (`xa_af_dec_mix_test`) decodes two MP3 streams and mixes the output.

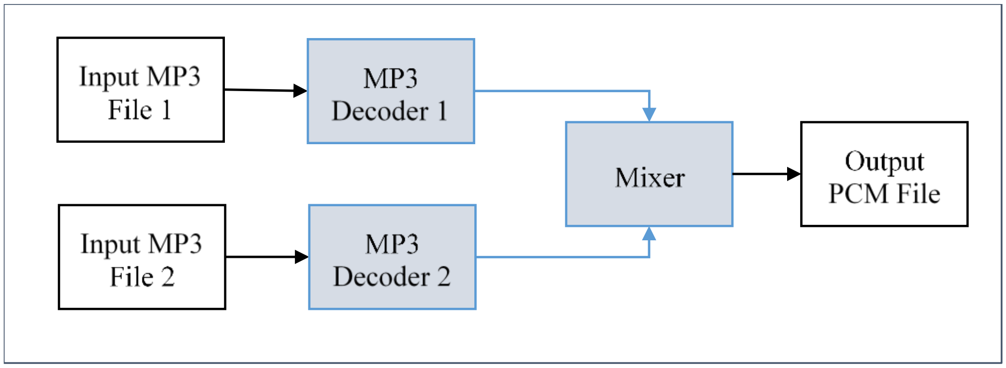


Figure 4-3 Testbench 3 (dec-mix) Block Diagram

Testbench 4 (`xa_af_full_duplex_test`) decodes an MP3 stream and simultaneously encodes an MP3 stream.

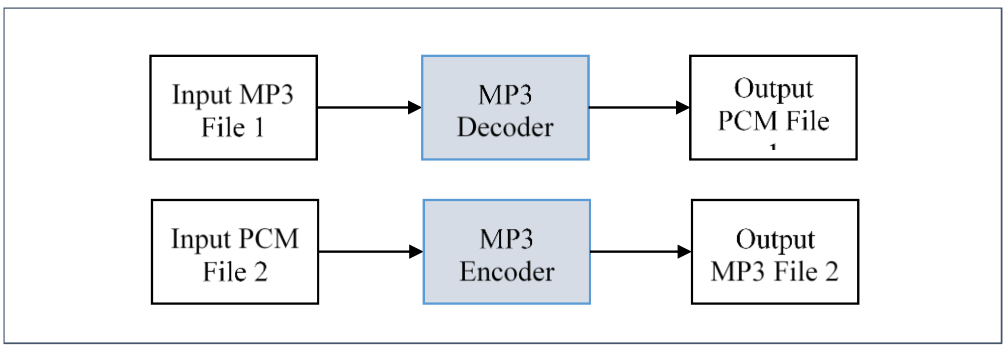


Figure 4-4 Testbench 4 (full-duplex) Block Diagram

Testbench 5 (`xa_af_amr_wb_dec_test`) decodes AMR-WB speech streams.

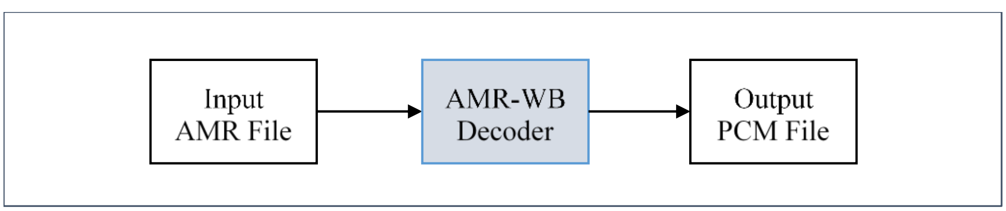


Figure 4-5 Testbench 5 (amr-wb-dec) Block Diagram

Testbench 6 (`xa_af_src_pp_test`) does a sample rate conversion.

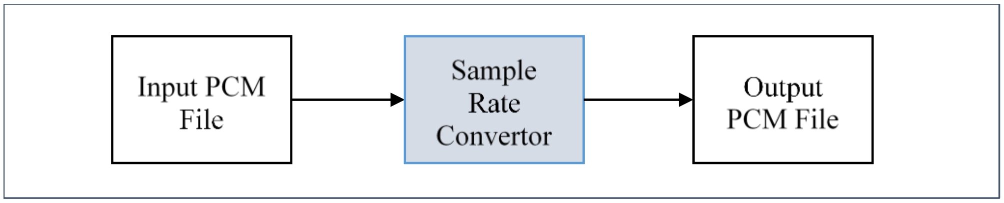


Figure 4-6 Testbench 6 (sample-rate-convert) Block Diagram

Testbench 7 (`xa_af_aac_dec_test`) decodes AAC streams.

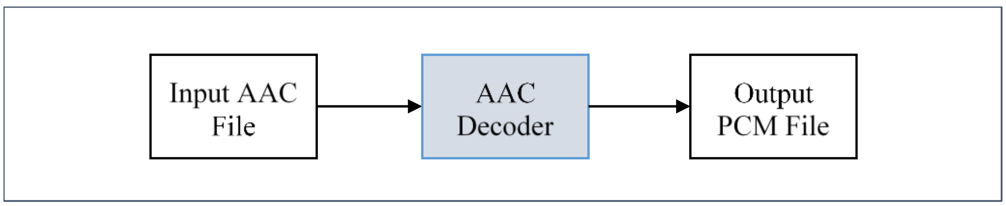


Figure 4-7 Testbench 7 (aac-dec) Block Diagram

Testbench 8 (`xa_af_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.

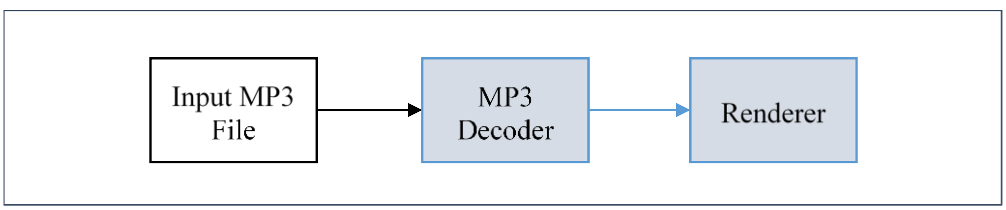


Figure 4-8 Testbench 8 (mp3-dec-renderer) Block Diagram

Testbench 9 (`xa_af_gain_rend_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.

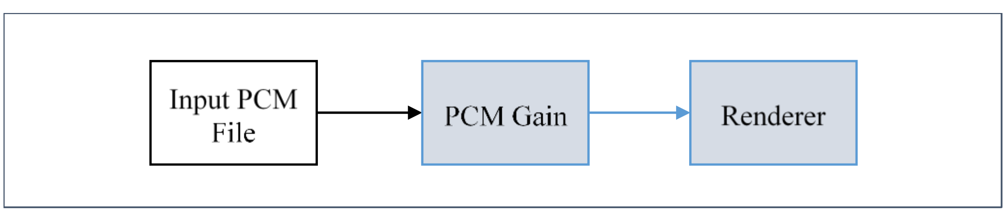


Figure 4-9 Testbench 9 (pcm-gain-renderer) Block Diagram

Testbench 10 (`xa_af_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.

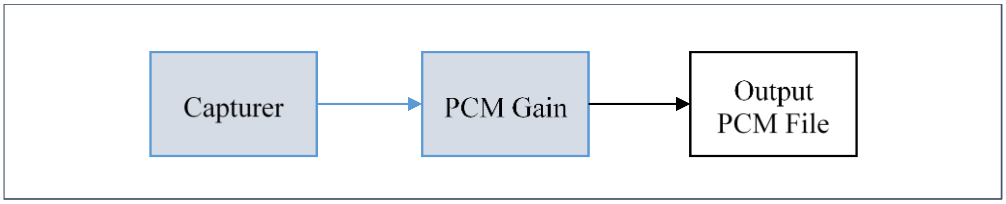


Figure 4-10 Testbench 10 (capturer-pcm-gain) Block Diagram

Testbench 11 (`xa_af_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.

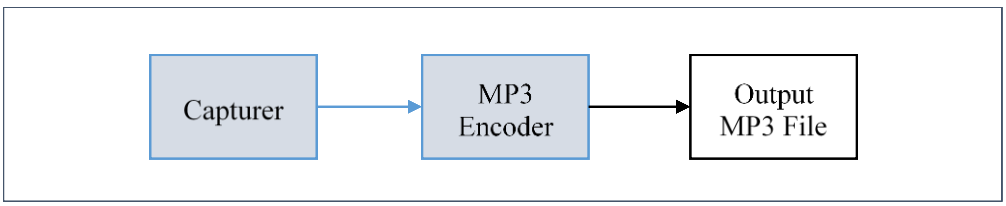


Figure 4-11 Testbench 11 (capturer-mp3-enc) Block Diagram

Testbench 12 (`xa_af_vorbis_dec_test`) decodes Ogg-Vorbis streams.

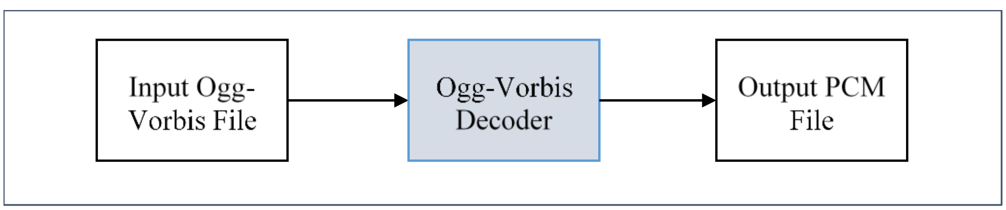


Figure 4-12 Testbench 12 (ogg-vorbis-dec) Block Diagram

Testbench 13 (`xa_af_pcm_mix_test`) applies gain to two PCM streams and mixes then to produce the output.

Note, this testbench demonstrates runtime pause, resume, probe start, and probe stop operations. Refer to testbench help for details on how to exercise these operations at runtime.

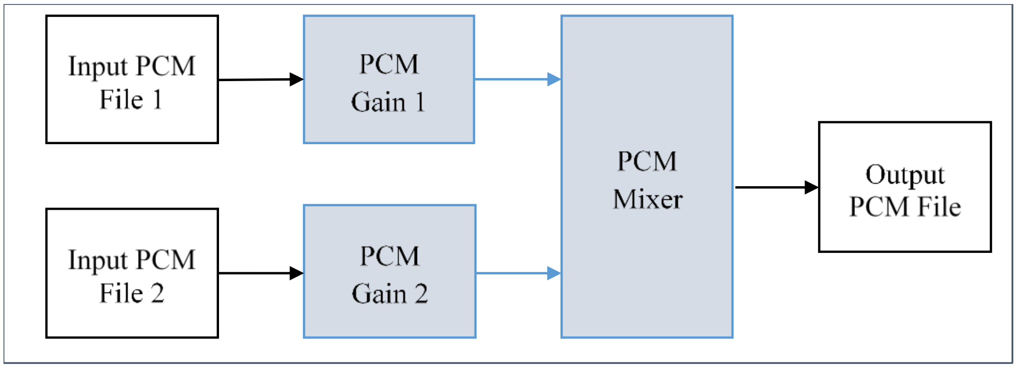


Figure 4-13 Testbench 13 (pcm-mix) Block Diagram

Testbench 14 (`xa_af_playback_usecase_test`) decodes two MP3 streams and one AAC stream and mixes the output. This mixer output is split into (copied to) two PCM streams, gain is applied on one stream and sample rate is converted on another stream. Second AAC decoder can be created and connected to mixer at runtime.

Note, this testbench demonstrates runtime pause, resume, disconnect, re-connect, probe start, and probe stop operations. Refer to testbench help for details on how to exercise these operations at runtime.

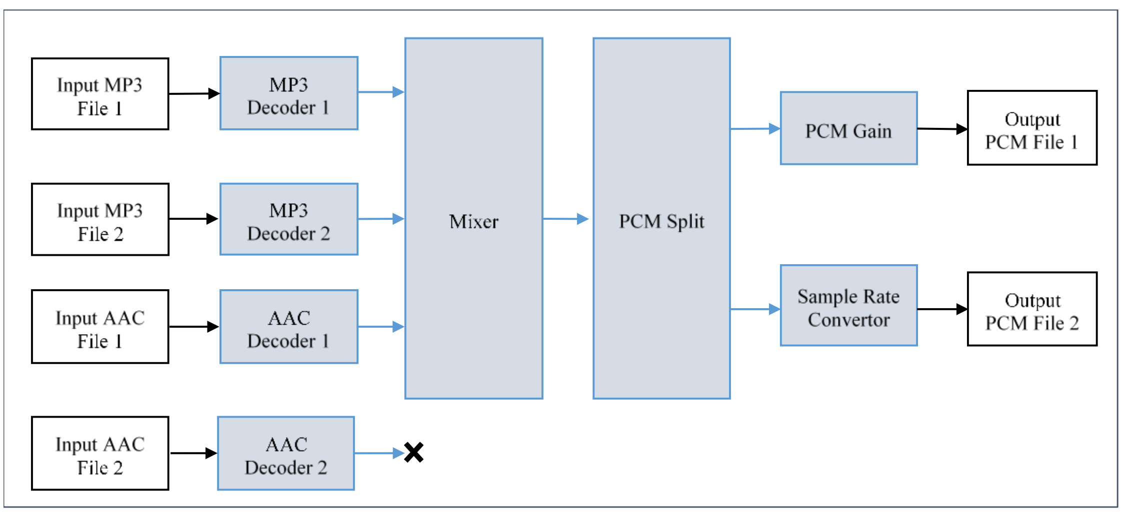


Figure 4-14 Testbench 14 (playback-usecase) Block Diagram

Testbench 15 (`xa_af_renderer_ref_port_test`) demonstrates use of renderer optional port as feedback or reference path for echo cancellation type of applications.

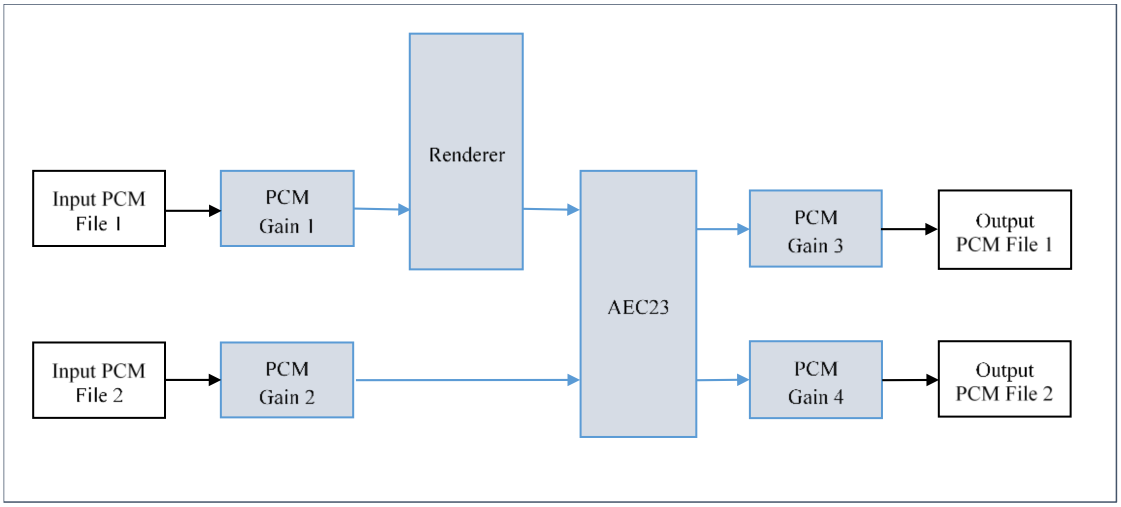


Figure 4-15 Testbench 15 (renderer-ref-port) Block Diagram

Table 4-1 summarizes component header file, component wrapper file and component library dependencies for each of fifteen testbenches included in XAF package.

Table 4-1 Component Dependencies for Testbenches

No	Testbench source file	Component wrapper files	Component header files	Component libraries
1	xaf-pcm-gain-test.c	xa-pcm-gain.c	xa-pcm-gain-api.h	-
2	xaf-dec-test.c	xa-mp3-decoder.c	xa_mp3_dec_api.h	xa_mp3_dec.a
3	xaf-dec-mix-test.c	xa-mp3-decoder.c xa-mixer.c	xa_mp3_dec_api.h xa-mixer-api.h	xa_mp3_dec.a
4	xaf-full-duplex-test.c	xa-mp3-decoder.c xa-mp3-encoder.c	xa_mp3_dec_api.h xa_mp3_enc_api.h	xa_mp3_dec.a xa_mp3_enc.a
5	xaf-amr-wb-dec-test.c	xa-amr-wb-decoder.c	xa_amr_wb_codec_api.h xa_amr_wb_dec_definitions.h	xa_amr_wb_codec.a
6	xaf-src-test.c	xa-src-pp.c	xa_src_pp_api.h	xa_src_pp.a
7	xaf-aac-dec-test.c	xa-aac-decoder.c	xa_aac_dec_api.h	xa_aac_dec.a
8	xaf-mp3-dec-rend-test.c	xa-mp3-decoder.c xa-renderer.c	xa_mp3_dec_api.h xa-renderer-api.h	xa_mp3_dec.a
9	xaf-gain-renderer-test.c	xa-pcm-gain.c xa-renderer.c	xa-pcm-gain-api.h xa-renderer-api.h	-

No	Testbench source file	Component wrapper files	Component header files	Component libraries
10	xaf-capturer-pcm-gain-test.c	xa-capturer.c xa-pcm-gain.c	xa-capturer-api.h xa-pcm-gain-api.h	-
11	xaf-capturer-mp3-enc-test.c	xa-capturer.c xa-mp3-encoder.c	xa-capturer-api.h xa_mp3_enc_api.h	xa_mp3_enc.a
12	xaf-vorbis-dec-test.c	xa-vorbis-decoder.c	xa_vorbis_dec_api.h	xa_vorbis_dec.a
13	xaf-pcm-mix-test.c	xa-pcm-gain.c xa-pcm-split.c	xa-pcm-gain-api.h xa-pcm-split-api.h	-
14	xaf-playback-usecase-test.c	xa-mp3-decoder.c xa-aac-decoder.c xa-mixer.c xa-pcm-split.c xa-pcm-gain.c xa-src-pp.c	xa_mp3_dec_api.h xa_aac_dec_api.h xa-mixer-api.h xa-pcm-split-api.h xa-pcm-gain-api.h xa_src_pp_api.h	xa_mp3_dec.a xa_aac_dec.a xa_src_pp.a
15	xaf-renderer-ref-port-test.c	xa-pcm-gain.c xa-renderer.c xa-aec23.c	xa-pcm-gain-api.h xa-renderer-api.h xa-aec23-api.h	-

4.2 XAF Package Directory Structure

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`
- `xaf-pcm-mix-test.c`
- `xaf-playback-usecase-test.c`
- `xaf-renderer-ref-port-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/`)

- `include/audio` – API header files for different audio components.
- `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.

XAF library directories (`/algo/`)

- `hifi-dpf/` – DSP Side XAF source and include files.
- `host-apf/` – App Side XAF source and include files. Includes XAF Developer APIs implementation.
- `xa_af_hostless/` – XAF common internal header files.

XAF include directories (`/include/`)

- `audio/` – XAF processing class specific header files. Also includes API, error, memory, type definition standard header files.
- `sysdeps/freertos` – FreeRTOS OSAL API definition header files.
- `sysdeps/xos` – XOS OSAL API definition header files.
- `xaf-api.h` – XAF Developer APIs header file.
- `xf-debug.h` – XAF debug trace support header file.

4.3 Build and Execute using tgz Package

4.3.1 Making the Executable

Before building the executable, ensure the environment variable `$XTENSA_CORE` is set correctly. The make commands mentioned below would build XAF Library and testbenches with XOS.

To build XAF Library and testbenches with FreeRTOS as RTOS:

1. Follow steps mentioned in Section 4.5 to build FreeRTOS library.
2. Use the make commands mentioned below with the options specified in square brackets **[]**. Note, `FREERTOS_BASE` directory should be `<BASE_DIR>/FreeRTOS` from step 1 above.

XAF Library:

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, enter

```
$ xt-make clean all install [XA_RTOS=freertos FREERTOS_BASE=<dir>]
```

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

Testbench 1 Only:

To build the pcm-gain testbench application (shown in Figure 4-1 above), follow these steps:

1. Go to /test/build.
2. At the prompt, enter:

```
$ xt-make -f makefile_testbench_sample clean all [XA_RTOS=freertos FREERTOS_BASE=<dir>]
```

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 and the respective API header files 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
```

Copy these API header files to the following directory.

```
/test/include/audio/xa_mp3_dec_api.h
/test/include/audio/xa_mp3_enc_api.h
/test/include/audio/xa_amr_wb_codec_api.h
/test/include/audio/xa_src_pp_api.h
/test/include/audio/xa_aac_dec_api.h
/test/include/audio/xa_vorbis_dec_api.h
```

1. Go to /test/build.
2. At the prompt, enter:

```
$ xt-make -f makefile_testbench_sample clean all-dec [XA_RTOS=freertos
FREERTOS_BASE=<rtos_dir>]
```

This will build all the testbench applications.

Special Build Settings

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

4.3.2 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. Refer to individual testbench help to get more details on command line options to run different test cases. Note, there is no difference in run commands for XAF with XOS or FreeRTOS.

Testbench 1 only:

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

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

All Testbenches:

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

```
$ 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 `renderer_out.pcm` in the execution directory. Similarly, the input for capturer testbench is read from the input file `capturer_in.pcm` and is expected to be present in the execution directory.

4.4 Build and Execute using xws Package

Note	The above testbenches require Xtensa Xplorer version 8.0.11 or newer.

4.4.1 Working with XAF xws Package

Following are the steps for importing to Xtensa Xplorer and building testbenches. By default, XAF Library and testbenches are built with XOS. To use FreeRTOS, refer to instructions in Section 4.4.2 below.

1. To import the HiFi Audio Framework Xtensa Workspace file (extension xws) into Xplorer, click **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 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 the **Run** button.
   ```
   -infile:<input PCM file> -outfile:<output PCM file>
   ```
5. To build and run other testbenches, follow these steps:
   a. Copy the library binary and API header file of the component (if required) to the location test/plugins/cadence/<component>/lib/ and test/include/audio respectively. Refer to Table 4-1 for component dependencies of various testbenches.

b. Ensure that following symbols are defined in “Build Properties” under “Symbols” tab. Note, you may choose to define only required symbols for the testbench from the list below.

```
-DXA_PCM_GAIN=1
-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 (e.g. ${*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 the one you want to run, by right-clicking those files and selecting “Build\Exclude”.

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

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

g. Note, if more than required components are enabled in `test/plugins/xa-factory.c` (for example, due to default enabled “Symbols” as mentioned in step b above) and respective component wrappers and libraries are not included in compilation, a dummy wrapper function can be defined in testbenches to avoid compilation errors. For example, a dummy wrapper function for MP3 Decoder can be defined as follows.

```
XA_ERRORCODE    xa_mp3_decoder(xa_codec_handle_t    var1,
WORD32 var2, WORD32 var3, pVOID var4) {return 0;}
```

6. To enable trace prints for analysis or debugging, add `XF_TRACE = 1` in the “Symbols” tab for both 'libxa_af_hostless' and 'testxa_af_hostless' projects.

4.4.2 Switching to FreeRTOS with XAF xws Package

Following are the steps to use FreeRTOS with XAF xws package.

1. Build FreeRTOS library using steps mentioned in Section 4.5. `<BASE_DIR/FreeRTOS>` path is defined as per this step.
2. For 'libxa_af_hostless' project, modify include paths for common target as below.

   (Go to T:Debug, select Modify, select Target as Common in the new window that appears, select 'Include Paths' tab)

Replace `'${workspace_loc}/libxa_af_hostless/build/../include/sysdeps/xos/include'` with `'${workspace_loc}/libxa_af_hostless/build/../include/sysdeps/freertos/include'`

3. For 'libxa_af_hostless' project, add following include paths for common target.

   ```
   <BASE_DIR>/FreeRTOS/lib/include
   <BASE_DIR>/FreeRTOS/lib/include/private
   <BASE_DIR>/FreeRTOS/lib/FreeRTOS/portable/XCC/Xtensa
   <BASE_DIR>/FreeRTOS/demos/cadence/sim/common/config_files
   ```

4. For 'libxa_af_hostless' project, update Symbols as below.

   (Go to T:Debug, select Modify, select Target as Common in the new window that appears, select 'Symbols' tab)

   Replace `'HAVE_XOS'` with `'HAVE_FREERTOS'` in Defined Symbols list.

5. For 'testxa_af_hostless' project, modify include path for common target as below.

   (Go to T:Debug, select Modify, select Target as Common in the new window that appears, select 'Include Paths' tab)

   Replace `'${workspace_loc}/libxa_af_hostless/include/sysdeps/xos/include'` with `'${workspace_loc}/libxa_af_hostless/include/sysdeps/freertos/include'`

6. For 'testxa_af_hostless' project, add following include path for common target.

   ```
   <BASE_DIR>/FreeRTOS/lib/include
   <BASE_DIR>/FreeRTOS/lib/include/private
   <BASE_DIR>/FreeRTOS/lib/FreeRTOS/portable/XCC/Xtensa
   <BASE_DIR>/FreeRTOS/demos/cadence/sim/common/config_files
   ```

7. For 'testxa_af_hostless' project, update Symbols as below.

   (Go to T:Debug, select Modify, select Target as Common in the new window that appears, select 'Symbols' tab)

   Replace `'HAVE_XOS'` with `'HAVE_FREERTOS'` in Defined Symbols list.

8. For 'testxa_af_hostless' project, update additional linker options as below.

   (Go to T:Debug, select Modify, select Target as Common in the new window that appears, select 'Addl linker' tab)

   Replace `'-lxos'` in Additional linker options with

   `'-L<BASE_DIR>/FreeRTOS/demos/cadence/sim/build/<your_hifi_core> -lfreertos'`

9. Clean and Build 'testxa_af_hostless' project, it should now run with FreeRTOS.

To switch back to XOS, revert steps 2 to 8 and Clean and Build 'testxa_af_hostless' project.

4.5 Building FreeRTOS for XAF

This section describes how to build the required version of FreeRTOS library to be used with XAF. Note, FreeRTOS compilation is only supported under Linux environment.

1. Copy /build/getFreeRTOS.sh from XAF Package to the directory of choice outside XAF Package under Linux environment. This directory is referred as <BASE_DIR> in following steps.
2. Set up environment variables to have Xtensa Tools in $PATH and $XTENSA_CORE defined to your HiFi core.
3. Execute getFreeRTOS.sh, this should download and build FreeRTOS library in <BASE_DIR/FreeRTOS>. The FreeRTOS library will be created in <BASE_DIR>/FreeRTOS/demos/cadence/sim/build/<your_hifi_core> directory.
4. $ ./getFreeRTOS.sh
5. You may copy <FreeRTOS> directory from Linux to Windows for building XAF Library and testbenches. In that case, the destination directory on Windows would be your new <BASE_DIR>.

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 Codec Application Programming Interface [2]. The interface (API) is a C-callable API that is exposed by all the HiFi based Audio 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 get_config for the following configuration parameters for the PCM data ports.

   `XA_CODEC_CONFIG_PARAM_CHANNELS`: Number of channels.

   `XA_CODEC_CONFIG_PARAM_SAMPLE_RATE`: Sampling rate.

   `XA_CODEC_CONFIG_PARAM_PCM_WIDTH`: PCM width.
3. XAF requires all MIMO class components to support set_config for the following configuration parameters to share port pause, resume, connect and disconnect information with component.

   `XA_MIMO_PROC_CONFIG_PARAM_PORT_PAUSE`: specified port is paused

   `XA_MIMO_PROC_CONFIG_PARAM_PORT_RESUME`: specified port is resumed

   `XA_MIMO_PROC_CONFIG_PARAM_PORT_CONNECT`: specified port is connected

   `XA_MIMO_PROC_CONFIG_PARAM_PORT_DISCONNECT`: specified port is disconnected
4. 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 must be followed to integrate the component library into XAF. 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. These steps are common for tgz and xws packages.

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. These steps are common for tgz and xws package.

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 port and a single output port and using audio codec as parent class), or
   - `xa_mixer_factory` (for components with multiple input ports and a single output port and using mixer as parent class),
   - `xa_renderer_factory` (for components with a single input port and zero or one optional output port and using renderer as parent class)
   - `xa_capturer_factory` (for components with zero input port and single output port and using capturer as parent class)
   - `xa_mimo_proc_factory` (for components with multiple input ports and multiple output ports and using mimo as parent class)

   MP3_DEC_EG: xa_audio_codec_factory

c. The function name for the audio component entry point, as defined in the component wrapper file created in Integration Step 1.

   MP3_DEC_EG: xa_mp3_decoder

7. In the constant definition of `xf_io_ports` (in `xa_factory.c`), add the port information based on `xaf_comp_type` for the new audio component. This step is not needed if `xaf_comp_type` for the new audio component already exists in the `xf_io_ports` definition.

   MP3_DEC_EG: The line below in `xa_factory.c`

```
{1, 1},      /* XAF_DECODER */
```

8. 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, refer to `test/src/ xaf-dec-mix-test.c` (MP3 decoder and mixer) and `xaf-mp3-dec-rend-test.c` (MP3 decoder and renderer).

Integration Step 3: Compile the application to use the component

The following steps are listed for tgz package (makefile based usage). For xws package, refer to Section 4.4.1 for further steps on how to include new application and component in xws project, and how to build and run it.

9. Update the `build/makefile_testbench` file appropriately to include component wrapper file and library into compilation.

MP3_DEC_EG:

```
XA_MP3_DECODER = 1
ifeq ($(XA_MP3_DECODER), 1)
PLUGINLIBS_MP3_DEC = $(ROOTDIR)/test/plugins/cadence/mp3_dec/lib/xa_mp3_dec.a
PLUGINOBJS_MP3_DEC += xa-mp3-decoder.o
INCLUDES += -I$(ROOTDIR)/test/plugins/cadence/mp3_dec
CFLAGS += -DXA_MP3_DECODER=1
vpath %.c $(ROOTDIR)/test/plugins/cadence/mp3_dec
endif
```

10. Update the `build/makefile_testbench` file appropriately to include the application source file into compilation and create executable binary.

    MP3_DEC_EG:

    ```
    APP2OBJS = xaf-dec-test.o.
    BIN2 = xa_af_dec_test
    ```

    Refer to `BIN2` compilation rules and dependencies in `build/makefile_testbench` file. Create similar rules and resolve the dependencies for new application.

11. Update the `build/makefile_testbench` file to add new application in the create (`all` or `all-dec`) and run (`run` or `run-dec`) targets

    MP3_DEC_EG:

    ```
    all: $(BIN2)
    run:
      $(RUN) ./$(BIN2) -infile:$(TEST_INP)/hihat.mp3 -outfile:$(TEST_OUT)/hihat_dec_out.pcm
    ```

12. Build and test the application. Refer to the procedure in Section 4.3.

13. Note, if more than required components are enabled in `test/plugins/xa-factory.c` (for e.g. due to default enabled switches in `build/makefile_testbench`) and respective component wrappers and libraries are not included in compilation, a dummy wrapper function can be defined in testbenches to avoid compilation errors.

    MP3_DEC_EG:

    ```
    /* Dummy unused functions */
    XA_ERRORCODE xa_mp3_decoder(xa_codec_handle_t var1, WORD32 var2,
    WORD32 var3, pVOID var4) {return 0;}
    ```

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 components as they are already part of XAF package. 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 [10]	Audio [2]	Decodes Ogg data	Folder: vorbis`_dec` Application: `xaf-vorbis-dec-test.c`
Cadence Opus encoder [11]	Speech [3]	Encodes Opus data	Folder: `opus`

6. Known Issues

The current version of XAF has only been tested with Version RI-2019.2 of the Xtensa tool chain. The Instruction Set Simulator (ISS) has been used in the cycle-accurate simulation mode. XAF does not support the fast functional “TurboXim” mode of Instruction Set Simulator (ISS).

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` API and depends on the two parameters `audio_comp_buf_size` and `audio_frmwk_buf_size` passed to this function.

1. `audio_comp_buf_size`: This is the memory allocated by XAF for usage by audio components. Local buffers required by audio components like connect buffers between components, persist buffers, scratch buffer will be allocated from this memory.

2. `audio_frmwk_buf_size`: This is the memory allocated by XAF for communication between application and audio components: Shared buffers required to transfer data and messages between application and audio components will be allocated from this memory.

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:

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

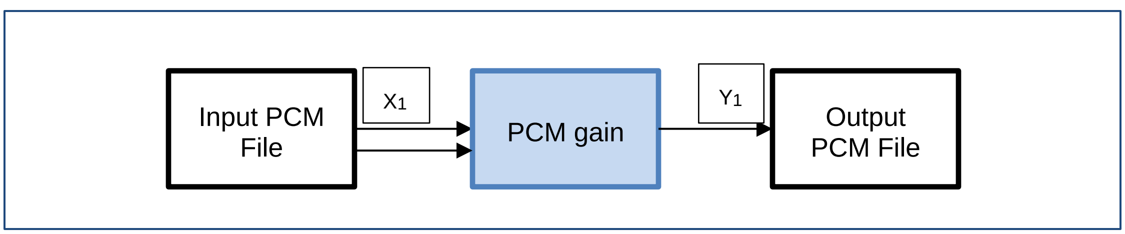
  In this version of XAF, the size of input buffer from application to the audio component is fixed at 4 KB, via the compile time constant `XAF_INBUF_SIZE`. Furthermore, some memory is also required by XAF itself. The size of the memory required by XAF is 16 KB, independent of the number of components.

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

The 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`.

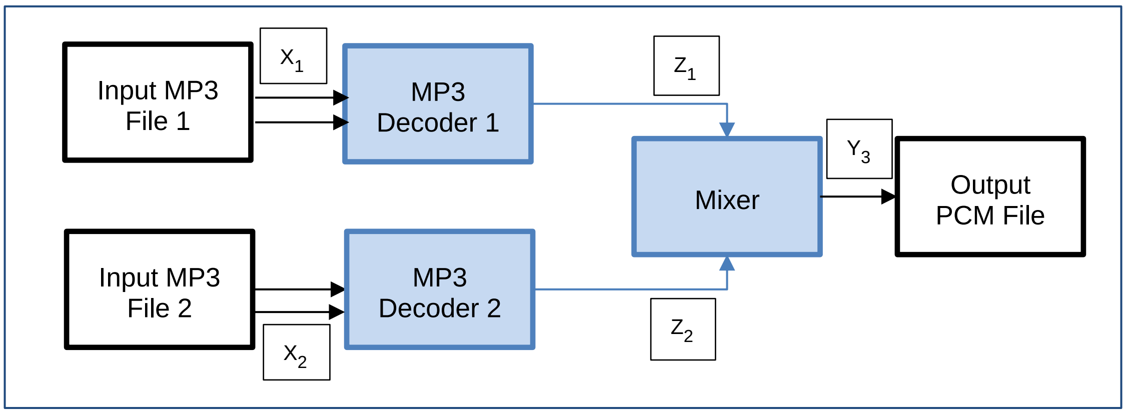
audio_frmwk_buf_size Computation:

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. Appendix: OSAL APIs

Operating System Abstraction Layer (OSAL) is defined for all RTOS functionality requirements in XAF. The list of all OSAL APIs defined and used in XAF is given below. Cadence XOS and FreeRTOS are supported with XAF. Porting XAF to a new RTOS would require implementation of these OSAL APIs with that new RTOS.

Note, Timer APIs mentioned below are only used by capturer and renderer components to mimic real time interrupts and by testbenches for MCPS measurement. The timer APIs are not required by XAF internal implementation.

API Class	OSAL API Defined in XAF
Message Queue API	xf_msgq_t **__xf_msgq_create** (size_t n_items, size_t item_size); void **__xf_msgq_destroy** (xf_msgq_t q); int **__xf_msgq_send** (xf_msgq_t q, const void *data, size_t sz); int **__xf_msgq_recv** (xf_msgq_t q, void *data, size_t sz); int **__xf_msgq_recv_blocking**(xf_msgq_t q, void *data, size_t sz); int **__xf_msgq_empty** (xf_msgq_t q); int **__xf_msgq_full** (xf_msgq_t q);
Thread API	int **__xf_thread_init** (xf_thread_t *thread); int **__xf_thread_create** (xf_thread_t *thread, xf_entry_t *f, void *arg, const char *name, void *stack, unsigned int stack_size, int priority); void __xf_thread_yield (void); int **__xf_thread_cancel** (xf_thread_t *thread); int **__xf_thread_join** (xf_thread_t *thread, int32_t * p_exitcode); int **__xf_thread_destroy** (xf_thread_t *thread); const char * **__xf_thread_name** (xf_thread_t *thread); int **__xf_thread_sleep_msec** (uint64_t msecs); int **__xf_thread_get_state** (xf_thread_t *thread);
Mutex API	void **__xf_lock_init** (xf_lock_t *lock); void **__xf_lock_destroy** (xf_lock_t *lock); void **__xf_lock** (xf_lock_t *lock); void **__xf_unlock** (xf_lock_t *lock);
Event API	void **__xf_event_init** (xf_event_t *event, uint32_t mask); void **__xf_event_destroy** (xf_event_t *event);

API Class	OSAL API Defined in XAF
	unsigned int **__xf_event_get** (xf_event_t *event); void **__xf_event_set** (xf_event_t *event, uint32_t mask); void **__xf_event_set_isr** (xf_event_t *event, uint32_t mask); void **__xf_event_clear** (xf_event_t *event, uint32_t mask); void **__xf_event_wait_any** (xf_event_t *event, uint32_t mask); void **__xf_event_wait_all** (xf_event_t *event, uint32_t mask);
Interrupt API	int **__xf_set_threaded_irq_handler** (int irq, xf_isr *irq_handler, xf_isr *threaded_handler, void *arg); int __xf_unset_threaded_irq_handler (int irq); unsigned long __xf_disable_interrupts (void); void **__xf_restore_interrupts** (unsigned long prev); void __xf_enable_interrupt (int irq); void __xf_disable_interrupt (int irq);
Timer API	int **__xf_timer_init** (xf_timer_t *timer, xf_timer_fn_t *fn, void *arg, int autoreload); unsigned long **__xf_timer_ratio_to_period** (unsigned long numerator, unsigned long denominator); int **__xf_timer_start** (xf_timer_t *timer, unsigned long period); int **__xf_timer_stop** (xf_timer_t *timer); int **__xf_timer_destroy** (xf_timer_t *timer);

OSAL APIs are declared in following header files for XOS:

```
/include/sysdeps/xos/include/osal-msgq.h
/include/sysdeps/xos/include/osal-thread.h
/include/sysdeps/xos/include/osal-timer.h
/include/sysdeps/xos/include/osal-isr.h
```

OSAL APIs are declared in following header files for FreeRTOS:

```
/include/sysdeps/freertos/include/osal-msgq.h
/include/sysdeps/freertos/include/osal-thread.h
/include/sysdeps/freertos/include/osal-timer.h
/include/sysdeps/freertos/include/osal-isr.h
```

9. References

[1] *Xtensa XOS Reference Manual* – For Version RI-2019.2 of the Xtensa tool chain, this is provided as part of the Xtensa tool chain, <TOOLS_INSTALL_PATH>/XtDevTools/downloads/RI-2019.2/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 version 2.7, 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)

[11] *Cadence Opus Codec* – Library version 1.8, API version 1.0 (HiFi Mini, HiFi 2, HiFi 3 and HiFi 4)

[12] *Xtensa port of FreeRTOS* - https://github.com/foss-xtensa/amazon-freertos/tree/xtensa-v2.0-xaf