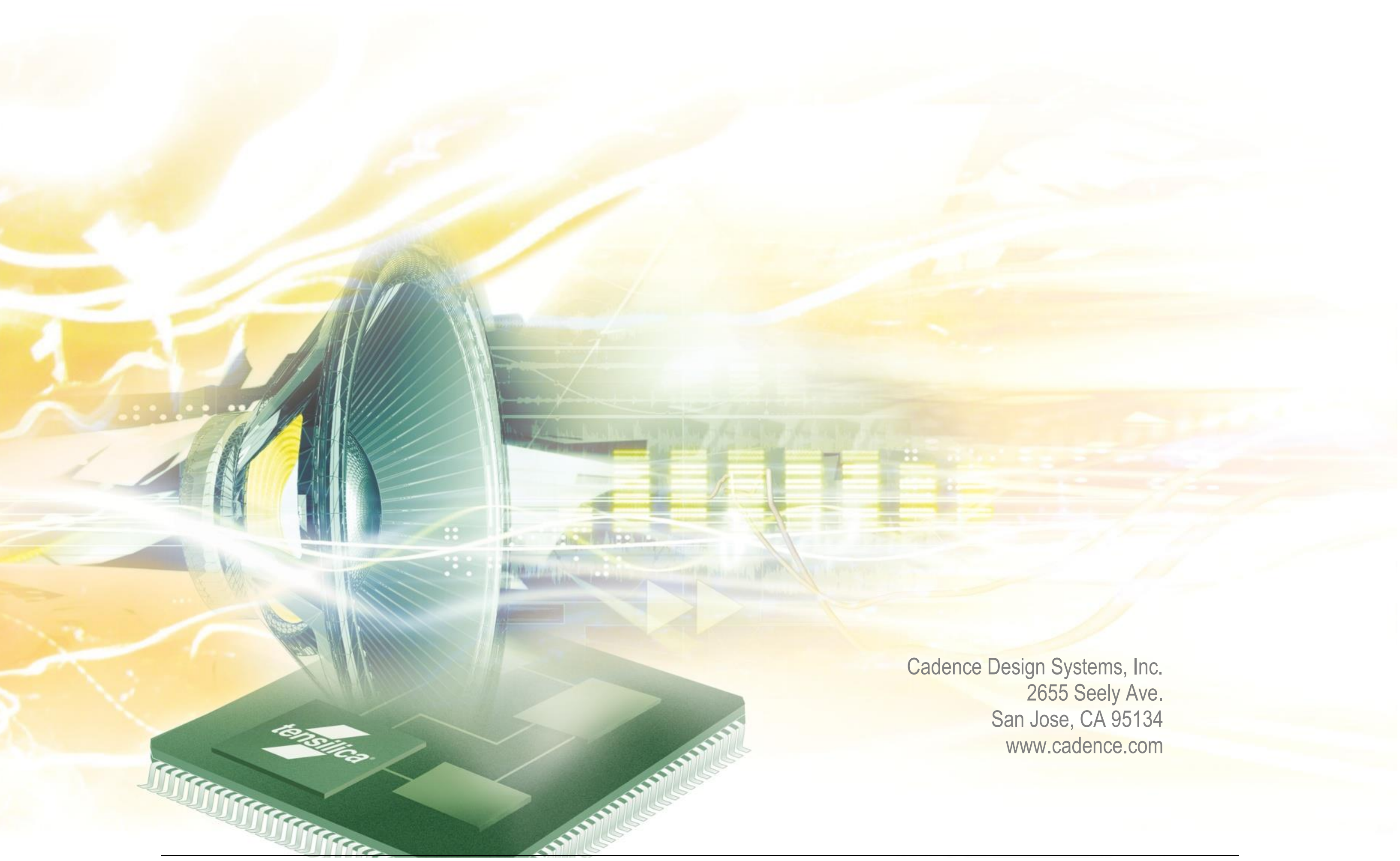
cadence®

Xtensa Audio Framework (Hostless)

Programmer's Guide

For HiFi DSPs and Fusion F1 DSP



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

Version: 3.1
Last Updated: May 2022

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

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 “TurboXim” ISS mode restriction with XAF. Sample Rate Converter component wrapper is updated to work with Sample Rate Converter 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 1.4.3. Updated Memory and Timings tables on Xtensa tools chain version RI-2019.2. Added support for Opus encoder plugin component.
2.3	Maintenance release. Added support for Fusion F1 DSP. Renamed App Side XAF to App Interface Layer and DSP Side XAF to DSP Interface Layer. Updated XAF error codes. Updated parameters range for xaf_comp_set_config, xaf_comp_get_config, and xaf_connect APIs. Renamed PCM Mixer component plugin to MIMO Mixer.
2.6	General Availability release. Added asynchronous event communication support between two components, between a component and application and between framework and application. Added support for components to request self-scheduling. Added new XAF Developer APIs for event communication: xaf_create_event_channel, xaf_delete_event_channel.

	Added new XAF Developer APIs to initialize default configuration parameters xaf_adev_config_default_init, xaf_comp_config_default_init. Updated prototype for XAF Developer API: xaf_adev_open, xaf_comp_create. Renamed XAF Developer APIs for backward compatibility: xaf_adev_open_deprecated, xaf_comp_create_deprecated. Updated Memory and Timings tables on Xtensa tools chain version RI-2019.2.
2.10	Added new XAF Developer APIs xaf_comp_set_config_ext, xaf_comp_get_config_ext Added support for input-port bypass Added support for decoder initialization without input Added Component Processing Errors section Updated XAF software stack diagram Updated Memory and Timings tables on Xtensa tools chain version RI-2021.6 with XT-CLANG compiler Added support for Opus decoder Added full-duplex Opus test example Removed standalone testbenches of AAC decoder, SRC post-proc, Ogg-Vorbis decoder Added TFLM support, library build steps, test examples and reference Updated steps for ‘Working with XAF xws Package’ (section 4.4.1) Updated XAF Sample Applications (section 4.1), Component Dependencies for Testbenches (Table 4-1), Example Components (Table 5-1)
3.1	Changes to have a common code base for both XAF-hostless and Multicore XAF solutions. Added software-stack diagrams for Multicore Extended multicore support up to 256 cores (8 bits of message-ID) Updated `xaf_adev_config_t` added `cb_compute_cycles` call-back function to collect execution cycles of worker DSPs and added `cb_stats` shared object pointer passed to the call-back function above. Added shared structure `xaf_perf_stats_s` consists of shared memory and execution cycle variables for each worker DSP. Added support for level triggered interrupt. Added support for global mutex locks using L32EX/S32EX instructions when XCHAL_HAVE_EXCLUSIVE is set which is mutually exclusive to the other type of instruction S32C1I. Added changes in `xaf_adev_config` structure for passing separate shared memory pointers for framework buffers and dsp shared buffers. Updated the terminology section 1.2.1 with Multicore specific terms. Updated Flowgraph Sequence for API Calls (Figure 3-2) Added the diagrams for Multicore-XAF: Application software stack diagram (Figure 2-2), Software architecture diagram (Figure 2-4), Memory architecture diagram (Figure 2-5). Added the APIs: `xaf_dsp_open`, `xaf_dsp_close.` Updated the APIs `xaf_adev_open, xaf_adev_close,xaf_get_mem_stats` for multicore changes.

	Updated the examples for `xaf_comp_get_config_ext/xaf_comp_set_config_ext`. Updated Xtensa Audio Framework Package, Build and Execute using XWS and TGZ Package sections, Memory guidelines for multicore-XAF. Added section 4.7 Building Multicore Subsystem. Added Multicore IPC abstraction API List to Appendix: OSAL APIs.

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 can 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 multicore version of XAF described in this guide is designed to work with a subsystem having single or multiple DSPs. Note that it does not assume any Host or controller core and it is a Hostless multicore solution in that sense.

For this document, HiFi DSPs include Fusion F1 DSP.

1.1 Document Overview

This guide covers all the information required to create, configure, and run audio processing chains using XAF Developer APIs. Section 1.4.3 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. Section 6 lists known issues. Section 7 provides memory allocation guidelines. Section 8 lists Operating System Abstraction Layer APIs. Section 9 provides references.

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 which can be either local-memory or global shared-memory.

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.

Event: An asynchronous message raised by a component to another component, or to application or to the framework.

Worker thread: OS threads running on the DSP Interface layer at various priority levels supported by XAF.

NCORES: Number of cores in the subsystem.

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

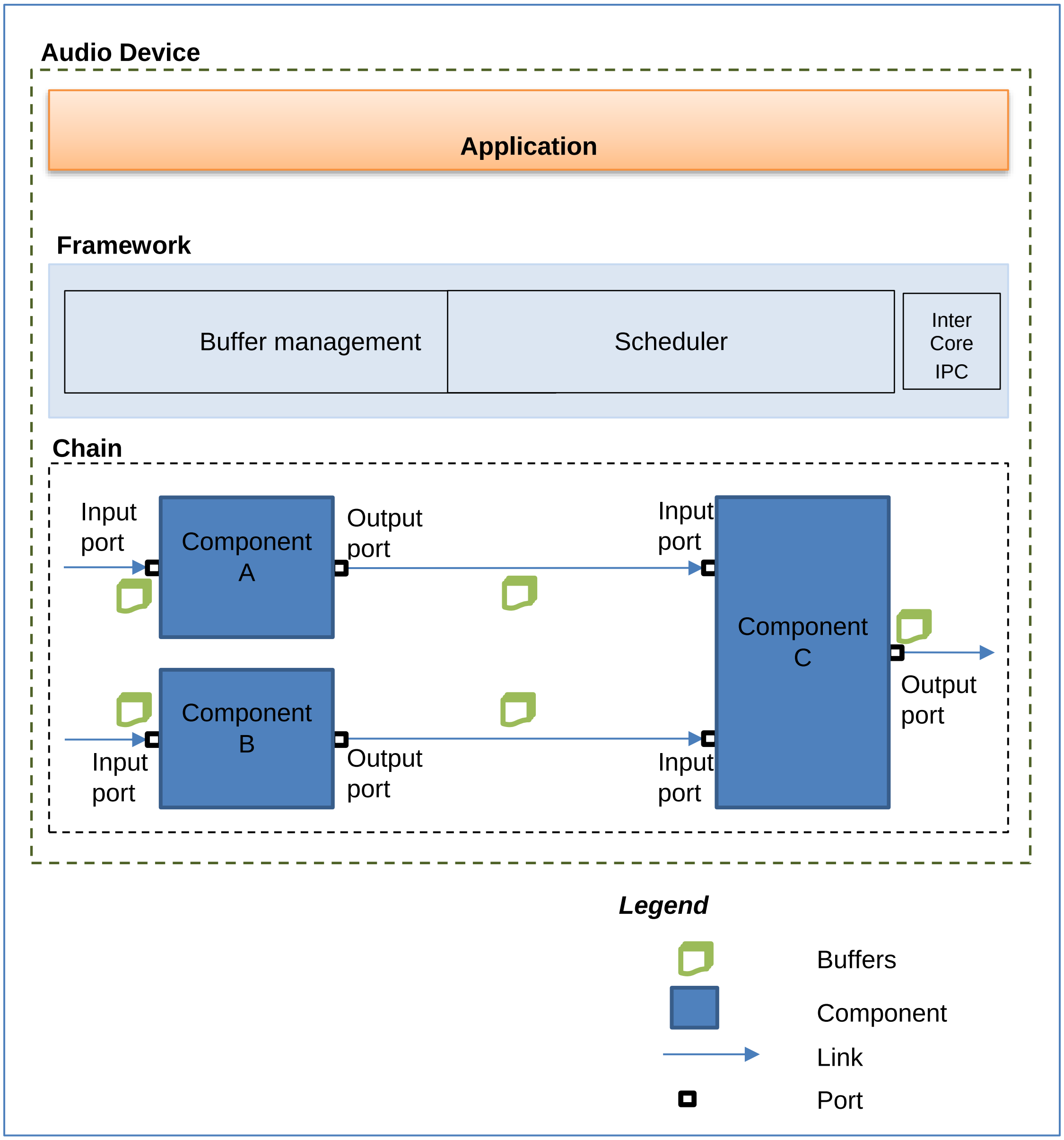


Figure 1-1 XAF Terminology

Multicore XAF: A subsystem with more than one core with a core designated as Master-DSP and remaining cores as Worker-DSPs. Each worker-DSP communicates with the application via Master-DSP.

Inter Core IPC: Inter-Core Communication is an abstraction layer which facilitates communication between any two DSPs in the subsystem using interrupts, global shared memory and global locks provided by the subsystem.

Master DSP: The DSP or core that has the Application-Interface-Layer. The test application and Worker-DSPs communicate through this Master-DSP.

Worker DSP: The DSP or core in the subsystem which is other than the Master-DSP. The Worker-DSPs communicate with the test application with the help of Inter-core IPC-layer and Master-DSP.

Cache Management: The Inter-core IPC layer carries out the necessary invalidation, writeback for memory synchronization when the cache is enabled on a DSP.

1.2.2 Port Numbering of Components in XAF

In XAF, port numbering of an audio component starts with 0 for the 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.

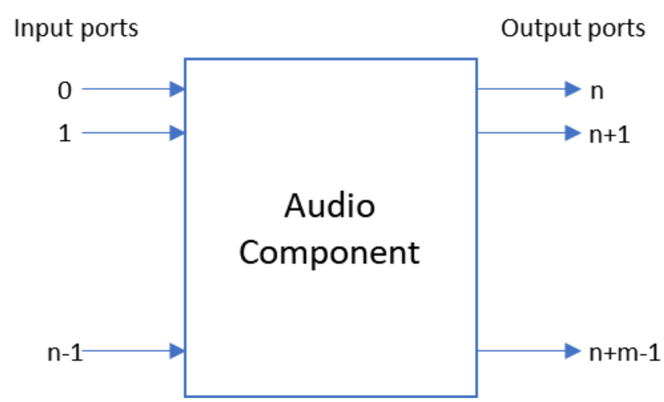


Figure 1-2 Port Numbered Audio Component

1.3 Xtensa Audio Framework Specifications

This section provides XAF specifications.

1.3.1 Feature Set

API Features:

- Ability to create components on a given DSP in the subsystem 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.
- Ability to raise and communicate events between components, or from a component to the application or from a component to the framework.

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. Because 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 programmers 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 example, microphone capture or speaker playback).

- Allows the creation and deletion of event communication channels between two components and between a component and application. Components can send asynchronous messages to application or to another component. Also, component execution errors can be communicated to the application using event channels.
- Allows component to request scheduling for itself. Note if component is already scheduled this request is ignored.
- 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).
- Supports multicore DSP subsystem of a combination of DSPs. The number of cores in the subsystem is configurable between 1 and 16, which is the maximum.

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 Converter, AAC decoder, Opus decoder and 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.
- TFLM inference support: With XAF, one can construct and execute tflm (TensorFlow[13] Lite Micro) inference models with different types inference engines for audio and image input data. The inference model libraries which are downloaded and built using XTENSA tools, are made readily usable by writing appropriate wrapper code (component plugins) which then interacts with the example application through XAF. The test application can provide the input data to the plugins through XAF and retrieve the outputs or inference results seamlessly. This is demonstrated by working examples like micro-speech inference and person-detect inference.

Supported Configurations:

- For NCORES=1, HiFi cores: HiFi 3, HiFi 4, HiFi 5, Fusion F1
- For NCORES>1, HiFi cores: HiFi 3, HiFi 4, HiFi 5.
- Xtensa Tools Chain: Version RI-2021.6
- Compiler: XT-CLANG
- RTOS: Cadence XOS [1] or FreeRTOS (Version 10.2.1) [12] (see details in Section 2.1.3)

Note	XAF is only tested with supported configurations mentioned above with up to NCORES=8, and it must be used with one of the supported configuration combinations.

Limitations:

- Only one instance of XAF can run at a time.

- 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 components cannot have multiple input ports or output ports connected to application.
- XTSC supports up to a maximum number of 16 cores (NCORES>1)

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 (NCORES=1)

Table 1-1 Library Memory

Text (Kbytes)				Data (Kbytes)
Fusion F1	HiFi 3	HiFi 4	HiFi 5	
54.3	50.6	57	66.4	0.8

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-2021.6 of the Xtensa tool chain with XOS and compiled with XT-CLANG.

The size of the total runtime memory allocated by XAF depends mainly on the two parameters `audio_framework_buffer_size` and `audio_component_buffer_size` of the `xaf_adev_config_t` structure which is 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_component_buffer_size` parameter of the `xaf_adev_config_t` structure 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_framework_buffer_size` parameter of the `xaf_adev_config_t` structure 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)			
		Fusion F1	HiFi 3	HiFi 4	HiFi 5
1	Local memory allocated by XAF for use by audio components	81	81	81	81
2	Shared memory allocated by XAF for communication between application and audio components	36.0	36.0	36.0	36.0
3	Memory used by XAF structures	44.8	44.8	44.8	44.8
	Total	161.8	161.8	161.8	161.8

Note The measurements are done with Version RI-2021.6 of the Xtensa tool chain.

Note For Testbench 1, `audio_framework_buffer_size` = 128 KB and `audio_component_buffer_size` = 256 KB are passed during `xaf_adev_open()` call. The actual memory used by XAF for Testbench 1 processing chain is shown in Table 1-2.

1.4.2 Timings (NCORES=1)

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 depends 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)			
		Fusion F1	HiFi 3	HiFi 4	HiFi 5
Testbench 1 – PCM Gain (Mono, 44.1KHz, Buffer size = 4096 samples)	XAF	0.5	0.6	0.6	0.5
	Total	0.7	3.7	2.7	2.6

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/Fusion F1 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-2021.6 of the Xtensa tool chain with XOS and compiled with XT-CLANG.

1.4.3 Memory (NCORES=2)

Table 1-4 Library Memory

Text (Kbytes)			Data (Kbytes)
HiFi 3	HiFi 4	HiFi 5	
55.6	62.7	73.0	0.8

Note	Other than for Text and Data, XAF uses ~2.3 Kbytes for bss. The measurements exclude the memory required by RTOS and the standard C library. The measurements are done with Version RI-2021.6 of the Xtensa tool chain with XOS and compiled with XT-CLANG.

The size of the total runtime memory allocated by XAF depends mainly on the three parameters `audio_framework_buffer_size`, `audio_shmem_buffer_size`, and `audio_component_buffer_size` parameters of the `xaf_adev_config_t` structure which is 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 four 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_component_buffer_size` parameter of the `xaf_adev_config_t` structure 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_framework_buffer_size` parameter of the `xaf_adev_config_t` structure passed to the `xaf_adev_open()` function.
3. Shared memory allocated by XAF for communication between audio components on different DSPs: This is the memory allocated by XAF to transfer data and messages between audio components created on different DSPs and it is controlled by `audio_shmem_buffer_size` parameter of the `xaf_adev_config_t` structure passed to the `xaf_adev_open()` function.
4. Memory used by XAF structures: This memory is allocated by XAF for its internal data structures on each DSP in the subsystem.

Table 1-5 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 created on worker-DSP in a 2-core subsystem with XCHAL_HAVE_EXCLUSIVE configuration option enabled).

Table 1-5 Runtime Memory

No	Memory breakup	RAM (Kbytes)		
		HiFi 3	HiFi 4	HiFi 5
1	Local memory allocated by XAF for use by audio components	64.9	64.9	64.9
2	Shared memory allocated by XAF for communication between application and audio components	36.0	36.0	36.0
3	Shared memory allocated by XAF for communication between audio components on different DSPs	33.8	33.8	33.8
4	Memory used by XAF structures	59.0	59.0	59.0
	Total	193.7	193.7	193.7

Note The measurements are done with Version RI-2021.6 of the Xtensa tool chain.

Note For Testbench 1, `audio_framework_buffer_size` = 128 KB, `audio_shmem_buffer_size` = 2432 KB and `audio_component_buffer_size` = 256 KB are passed as parameters of the `xaf_adev_config_t` structure during `xaf_adev_open()` call. The actual memory used by XAF for Testbench 1 processing chain is shown in Table 1-5.

1.4.4 Timings (NCORES=2)

Table 1-6 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 depends 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 created on worker-DSP in a 2-core subsystem with XCHAL_HAVE_EXCLUSIVE configuration option enabled) with XOS.

Table 1-6 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	1.3	1.2	1.2
	Total	4.4	3.3	3.3

Note	Performance specification measurements are carried out on a cycle-accurate Xtensa System-C (XTSC) execution environment with 1 cycle-delay/wait-state for memory access for HiFi 3/HiFi 4/HiFi 5 cores. This is nearly 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-2021.6 of the Xtensa tool chain with XOS and compiled with XT-CLANG.

2. Xtensa Audio Framework Architecture Overview

2.1 Application Software Architecture with Xtensa Audio Framework

Figure 2-1 and Figure 2-2 show various building blocks of application software based on XAF in single core and multicore subsystems respectively. Note that in these figures the application, RTOS, and audio components are not part of XAF. These building blocks are briefly described in the following sections.

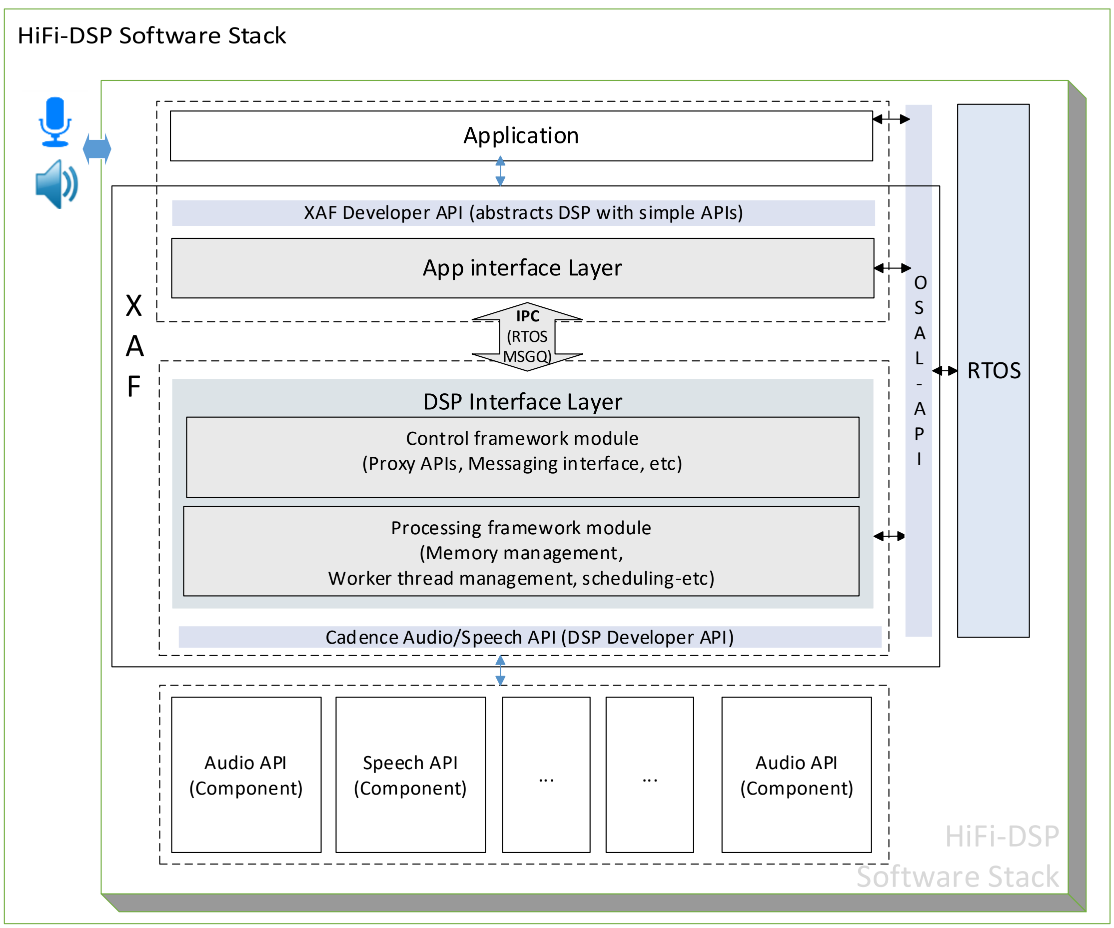


Figure 2-1 Application Software Stack Diagram (single core)

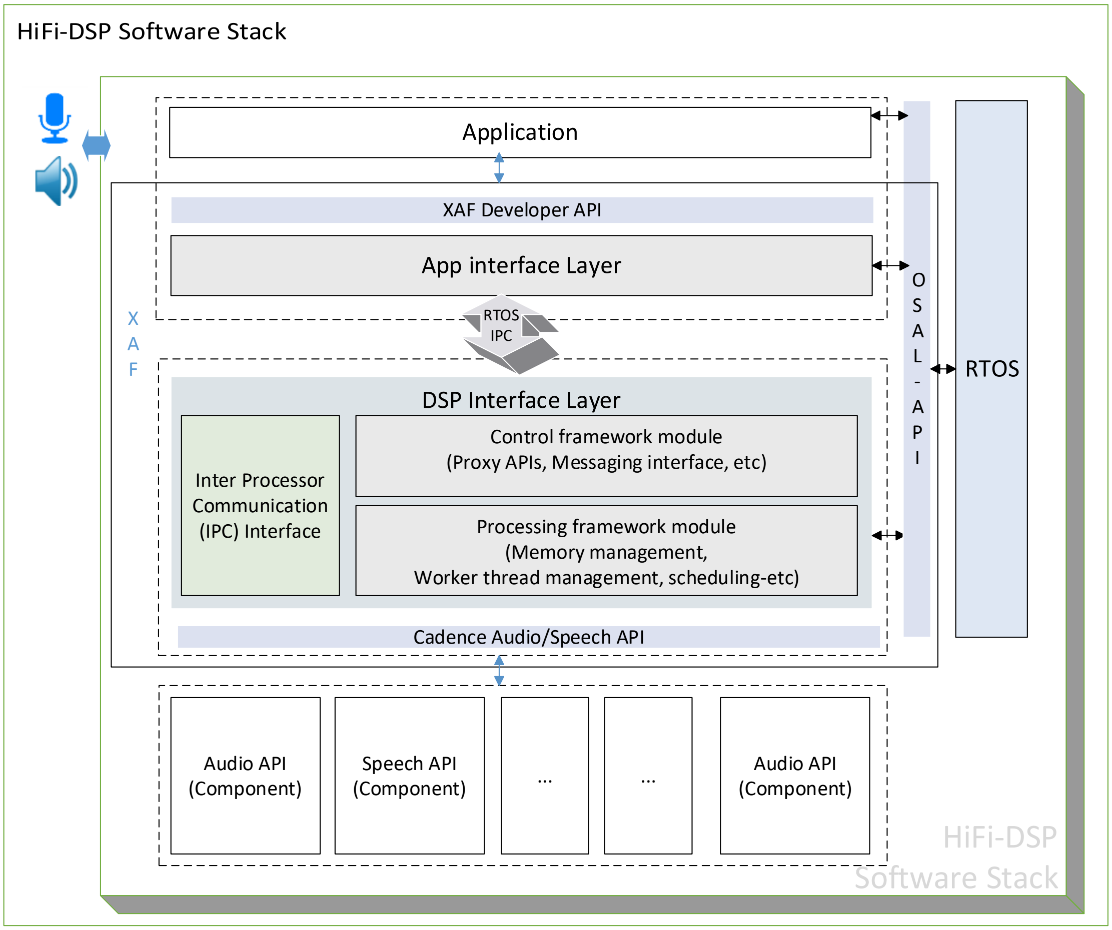


Figure 2-2 Application Software Stack Diagram (multi core)

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. In a multicore subsystem, the processing chains can be partitioned between multiple DSPs.

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 2-3 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.

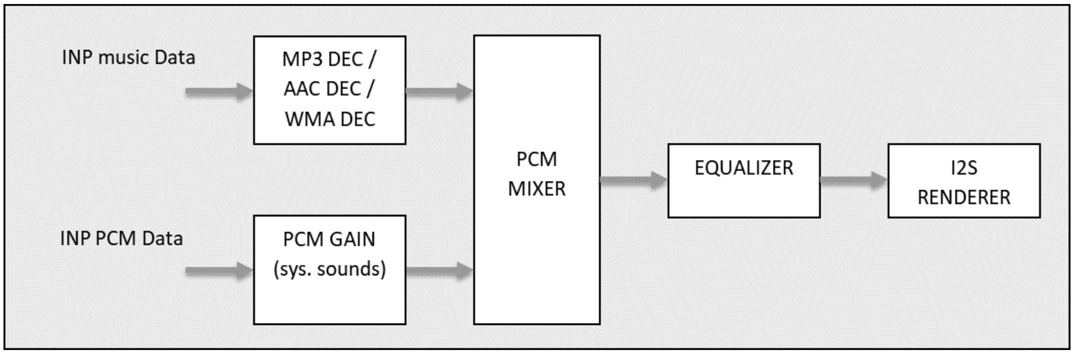


Figure 2-3 Example Music Playback Processing Chain

2.1.2 Xtensa Audio Framework Building Blocks

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 Interface Layer
- Inter-Process Connect (IPC)
- DSP Interface Layer

App Interface Layer

App (Application) Interface Layer is responsible for building and maintaining audio processing chains as per application's need. There is no actual audio processing done at this layer. Instead, it is a control code that runs in application thread context at highest priority with respect to the other two building blocks. App Interface Layer manages the operation of underlying DSP Interface Layer by sending commands and receiving responses from it. App Interface Layer also creates an IPC thread that receives responses from the DSP Interface Layer for the commands sent from the application. This thread runs at higher priority than the DSP Interface Layer thread.

IPC

Inter-Process Connect (IPC) is the communication link between App Interface Layer and DSP Interface Layer. It passes commands and responses between two layers and it has no knowledge about information being passed.

DSP Interface Layer

DSP Interface Layer does the actual audio processing based on commands received from App Interface Layer and sends responses back to App Interface Layer after command completion. Based on commands received from App Interface Layer, it creates, configures, and connects components to create processing chain and executes the components to perform audio processing. DSP Interface Layer runs in a separate thread context at lowest priority with regard to the other two building blocks. In DSP Interface Layer, by default 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 3-22 for details). Note, in this case multiple DSP worker threads will be created based on the number of different priority components. An example application for pre-emption could be where capturer and renderer components are configured with higher priority with respect to other data processing components so that processing of captured microphone data or playback of output PCM data is done in timely fashion without any gaps.

2.1.3 Multicore XAF

Multicore XAF Requirements

- Inter-core communication takes place through the shared memory which should be accessible by all the cores.
- Local memories of a core must be accessible from other cores through the in-bound PIF.
- Interrupts should be supported by all DSPs for inter-core communication. Each core needs at least one edge-triggered or level-triggered interrupt (<= EXCM Level) dedicated for inter-processor notification.
- The cores must be configured with Processor ID option.
- All cores must have same endianness.
- For inter-core synchronization, conditional stores (S32C1I) or have XCHAL_HAVE_EXCLUSIVE enabled with exclusive load/stores (L32EX/S32EX) should be supported on all cores.
- For cores with XCHAL_HAVE_EXCLUSIVE, global lock object should be located in a memory region with attributes: non-cacheable, shareable, bufferable.
- Cache-line size must be identical for all the cores.

In addition to the building blocks described in section 2.1.2, multicore XAF subsystem includes the following changes as shown in Figure 2-2.

Inter Core Communication Interface

Inter Core communication (Inter Core IPC) interface is the communication link between available DSP cores. This is a thin IPC layer implemented using software linked list. The message and payload buffers are allocated on the DSP shared memory pool. Interrupt notification mechanism to other cores is done using edge-triggered or level-triggered interrupt. Interrupt numbers (XA_EXTERNAL_INTERRUPT_NUMBER) for all the cores is identical. The

interrupted core resumes processing if it was blocked on the interrupt event. All messages and payloads for Inter Core IPC MSGQ should be cache-line size aligned.

Message Communication in Multicore Scenario

Application pipeline is created by the master DSP. Any command (Developer API call from application) from the App Interface Layer is sent to the DSP Interface Layer of the Master DSP through RTOS IPC channel. The App Interface Layer runs only on the Master DSP. DSP Interface Layer on the Master DSP, either routes the messages to self (its local message queue) or enqueues it into the destination core's Inter Core IPC MSGQ by using IPC locks and then interrupts the destination core. DSP Interface Layer runs on all DSPs. Processing of commands and responses by DSP Interface Layer on each DSP core is identical. Figure 2-4 shows the multicore software architecture and inter core IPC message communication details.

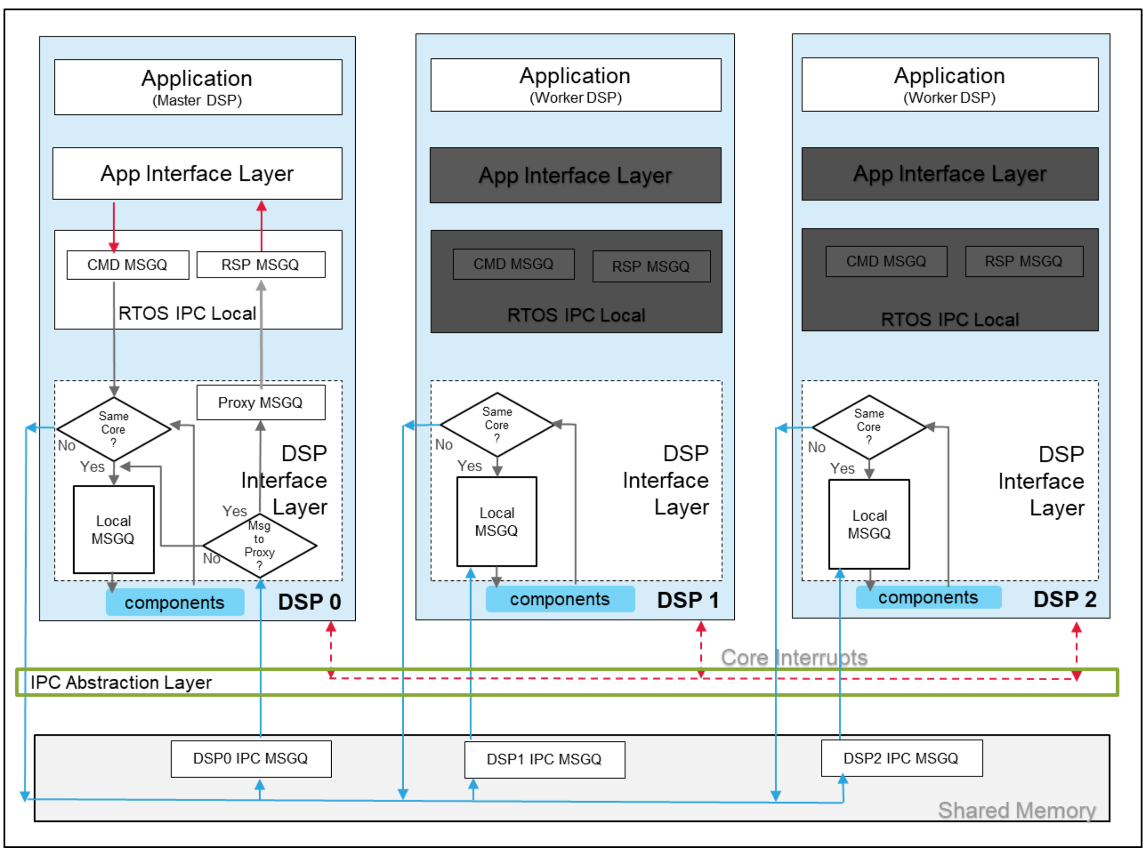


Figure 2-4 Multicore-XAF Software Architecture

DSP Interface Layer on each DSP checks for any messages in its Inter Core IPC MSGQ or from its Local MSGQ. If a message is available, it is processed by the DSP Interface Layer. Once the message is processed successfully, depending on the destination, the response is sent to either Proxy IPC MSGQ on the Master DSP if the destination is App Interface Layer or is enqueued into another core's Inter Core IPC MSGQ if the destination is a component on another core or enqueued into its own Local MSGQ if the destination component is within the same core.

Memory Architecture

Three different types of memory pools are required in multicore XAF:

- Application shared memory pool

  To allocate buffers that interact with the application. Only the Master DSP allocate these buffers.

- DSP shared memory pool
- To allocate the connect buffers between components on different DSP cores. Any DSP core can allocate buffers from this DSP shared memory pool. Allocate and free operations from this pool is protected by platform specific global lock.
- DSP local memory pool
- To allocate memory for input, persistent, scratch, stack, component buffers, event buffers and connect buffers between audio components on the same DSP core.

Figure 2-5 shows an example pipeline in which the blocks A, B, C, D are the audio components in the subsystem consisting Master DSP-0, worker DSP-1 and worker DSP2.

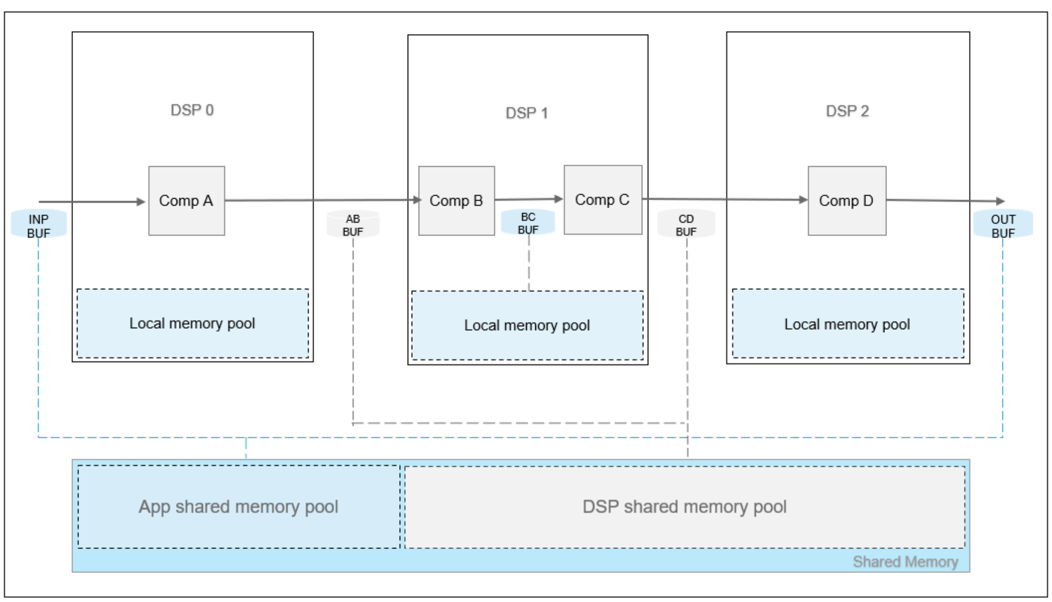


Figure 2-5 Multicore-XAF Memory Architecture

2.1.4 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 V10.2.1 [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 about downloading and building FreeRTOS for XAF.

2.1.5 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 APIs 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.
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.

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

Component Type	Input		Output		Component Description
	Ports	PCM	Ports	PCM	
MIMO	4[2]	Y	4[3]	Y	Multi-Input Multi-Output (MIMO) component process input PCM data to generate output PCM data.

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

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

2.2 Internal Architecture Details of Xtensa Audio Framework

This section provides detailed information about the 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 Interface Layer, Inter-Process Connect (IPC), and DSP Interface Layer. App Interface Layer and DSP Interface Layer pass control and data using commands and responses through Inter-Process Connect as shown in Figure 2-6.

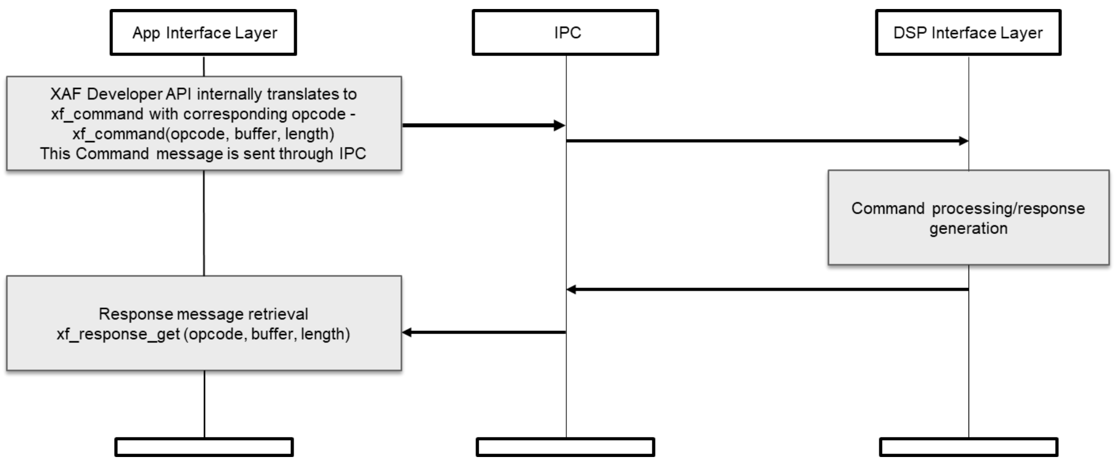


Figure 2-6 XAF Command and Response Flow

All of the XAF Developer API calls except `xaf_comp_process` and `xaf_probe_start` API calls are blocking or synchronous; that is, the API call waits for response from DSP Interface Layer for command completion. A synchronous example of XAF Developer API is `xaf_comp_set_config` API (see Table 3-8 for details). Figure 2-7 shows the control flow sequence for `xaf_comp_set_config`.

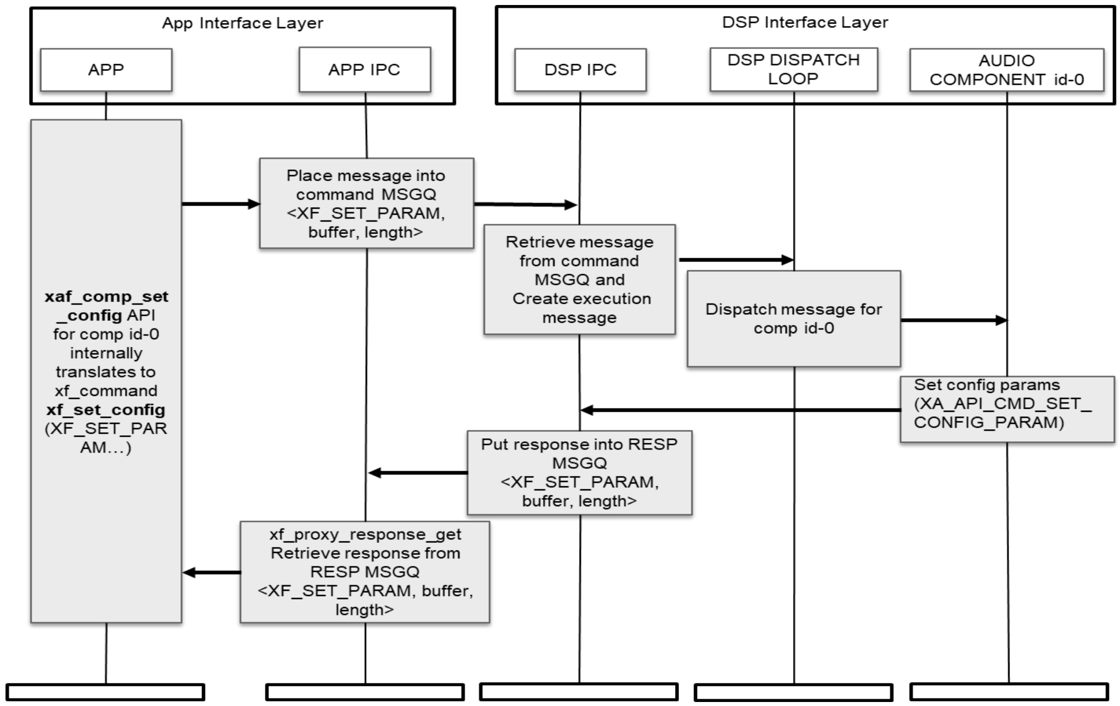


Figure 2-7 XAF Developer API `xaf_comp_set_config` Control Flow

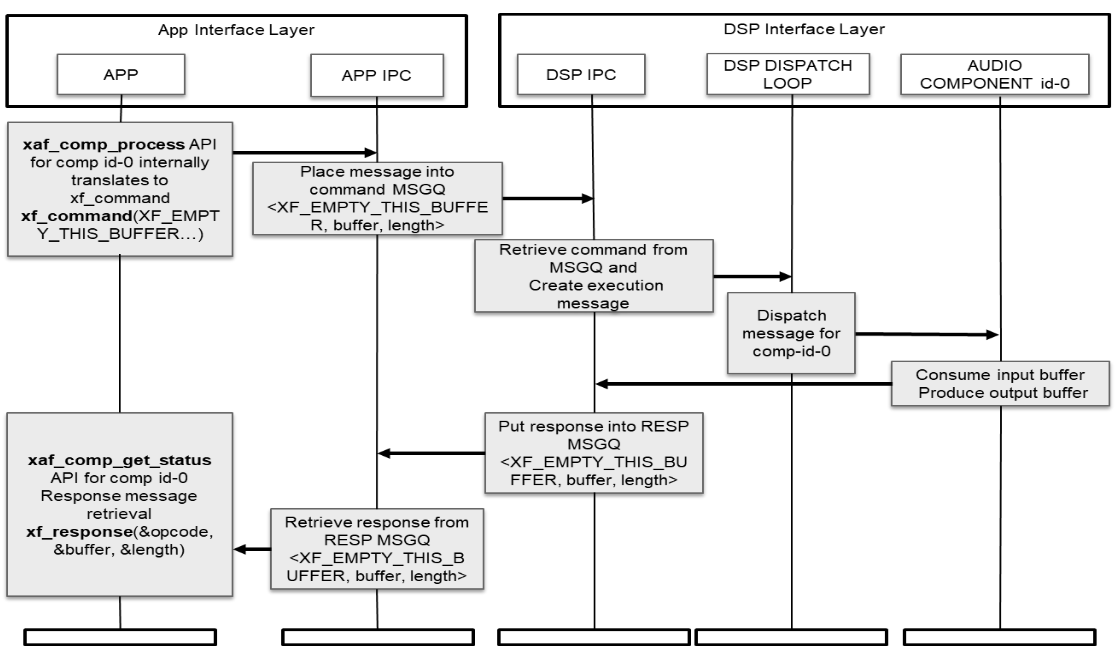


Figure 2-8 XAF Developer API `xaf_comp_process` Control Flow

XAF Developer APIs `xaf_comp_process` (see Table 3-14 for API details) and `xaf_probe_start` (see Table 3-18 for API details) are non-blocking or asynchronous. Specifically, the API call does not wait for response from DSP Interface Layer for command completion, rather the response from DSP Interface Layer can be queried for by `xaf_comp_get_status` API (see Table 3-15 for API details) at any later point of time. Figure 2-8 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 the application. Note that the `xaf_comp_get_status` API call blocks if there is any pending response on the component.

Audio components connected with each other on DSP Interface Layer also use commands and responses to share data with each other through local message queue. Note, this local message queue is internal to DSP Interface Layer and different from IPC, the API between App Interface Layer and DSP Interface Layer. The audio component communication is shown in Figure 2-9 where the 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 that for simplification and ease of understanding, Figure 2-8 and Figure 2-9 do not show all transactions.

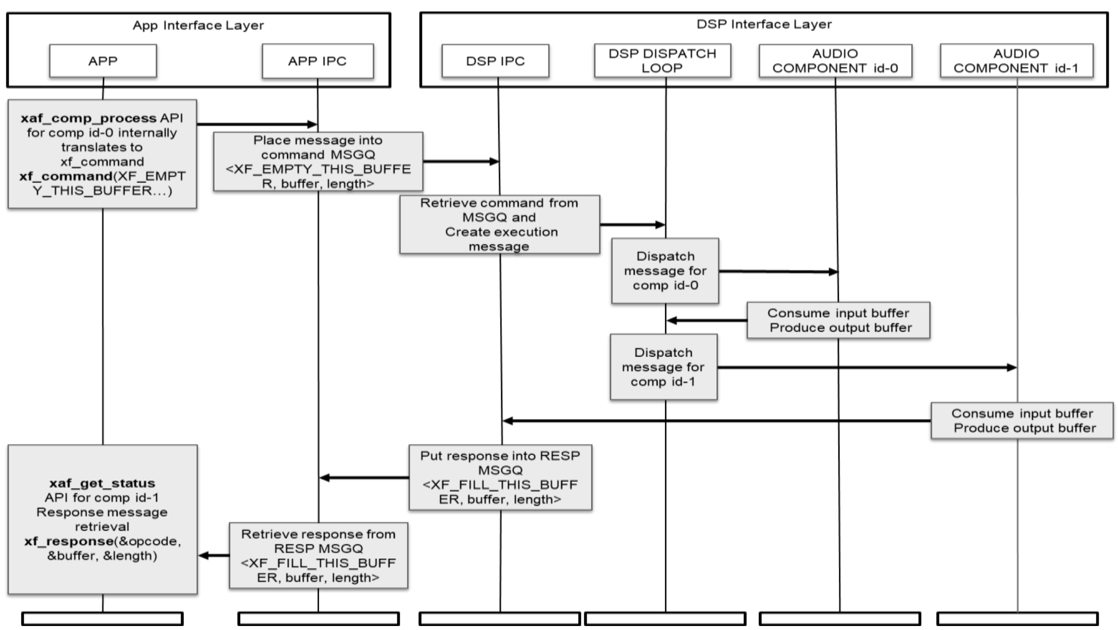


Figure 2-9 XAF Control Flow Between Audio Components

2.2.2 Control and Data Flow in Multicore XAF

Section 2.2.1 provides a generic overview of control and data flow in XAF. This section aims to provide illustrations of how components are created on master and worker DSPs in a multicore subsystem and how data flow occurs between two components that are created on two different DSP cores.

Figure 2-10 shows how components are created on the master DSP (DSP0) using `xaf_comp_create` API.

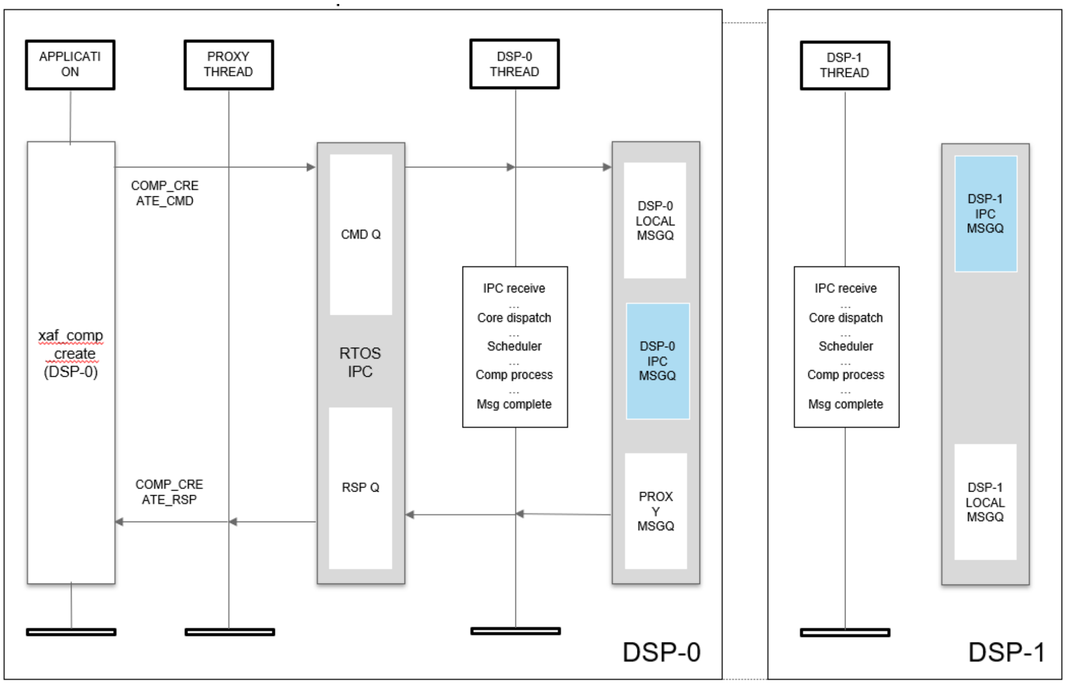


Figure 2-10 Command Flow for Component Creation on Master DSP

Figure 2-11 shows how components are created on a worker DSP (DSP1) using `xaf_comp_create` API.

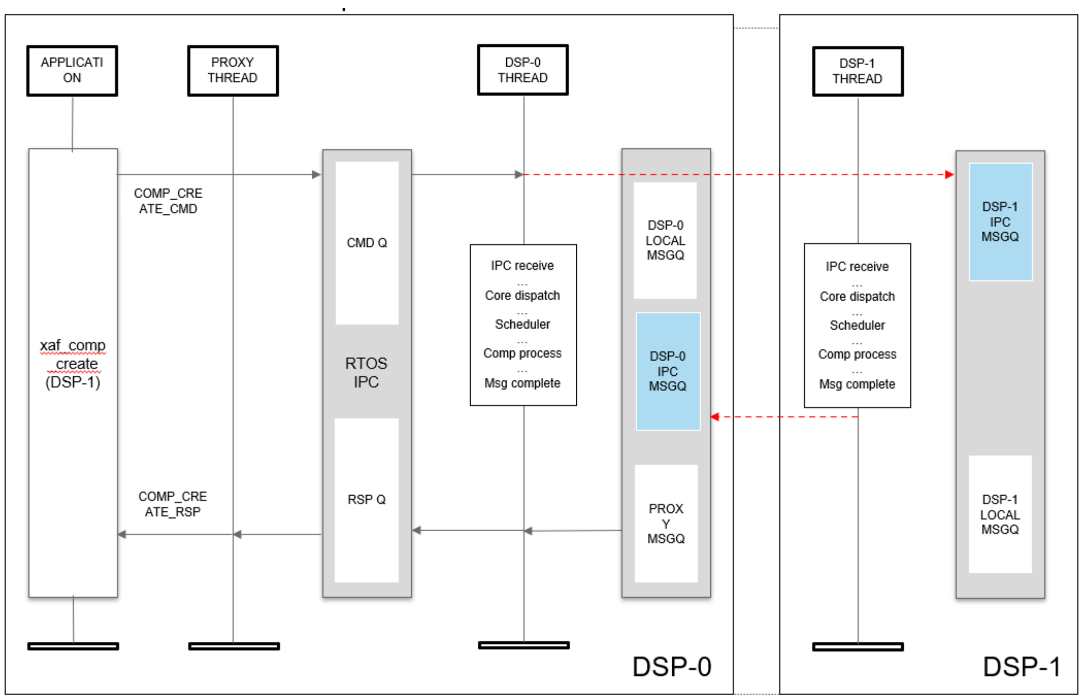


Figure 2-11 Command Flow for Component Creation on Worker DSP

Figure 2-12 shows how data flow occurs between two DSPs, namely DSP0 and DSP1.

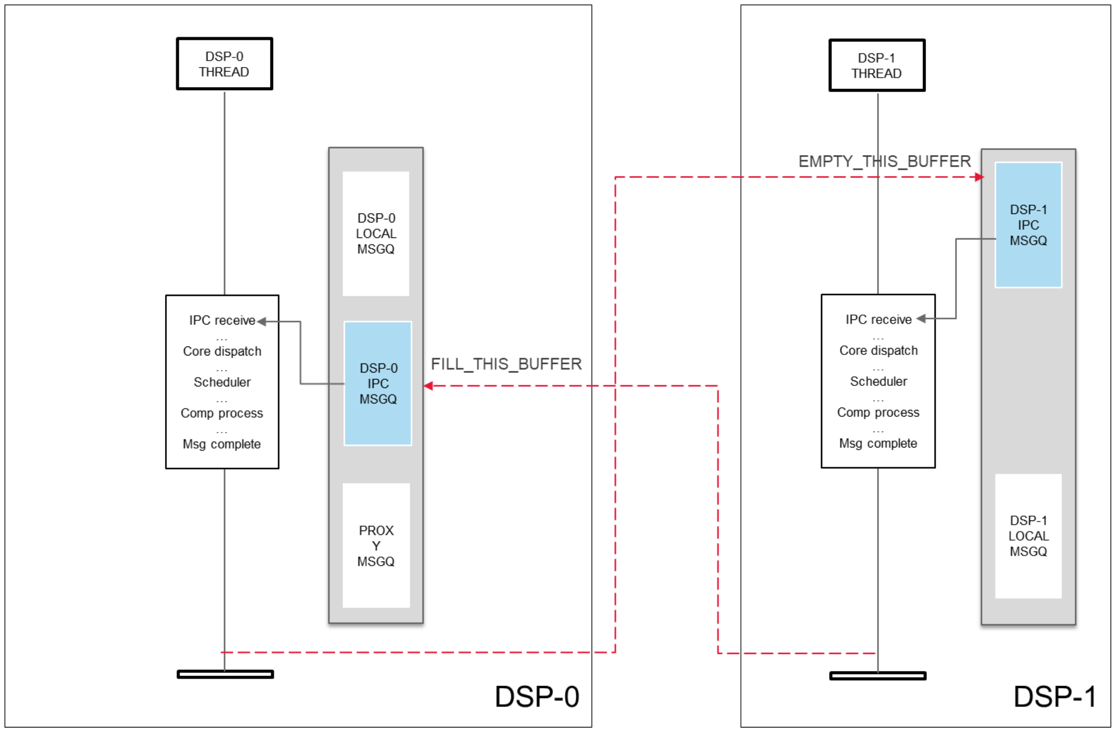


Figure 2-12 Command Flow for data processing between components on different DSPs

This diagram explains how data is processed and communicated between components on different DSPs. Similar to command flow described in Figure 2-9, the source component on DSP-0 sends data to the destination component on DSP-1 (EMPTY_THIS_BUFFER) and after consuming this input data, the destination component returns the buffer back to the source component (FILL_THIS_BUFFER). Here the red dotted lines denote Inter-DSP communication.

2.2.3 Audio Component Processing Details in DSP Interface Layer

DSP Interface Layer uses an object-oriented class like architecture for managing, scheduling, and executing various audio components as shown in Figure 2-13. 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 I/O ports. It also defines pause, resume, connect, and disconnect functionality for the class. The following derived classes are defined in the current XAF version.

- 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 or Acoustic Echo Canceler. 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.

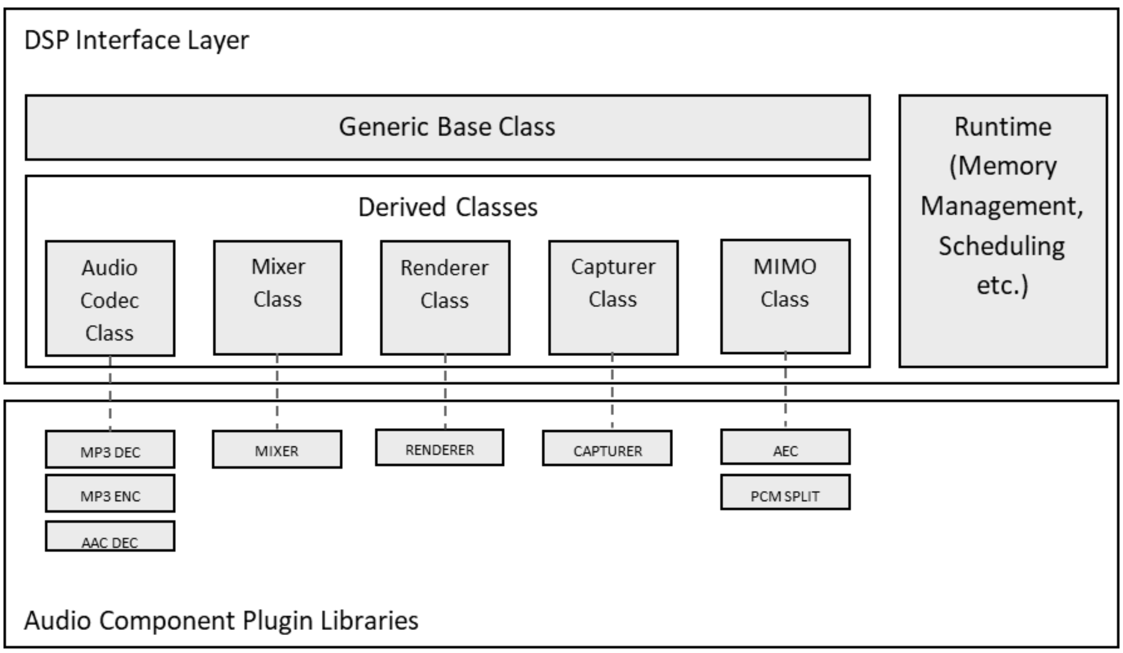


Figure 2-13 DSP Interface Layer Audio Component Architecture

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 that the 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 2-14 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 (Earliest Deadline First) scheduling policy used in post-process for rescheduling of the component is described in section 2.2.4.

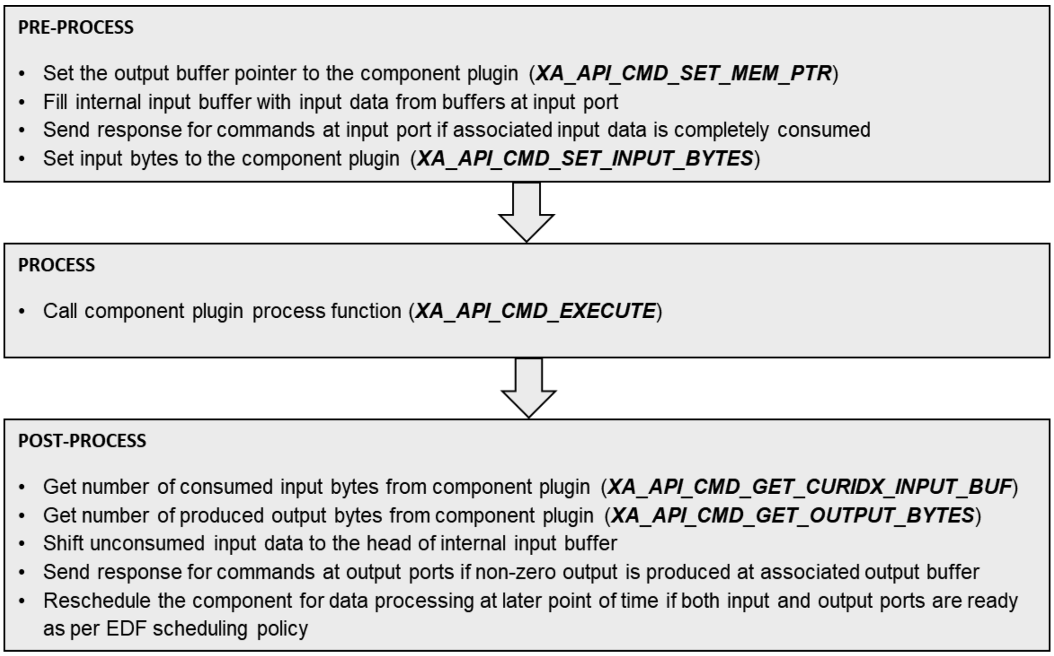


Figure 2-14 XAF Audio Codec Class Process Sequence

2.2.4 Audio Component Management

To explain XAF audio component I/O 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 the application and is connected to MP3 Encoder, and output of MP3 Encoder is sent back to the application. When PCM Gain component is created with two input buffers to receive data from the application and MP3 Encoder is created with one output buffer to send data back to the application, various buffers will be allocated in XAF as shown in Figure 2-15.

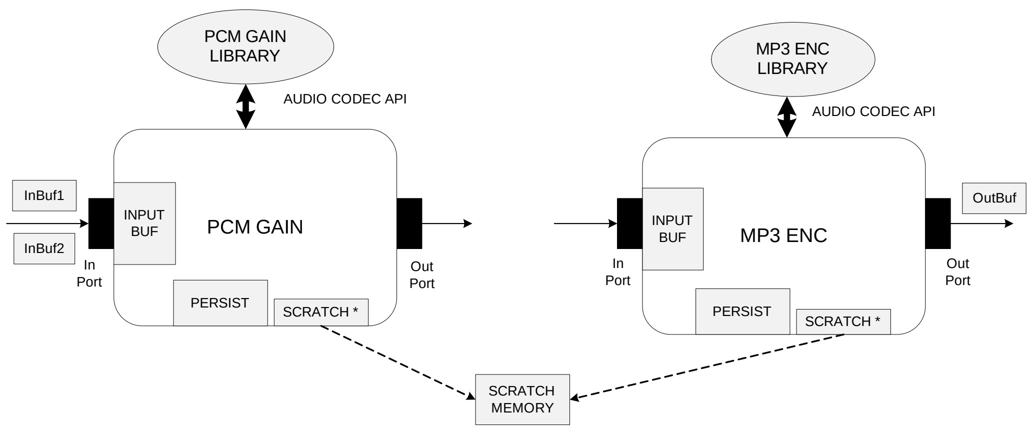


Figure 2-15 XAF Audio Components at Creation

Both PCM Gain and MP3 Encoder components have one input port and one output port, and are created as Audio Codec Class components. Normally, 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 the same priority, hence they run in the same thread context and share the scratch buffer. Note, XAF requires scratch memory size to be largest 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.

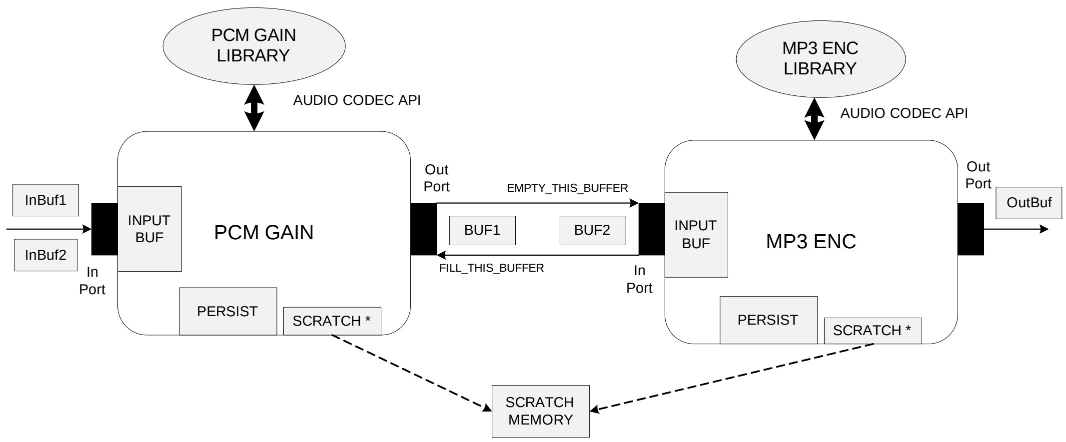


Figure 2-16 XAF Connected Audio Components

When PCM Gain component output port is connected to MP3 Encoder input port using `xaf_connect` API with two buffers (see Table 3-12 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 copied into component's internal input buffer if the buffer exists 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 component execution consumes some input data and produces some output data. If input and output ports are still ready after one execution, the component is scheduled for next execution at a later 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 the deadline measure 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 as shown in Figure 2-10, PCM Gain would get scheduled and executed four times for each execution of MP3 Encoder automatically in XAF.

2.2.5 Event Communication

XAF supports asynchronous event communication between two components, or between a component and application, or between framework and application. To enable event communication, an event channel is established using the `xaf_create_event_channel` API (Table 3-20) , and such a channel can be deleted using `xaf_delete_event_channel` API (Table 3-21). Event communication between two components can be established independent of a routed port between them.

A callback function is registered during component creation with the configuration parameter `XAF_COMP_CONFIG_PARAM_EVENT_CB`. At runtime, when source component detects an event, it notifies the framework through the callback function. Framework then queries for the associated payload from the source component using get-config-param API call. It then forms a notification message with the acquired payload and sends it to the destination component or application (as set-up by application programmer). The common message passing infrastructure already available in XAF is used for event communication as well. If the event destination is another component, upon receiving such notification message, the framework passes it to the destination component by set-config-param API call. Finally, the message (and the associated buffer) is sent back to framework (source component) for reuse. If the event destination is application, the message will be received by proxy / IPC thread in App Interface Layer. Application may register a separate callback function to receive events during device creation. If the callback function is available, application will be notified of the event and associated payload, else the event is ignored without raising any error. In either case, the message (and the associated buffer) is returned to the source.

The event channels are also used to communicate component processing errors to the application. Application developer must configure error channel creation with the configuration parameter `error_channel_ctl` during component creation with appropriate value, where `XAF_ERR_CHANNEL_DISABLE` indicates no error reporting (the default), `XAF_ERR_CHANNEL_FATAL` indicates only fatal error reporting, and `XAF_ERR_CHANNEL_ALL` indicates both fatal and non-fatal error reporting. Also between 1 and 4 error buffers of size 4 bytes each can be configured with the configuration parameter `num_err_msg_buf`.

For event to application channels (including error channels), event buffers are created at the App Interface Layer and are sent to the DSP Interface Layer during event channel creation. If an error occurs during event message handling, the error is updated onto its error field (Note, every message has an error field which is used to report errors back to the sender). Upon receiving such an error, the App Interface Layer communicates it to the application via the callback function and avoids sending the event buffer back to the DSP Interface Layer. If the event channel is an error channel, then the error code is also copied into the error buffer provided by the application. For non-error channels, the App Interface Layer sets the error flag which indicates to the application that the component is in error and any appropriate action can be taken by the application.

Note, for NCORES>1, the event channel buffers are allocated from shared memory for an event from component to application and for an event between components with source and destination on different DSPs.

The components can use the event callback function to request self-scheduling using `XAF_COMP_CONFIG_PARAM_SELF_SCHED` configuration parameter.

2.2.6 Extended set and get Config with Variable Parameter Length

XAF provides support to set and get the configuration parameters of variable length between the application and the component plugins through `xaf_set_config_ext` API (Table 3-10) and `xaf_get_config_ext` API (Table 3-11). The length of the parameter value can be more than 4 bytes and up to a maximum of 8 KB. These two APIs use a pair containing the configuration parameter ID and the pointer to `xaf_ext_buffer_t` structure.

There are two modes of using the extended set or get config API. In one mode (non-zero-copy), the parameter value from the application is copied into an internal framework buffer that is passed to the component plugin at DSP Interface Layer. In the other "zero-copy" mode, the pointer provided by the application is directly passed to the component plugin. The zero-copy mode can be activated by setting a flag (`XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY`) in `ext_config_flags`. Two macros are provided to set and clear the flag, namely `XAF_EXT_PARAM_SET_FLAG` and `XAF_EXT_PARAM_CLEAR_FLAG`.

In the normal mode (non-zero-copy), the application can configure the buffer size using `cfg_param_ext_buf_size_max` variable of the component config structure `xaf_comp_config_t` during component creation. The buffer size is used to allocate a dedicated buffer specifically for these two extended config APIs. Note that each call of `xaf_set_config_ext` or `xaf_get_config_ext` API supports up to eight configuration parameter IDs, so the size of `cfg_param_ext_buf_size_max` should be configured by the application accordingly.

2.2.7 Input Port Bypass Mode

When XAF queries for input port size, if the plugin returns a value of 0, then input-port bypass mode is activated on the component's input port. In this mode, the connect-buffer pointer is set as input buffer pointer for the component during each execution call, thus avoiding both the input-buffer copy overhead and the input buffer memory allocation.

However, for input bypass mode to work, the following is assumed: the output frame size of the preceding component should be equal to or multiple of input frame size of this component and this component must consume the data when presented for processing, even when the input length is less than input frame size (partial frame). Bypass mode is not supported for components that do not consume partial data. Note, an input port cannot be probed if bypass mode is enabled.

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.

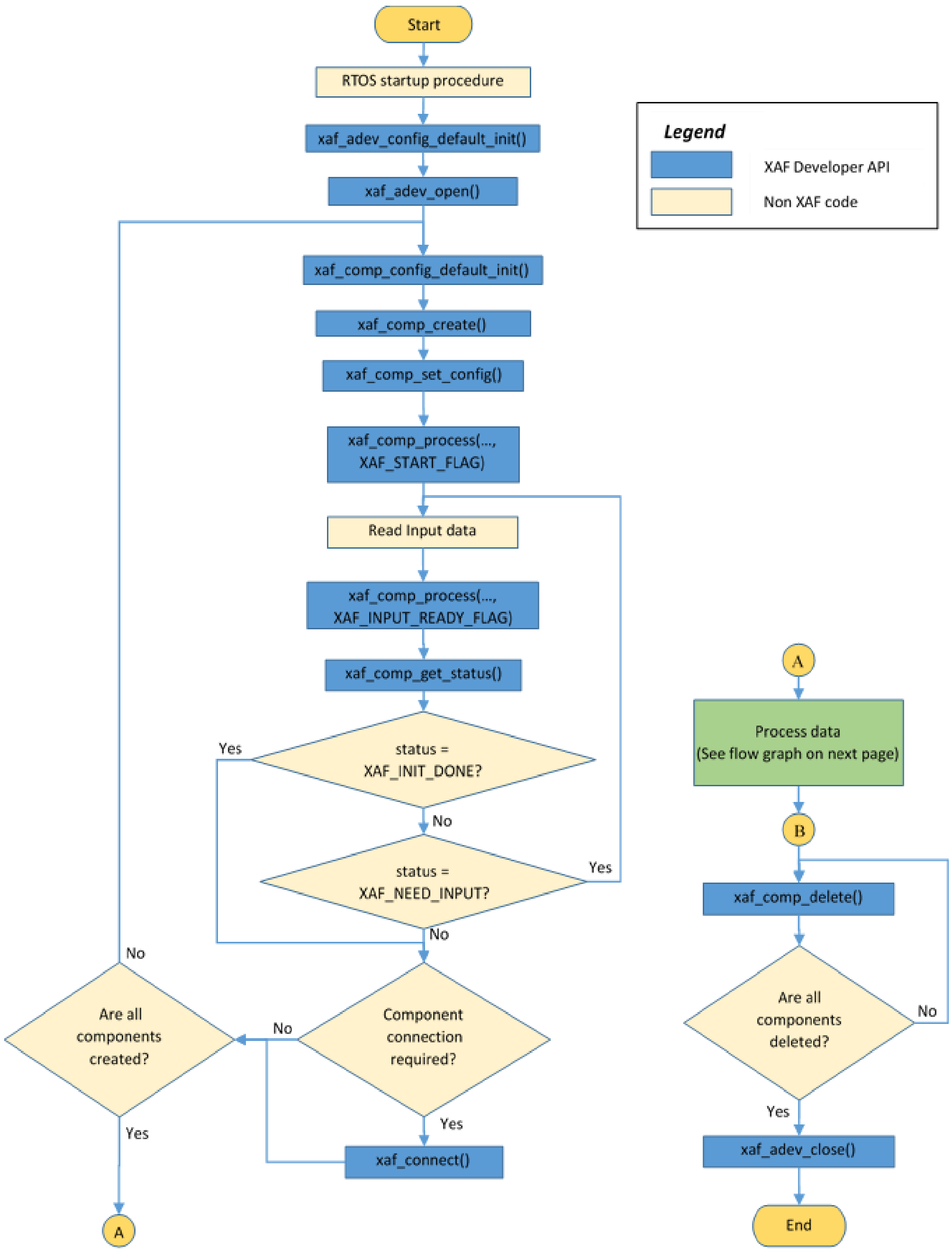
XAF Developer APIs are summarized in Table 3-1.

Table 3-1 XAF Developer APIs

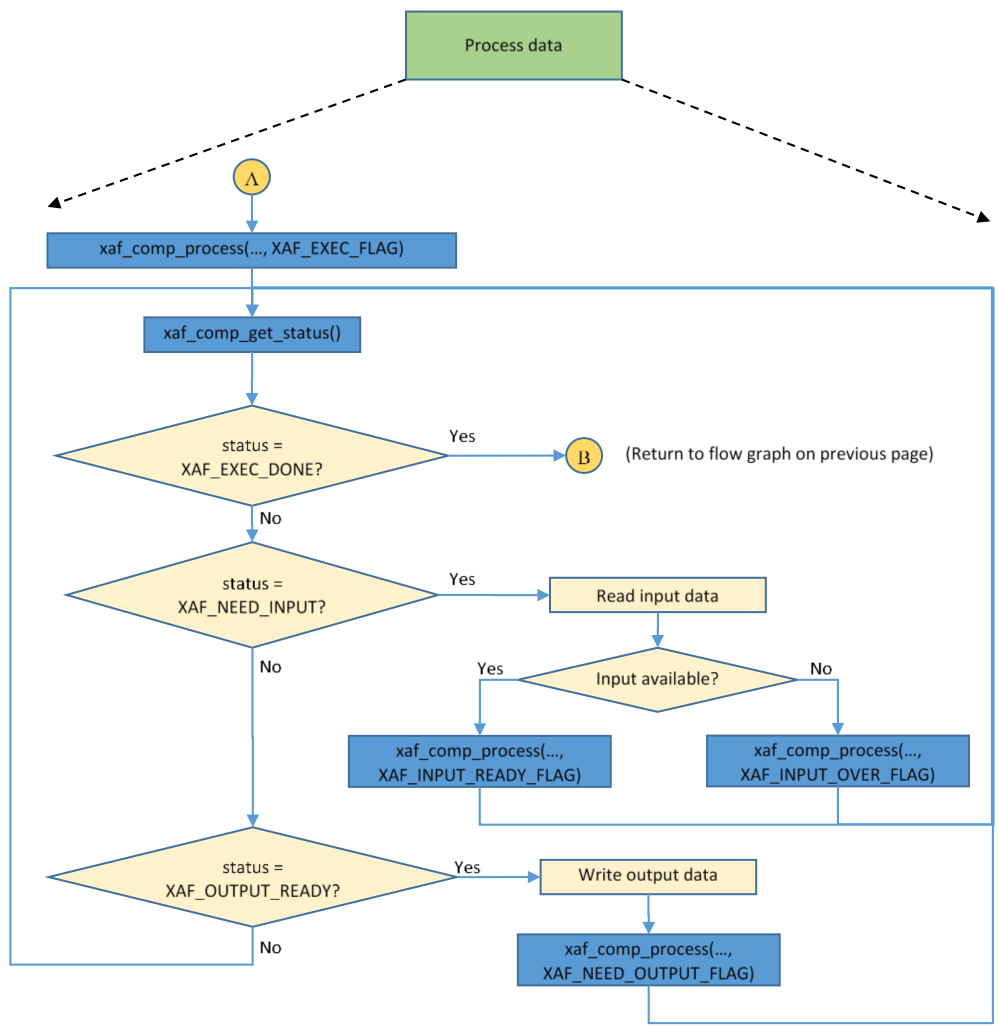
API Type	XAF Developer API	Can be called at runtime?
Startup API	xaf_adev_open	No
	xaf_dsp_open#	No
	xaf_comp_create	Yes
Configuration API	xaf_comp_set_config	Yes
	xaf_comp_get_config	Yes
	xaf_adev_set_priorities	No
	xaf_comp_set_config_ext	Yes
	xaf_comp_get_config_ext	Yes
Connect API	xaf_connect	Yes
	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_dsp_close#	No
	xaf_comp_delete	Yes
Information API	xaf_get_verinfo	Yes
	xaf_get_mem_stats	Yes
Event Communication API	xaf_create_event_channel	Yes
	xaf_delete_event_channel	Yes
Default Configuration API	xaf_adev_config_default_init	No
	xaf_comp_config_default_init	No

`xaf_dsp_open`, `xaf_dsp_close` APIs should be used by worker core application. These APIs are available only with NCORES > 1.

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 on Master Core

Following is a brief description of the flowgraph sequence on master core:

- Initialize XAF: The XAF is initialized by calling `xaf_adev_config_default_init` which updates the default values for `xaf_adev_config_t` structure parameters. This is followed by a call to `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_config_default_init` which updates the default values for `xaf_comp_config_t` structure parameters. This is followed by a call to `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 the XAF_START_FLAG flag and connected using xaf_connect.

Note: In general, audio decoder components require input data during initialization to determine input stream parameters, such as sample rate or number of channels. So the initialization loop shown in

- Figure 3-1 (a) that feeds input data to the component during initialization is required only for those audio decoder components which need input data to initialize, and such loop is not required for encoder or PCM data processing components or for the audio decoder components that can be initialized without input data.
- Process Data: Input and output data is passed to the components using `xaf_comp_process`. This must be performed only for components that must be supplied with input/output data (typically the edge components of the chain). The component status should be queried using `xaf_comp_get_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. Note, the component needs to be configured to enable probe feature before these APIs can be used run time.
    - Disconnect and reconnect components: Any connected output ports of a component can be dynamically disconnected by using `xaf_disconnect` API. Components also can be connected or reconnected dynamically by using `xaf_connect` API.
    - Event communication: An asynchronous event communication channel can be established between two components or between a component and application with `xaf_create_event_channel` API and the same can be deleted with `xaf_delete_event_channel` API.

- Self-scheduling: The components can request self-scheduling by raising the `XAF_COMP_CONFIG_PARAM_SELF_SCHED` event to framework.

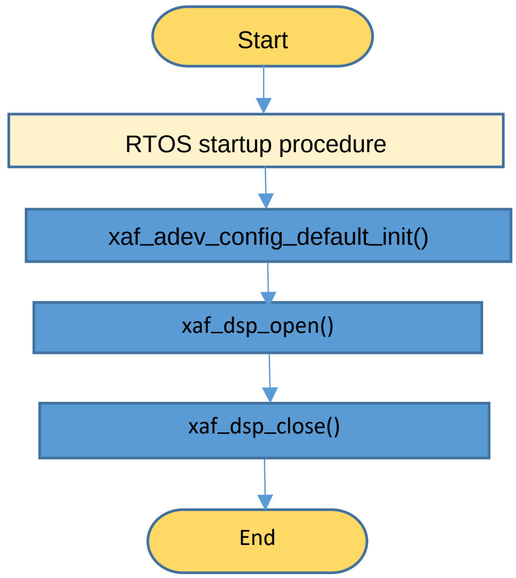


Figure 3-2 Flowgraph Sequence for API Calls of Testbench on Worker DSP

Following is a brief description of the flowgraph sequence on worker DSP cores:

- Open worker DSP: The worker DSP is initialized by calling `xaf_adev_config_default_init`, which updates the default values for `xaf_adev_config_t` structure parameters. This is followed by a call to_ `xaf_dsp_open`. All the required memory is allocated at this stage. A DSP-thread is created in this step.
- Close worker DSP: In the execution sequence, the worker core application blocks on the join call to the DSP-thread in `xaf_dsp_close`. Once the DSP-thread is joined, all the allocated memory is freed.

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_INVALIDPTR_ERR

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

- XAF_INVALIDVAL_ERR

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

- XAF_RTOS_ERR

  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 an incorrect response is received from the DSP Interface Layer for command sent by the XAF Developer API.

- XAF_MEMORY_ERR

  This error code indicates an internal error, caused due to memory allocation failure or availability issue.

3.2.2 Specific Errors

The following errors are specific to some APIs.

- XAF_ROUTING_ERR

  This error code indicates that the XAF Developer API `xaf_connect()` or `xaf_disconnect()` did not successfully connect or disconnect the two requested components.

- XAF_TIMEOUT_ERR

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

3.2.3 Component Processing Errors

The following APIs can return component processing error when it occurs:

`xaf_comp_get_config, xaf_comp_set_config,`

`xaf_comp_get_config_ext, xaf_comp_set_config_ext,`

`xaf_pause, xaf_resume,`

`xaf_connect, xaf_disconnect`

Note For `xaf_comp_set_config, xaf_comp_get_config, xaf_comp_set_config_ext, and xaf_comp_get_config_ext` APIs, the error returned is the first non-fatal error from the component when there are multiple parameters being queried and if multiple non-fatal errors occur.

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, xaf_adev_config_t *pconfig);`
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 Interface Layer and starts the IPC thread. It also allocates local memory to be used by the audio components on DSP Interface Layer and shared memory for communication between App Interface Layer and DSP Interface Layer.

<table>
<tr><td>Actual Parameters</td><td>

pp_adev

Address of pointer to audio device. This API call allocates memory for audio device and update this pointer with it.

pconfig

Pointer to an initialized structure that contains the necessary parameters for this API. The structure members are as follows:

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>UWORD32 audio_framework_buffer_size</td><td>Size of memory to be allocated for shared buffers and structures between App Interface Layer and DSP Interface Layer. This size must be aligned to 64 bytes and greater than or equal to 16 kB (for XAF structures). Refer to Section 7 for more details on memory guidelines.</td></tr>
<tr><td>UWORD32 audio_component_buffer_size</td><td>Size of memory to be allocated for various audio component buffers and structures required locally on DSP Interface Layer. This size must be aligned to 64 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.</td></tr>
<tr><td>xaf_mem_malloc_fxn_t *pmem_malloc</td><td>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.</td></tr>
<tr><td>xaf_mem_free_fxn_t *pmem_free</td><td>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);</td></tr>
</table>

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

	UWORD32 proxy_thread_priority	Priority level for the proxy thread at the App Interface Layer. This value must be greater than DSP thread priority.
	UWORD32 dsp_thread_priority	Priority level for the dsp thread at the DSP Interface Layer. This value must be lower than proxy thread priority.
	UWORD32 worker_thread_scratch_size[XAF_MAX_WORKER_THREADS]	Array of scratch memory sizes for worker threads.

	xaf_app_event_handler_fxn_t app_event_handler_cb	Callback function registered by application to receive events. If no application events required, this can be set to NULL and the events, if any are dropped at the App Interface Layer.
		
		`WORD32 (*xaf_app_event_handler_fxn_t)(pVOID comp_ptr, UWORD32 config_param_id, pVOID config_buf_ptr, UWORD32 buf_size, UWORD32 comp_error_flag)`
		
		`comp_ptr`
		Handle of component handle associated with the event.
		
		`config_param_id`
		Configuration parameter id for event to application.
		
		`config_buf_ptr`
		Event buffer pointer.
		
		`buf_size`
		Size of event buffer in bytes.
		
		`comp_error_flag`
		Indicates whether the event is error message(1) or not(0). For non-error event channels it indicates that the component is in fatal error.

Parameter (NCORES>1)	Description
`UWORD32 core`	Core ID of master core
`void *pshmem_frmwk`	Pointer to pre-allocated global shared memory used as framework buffer. The size of the buffer is defined by `audio_framework_buffer_size`.
`void *pshmem_dsp`	Pointer to pre-allocated global shared memory used as dsp shared memory buffer.
`UWORD32 audio_shmem_buffer_size`	Size of shared memory block used to allocate connect buffers and event buffers between cores and the shared structures. Refer to section 7 for more details on memory guidelines.
`int (*cb_compute_cycles)`	Function pointer to the computation routine which calculates the worker-thread cycles for all DSPs and also framework cycles for worker-DSP. This must have proto-type `int cb_total_frmwrk_cycles(xaf_perf_stats_t *cb_stats);` 1. Called during execution of `xaf_adev_close` to calculate worker-thread cycles for all DSPs 2. Called from `xaf_dsp_close` to calculate worker DSP cycles. Refer to `cb_total_frmwrk_cycles` function in `test/src/xaf-clk-test.c`
`xaf_perf_stats_t *cb_stats`	Pointer to a `xaf_perf_stats_t` structure that contains necessary parameters for calculating framework MCPS and memory consumption for the given core. The application must provide a valid structure pointer. This is updated by `xaf_dsp_close` with Worker-DSP memory stats.

The structure members are as follows:

Parameter	Description
`long long tot_cycles`	Total cycles consumed by all rtos threads
`long long frmwk_cycles`	Cycles consumed by framework
`long long dsp_comps_cycles`	Cycles consumed by DSP components
`int dsp_frmwk_buf_size_peak`	Peak usage of audio-framework buffer in the subsystem.
`int dsp_comp_buf_size_peak`	Peak usage of component buffer on a DSP
`int dsp_shmem_buf_size_peak`	Peak usage of shared memory buffer in the subsystem
`int dsp_xaf_buf_size_peak`	Local Memory used by framework structures on worker DSP.

Restrictions	■ **Prerequisite:** The RTOS startup procedure must be invoked before calling this function. Procedures for XOS and FreeRTOS are as follows. ■ 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, &adev_config);
```

Errors

- Common API Errors

Table 3-3 `xaf_adev_config_default_init` API

API	`XAF_ERR_CODE xaf_adev_config_default_init(xaf_adev_config_t *pconfig)`
Description	This API sets default values for audio device configuration.
Actual Parameters	`p_config` Pointer to an initialized `xaf_adev_config_t` structure. <table><tr><th>Structure variable</th><th>Default value</th></tr><tr><td>audio_component_buffer_size</td><td>512 KB</td></tr><tr><td>audio_framework_buffer_size</td><td>256 KB</td></tr><tr><td>proxy_thread_priority</td><td>XAF_PROXY_THREAD_PRIORITY(6)</td></tr><tr><td>dsp_thread_priority</td><td>XAF_DSP_THREAD_PRIORITY(5)</td></tr><tr><td>worker_thread_scratch_size</td><td>56 KB</td></tr><tr><td>pmem_malloc</td><td>NULL</td></tr><tr><td>pmem_free</td><td>NULL</td></tr><tr><td>app_event_handler_cb</td><td>NULL</td></tr><tr><th>Structure variable(NCORES > 1)</th><th>Default value</th></tr><tr><td>Core</td><td>MASTER_CORE_ID</td></tr><tr><td>pshmem_frmwk</td><td>NULL</td></tr><tr><td>pshmem_dsp</td><td>NULL</td></tr><tr><td>audio_shmem_buffer_size</td><td>0</td></tr><tr><td>cb_compute_cycles</td><td>NULL</td></tr><tr><td>cb_stats</td><td>NULL</td></tr></table>
Restrictions	Should be called before `xaf_adev_open` API

Example

```
ret = xaf_adev_config_default_init(&adev_config);
```

Errors

- Common API Errors

Table 3-4 `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. The device should be force closed *only* for a fatal error condition (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 3-5 `xaf_comp_create` API

API	`XAF_ERR_CODE xaf_comp_create(pVOID p_adev, pVOID *pp_comp, xaf_comp_config_t *pconfig);`
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 I/O buffer requirement is dependent upon the position of the component in the audio processing chain; see the parameter description for details.

<table>
<tr><td>Actual Parameters</td><td>

`p_adev`
Pointer to the audio device structure

`pp_comp`
Address of pointer to the audio component structure

`p_config`
Pointer to an initialized structure that contains the necessary parameters for this API. The structure members are as below:

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>xf_id_t comp_id</td><td>Component identifier string. e.g. “mixer”, “audio-decoder/mp3”, etc. It should match with <code>class_id</code>s defined under the constant definition of <code>xf_component_id</code> in <code>xa-factory.c</code> file (Refer to section 5 for details on how to add a new audio component in XAF).</td></tr>
<tr><td><code>xaf_comp_type comp_type</code></td><td>Type of audio component. Following are valid values:

Type — Description
<code>XAF_DECODER</code>: Decoder component
<code>XAF_ENCODER</code>: Encoder component
<code>XAF_MIXER</code>: Mixer component
<code>XAF_PRE_PROC</code>: Preprocessing component
<code>XAF_POST_PROC</code>: Post processing component
<code>XAF_RENDERER</code>: Renderer component
<code>XAF_CAPTURER</code>: Capturer component
<code>XAF_MIMO_PROC_12</code>: MIMO component with 1 input and 2 output ports
<code>XAF_MIMO_PROC_21</code>: MIMO component with 2 input and 1 output ports
<code>XAF_MIMO_PROC_22</code>: MIMO component with 2 input and 2 output ports
<code>XAF_MIMO_PROC_23</code>: MIMO component with 2 input and 3 output ports
<code>XAF_MIMO_PROC_10</code>: MIMO component with 1 input and 0 output ports
<code>XAF_MIMO_PROC_11</code>: MIMO component with 1 input and 1 output ports</td></tr>
<tr><td><code>UWORD32 num_input_buffers</code></td><td>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.</td></tr>
</table>

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

<table>
<tr><td></td><td>
<table>
<tr><td>UWORD32 num_output_ buffers</td><td>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.</td></tr>
<tr><td>pVOID (*pp_inbuf)[XAF_MAX_ INBUFS]</td><td>Pointer to the array to hold <code>ninbuf</code> input buffer addresses that have been allocated within XAF. If the pointer is NULL, the input buffer addresses will not be returned.</td></tr>
<tr><td>UWORD32 error_ channel_ctl</td><td>Variable to indicate what type of error channel to be created.
<table>
<tr><td>Enum</td><td>Numerical value</td><td>Type of error channel</td></tr>
<tr><td>XAF_ERR_CHANNEL _DISABLE</td><td>0</td><td>Will not create error channel</td></tr>
<tr><td>XAF_ERR_CHANNEL _FATAL</td><td>1</td><td>Error channel will only report fatal error</td></tr>
<tr><td>XAF_ERR_CHANNEL _ALL</td><td>2</td><td>Error channel will only report fatal and non-fatal error</td></tr>
</table>
</td></tr>
<tr><td>UWORD32 num_err_msg_ buf</td><td>Unsigned integer indicating the number of error buffers that are allocated to capture fatal or non-fatal errors.
Valid values: 1, 2, 3, 4.
Default value: 2</td></tr>
<tr><td>Parameter (NCORES > 1)</td><td>Description</td></tr>
<tr><td>UWORD32 core</td><td>Core ID of the component.</td></tr>
</table>
</td></tr>
<tr><td>Restrictions</td><td>Should not be called before <code>xaf_adev_open</code> API</td></tr>
</table>

Example

```
ret = xaf_comp_create(p_adev, pp_comp, &comp_config);
```

Errors

- Common API Errors

Table 3-6 `xaf_comp_config_default_init` API

<table>
<tr><td>API</td><td><code>XAF_ERR_CODE</code>
<code>xaf_comp_config_default_init(xaf_comp_config_t *pconfig)</code></td></tr>
<tr><td>Description</td><td>This API sets default values for component configuration.</td></tr>
<tr><td>Actual Parameters</td><td><code>p_config</code>
Pointer to an initialized <code>xaf_comp_config_t</code> structure.
<table>
<tr><td>Structure variable</td><td>Default value</td></tr>
<tr><td><code>comp_id</code></td><td>"post-proc/pcm_gain"</td></tr>
<tr><td><code>comp_type</code></td><td><code>XAF_POST_PROC</code></td></tr>
<tr><td><code>num_input_buffers</code></td><td>2</td></tr>
<tr><td><code>num_output_buffers</code></td><td>1</td></tr>
<tr><td><code>error_channel_ctl</code></td><td><code>XAF_ERR_CHANNEL_DISABLE(0)</code></td></tr>
<tr><td><code>num_err_msg_buf</code></td><td>2</td></tr>
<tr><td><code>pp_inbuf</code></td><td><code>NULL</code></td></tr>
<tr><td><code>cfg_param_ext_buf_size_max</code></td><td>0</td></tr>
<tr><td>Structure variable (NCORES > 1)</td><td>Default value</td></tr>
<tr><td><code>core</code></td><td>XF_CORE_ID_MASTER</td></tr>
</table></td></tr>
<tr><td>Restrictions</td><td>Should be called before <code>xaf_comp_create</code> API</td></tr>
</table>

Example

```
ret = xaf_comp_config_default_init(&comp_config);
```

Errors

- Common API Errors

Table 3-7 `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. **Note:** This API deletes any associated event channel with the component before initiating component deletion.

Example

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

Errors

- Common API Errors

Table 3-8 `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 32.
	
	`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
- Non-fatal error from component.

Table 3-9 `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 32.
	
	`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
- Non-fatal error from component.

Table 3-10 `xaf_comp_set_config_ext` API

<table>
<tr><td>API</td><td><pre>XAF_ERR_CODE xaf_comp_set_config_ext(pVOID p_comp,
                                     WORD32 num_param,
                                     WORD32 *p_param)</pre></td></tr>
<tr><td>Description</td><td>This API sets (writes) configuration parameters to the audio component of variable length.
num_param provides the number of configuration parameters to be set.
p_param points to an integer array containing ID and pointer to
xaf_ext_buffer_t structure pairs for all num_param parameters.
For example, for two parameters, p_param will contain ID1, EXT_BUF_PTR1, ID2, EXT_BUF_PTR 2.

Note, this API is intended for variable length data transfer between application and component plugins and not a replacement for the xaf_comp_set_config API used for component initialization or framework specific configuration parameters as discussed in Section 3.4.</td></tr>
<tr><td>Actual Parameters</td><td>p_comp
Pointer to the audio component structure

num_param
Integer containing the number of parameters to be set. The maximum number of parameters allowed per API call is 8.

p_param
Pointer to an integer array containing ID and a pointer to xaf_ext_buffer_t structure pairs.
xaf_ext_buffer_t structure has following members
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>UWORD32 max_data_size</td><td>Maximum data size that can be read or written</td></tr>
<tr><td>UWORD32 valid_data_size</td><td>Valid data size that can be read or written</td></tr>
<tr><td>UWORD32 ext_config_flags</td><td>XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY (bit offset 0) to indicate zero copy mode (refer to section 2.2.5).</td></tr>
<tr><td>UWORD8 *data</td><td>Pointer to data buffer</td></tr>
</table></td></tr>
<tr><td>Restrictions</td><td>Should not be called before xaf_comp_create API.</td></tr>
</table>

Example

```
/*… Test Application Part */
int data0[4], data1[6];               /*…     Variable     length
parameters */
WORD32 param_ext [N_PARAMS * 2 ];    /*… N_PARAMS = 2 */
bool is_shared_mem = false;      /* …flag for cache management */
int *p_shared_data0 = NULL;
int shared_mem_used=0;                /* …cumulative bytes used */
extern void *p_shared_mem;    /* …global shared memory pointer */
if((NCORES>1) && (XA_ZERO_COPY))     /*… XA_ZERO_COPY=1 if 'Zero
copy mode' feature is to be used for any one of the parameter */
{
      /*…shared memory is required only if parameter is to be set
on component on another DSP. */
      xaf_ext_buffer_t       *ext_buf=(xaf_ext_buffer_t       *)
p_shared_mem;
      shared_mem_used += (N_PARAMS *sizeof(xaf_ext_buffer_t));
      is_shared_mem = true;

      p_shared_data0 = (int) p_shared_mem + shared_mem_used;
      shared_mem_used += sizeof(data0);
      memcpy(p_shared_data0, &data0, sizeof(data0));
      XF_IPC_FLUSH(p_shared_data0,  sizeof(data0));  /*  …flush
shared memory before sending */
}
else
      xaf_ext_buffer_t ext_buf [N_PARAMS];

ext_buf[0].max_data_size = sizeof(data0);
ext_buf[0].valid_data_size = sizeof(data0);
ext_buf[0].ext_config_flags      |=      XAF_EXT_PARAM_SET_FLAG
(XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY);
if(p_shared_data0)
      ext_buf[0].data = (UWORD8 *)p_shared_data0; /* …pass the
pointer to component */
else
      ext_buf[0].data = (UWORD8 *)data0;

/* …data1 is usefor non-ZERO_COPY example */
ext_buf[1].max_data_size = sizeof(data1);
ext_buf[1].valid_data_size = sizeof(data1);
ext_buf[1].ext_config_flags     &=     XAF_EXT_PARAM_CLEAR_FLAG
(XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY);
ext_buf[1].data = (UWORD8 *)data1;

param_ext[0] = CONFIG_PARAM_ID0;
```

```
param_ext[1] = (WORD32)&ext_buf[0];
param_ext[2] = CONFIG_PARAM_ID1;
param_ext[3] = (WORD32)&ext_buf[1];


if(is_shared_mem == true)
      XF_IPC_FLUSH(ext_buf, sizeof(xaf_ext_buffer_t));

 ret = xaf_comp_set_config_ext(p_comp,
                               N_PARAMS,
                               &param_ext[0]);

/*… Plugin or component part */
#include “api.h”
xa_set_param (WORD32 param_id, void *p val)
{
if(param_id == CONFIG_PARAM_ID0)
{
      xaf_ext_buffer_t *ext_buf = p_val;
if((NCORES>1) && (XA_ZERO_COPY)){
      XF_IPC_INVALIDATE(ext_buf, sizeof(xaf_ext_buffer_t));
      XF_IPC_INVALIDATE(ext_buf->data,                  ext_buf-
>valid_data_size));
      }
      /* …copy parameters from ext_buf for further use in the
plugin */
}
}
```

Errors

- Common API Errors
- Non-fatal error from component.

Table 3-11 `xaf_comp_get_config_ext` API

API	`XAF_ERR_CODE xaf_comp_get_config_ext(pVOID p_comp,`
	`WORD32 num_param,`
	`WORD32 *p_param)`

Description	This API gets (reads) configuration parameters of variable length from the component. ▪ `num_param` provides the number of configuration parameters to get. `p_param` points to an integer array containing ID and a pointer to `xaf_ext_buffer_t` structure pairs for all `num_param` parameters. ▪ For example, for two parameters, p_param will contain ID1, EXT_BUF_PTR1, ID2, EXT_BUF_PTR 2. Upon successful execution of this API, `xaf_ext_buffer_t` structure field of `p_param` will be updated with values received from the 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 8.
	
	`p_param`
	Pointer to an integer array containing ID and a pointer to `xaf_ext_buffer_t` structure pairs.
	▪ For the `xaf_ext_buffer_t` structure details, refer to Table 3-10 xaf_comp_set_config_ext API.

Restrictions	Should not be called before `xaf_comp_create` API.

Example

```
	/* … Test Application Part */
	int data0[4], data1[6];              /*… Variable length Data */
	WORD32 param_ext [N_PARAMS * 2 ];    /*… N_PARAMS = 2 */

	bool is_shared_mem = false;     /* …flag for cache management */
	int *p_shared_data0 = NULL;
	int shared_mem_used=0;               /* …cumulative bytes used */
	extern void *p_shared_mem;    /* …global shared memory pointer */
	if((NCORES>1) && (XA_ZERO_COPY))     /*… XA_ZERO_COPY=1 if 'Zero
copy mode' feature is to be used for any of the parameter */
	{
		/*…shared memory is required only if parameter is to be set
	on component on another DSP. */
		xaf_ext_buffer_t       *ext_buf=(xaf_ext_buffer_t       *)
	p_shared_mem;
		shared_mem_used += (N_PARAMS * sizeof(xaf_ext_buffer_t));
		is_shared_mem = true;

		p_shared_data0 = (int) p_shared_mem + shared_mem_used;
		shared_mem_used += sizeof(data0);
	}
	else
		xaf_ext_buffer_t ext_buf [N_PARAMS];
	ext_buf[0].max_data_size = sizeof(data0);
	ext_buf[0].valid_data_size = 0;
	ext_buf[0].ext_config_flags      |=      XAF_EXT_PARAM_SET_FLAG
(XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY);
	if(p_shared_data0)
		ext_buf[0].data = (UWORD8 *) p_shared_data0;
	else
		ext_buf[0].data = (UWORD8 *)data0;

	/*     …data1     is     usefor     non-ZERO_COPY     example
*/ext_buf[1].max_data_size = sizeof(data1);
	ext_buf[1].valid_data_size = 0;
	ext_buf[1].ext_config_flags     &=     XAF_EXT_PARAM_CLEAR_FLAG
(XAF_EXT_PARAM_FLAG_OFFSET_ZERO_COPY);
	ext_buf[1].data = (UWORD8 *)data1;

	param_ext[0] = CONFIG_PARAM_ID0;
	param_ext[1] = (WORD32)&ext_buf[0];
	param_ext[2] = CONFIG_PARAM_ID1;
	param_ext[3] = (WORD32)&ext_buf[1];
	if(is_shared_mem == true)
		XF_IPC_FLUSH(ext_buf, sizeof(xaf_ext_buffer_t));
```

```
 ret = xaf_comp_get_config_ext(p_comp,
                               N_PARAMS,
                               &param_ext[0]);
if(p_shared_data0){
XF_IPC_INVALIDATE(ext_buf[0].data,  ext_buf[0].valid_data_size);
}
/* … copy the parameters from ext_buf for further use here on */

/*… Plugin or component part */
#include “api.h”
xa_get_param (WORD32 param_id, void *p val)
{
if(param_id == CONFIG_PARAM_ID0)
{
      xaf_ext_buffer_t *ext_buf = p_val;
      XF_IPC_INVALIDATE(ext_buf, sizeof(xaf_ext_buffer_t));
      memcpy(ext_buf->data, local_data0, sizeof(local_data0));
      XF_IPC_FLUSH(ext_buf->data, ext_buf->valid_data_size));
      XF_IPC_FLUSH(ext_buf, sizeof(xaf_ext_buffer_t));
}
}
```

Errors

- Common API Errors
- Non-fatal error from component.

Table 3-12 `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 that the 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.

Shared memory is used as connect buffers between components on different DSPs.

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: 1 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_ERR

  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.)
- Non-fatal error from component.

Table 3-13 `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 is 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 flushes the output port and frees buffers and message pool between ports, but does not cancel any pending processing of the component. Furthermore, 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 the 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_ERR

  Indicates that the API failed to disconnect the two requested ports (due to invalid port numbers, invalid components, or uninitialized source component, etc.)
- Non-fatal error from component.

Table 3-14 `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 must 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; that is, it delivers the process command to the audio component and returns. The audio component will process this request when all required resources (I/O 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 . **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>.</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>.</td></tr>
</table>
</td></tr>
<tr><td>Restrictions</td><td>Should not be called before <code>xaf_comp_create</code> API</td></tr>
</table>

Example

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

Errors

- Common API Errors

Table 3-15 `xaf_comp_get_status` API

<table>
<tr><td>API</td><td><pre>XAF_ERR_CODE xaf_comp_get_status(pVOID p_adev,
                                 pVOID p_comp,
                                 xaf_comp_status *p_status,
                                 pVOID p_info)</pre></td></tr>
<tr><td>Description</td><td>This API returns the status of the audio component and associated information. <code>p_adev</code> and <code>p_comp</code> should point to the valid audio device and audio component structures, respectively. This API will return one of following status and associated information.
Note: This API is a blocking API; that is, it may block for status from the DSP Interface Layer for a previously issued process command.</td></tr>
<tr><td>Actual Parameters</td><td><code>p_adev</code>
Pointer to the audio device structure

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

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

<code>p_info</code>
Pointer to array of size two WORD32 data types (pointer, size) to get information from the audio component associated with its status. When the <code>p_status</code> returned is <code>XAF_STARTING</code> or <code>XAF_INIT_DONE</code>, this buffer is not updated.</td></tr>
<tr><td>Restrictions</td><td>Should not be called before <code>xaf_comp_create</code> 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 3-16 `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`. That is, if input port is paused, input data consumption is paused on that port, and if output port is paused, output data production is 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 that 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
- Non-fatal error from component.

Table 3-17 `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`. That is, if input port is resumed, input data consumption is resumed on that port, and if output port is resumed, output data production is 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. Being hardware specific, Capturer or Renderer Class do not support `xaf_resume` API. The resume 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 resumed

Restrictions	Should not be called before `xaf_comp_create` API

Example

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

Errors

- Common API Errors
- Non-fatal error from component.

Table 3-18 `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 that the 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. Being hardware specific, Capturer or Renderer Class do not support `xaf_probe_start` API.
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 port 0 and port 1
xaf_comp_set_config(p_audioComp, 1, param);
ret = xaf_probe_start (p_audioComp);
```

Errors

- Common API Errors

Table 3-19 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 that if the 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. Being hardware specific, Capturer or Renderer Class do not support `xaf_probe_stop` API.
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 3-20 `xaf_create_event_channel` API

API	`XAF_ERR_CODE xaf_create_event_channel(pVOID p_src,UWORD32 src_config_param, pVOID p_dest, UWORD32 dst_config_param, UWORD32 nbuf, UWORD32 buf_size)`
Description	This API creates an event communication channel. Note, event communication channel can be created either between two components or between a component and the application.

Actual Parameters	`p_src`
	Pointer to the source audio component
	
	`src_config_param`
	Configuration parameter ID of the source component
	
	`p_dest`
	Pointer to the destination audio component. NULL indicates the event is for the application.
	
	`dst_config_param`
	Configuration parameter ID of the destination(sink) component. NULL indicates the event is for the application.
	
	`n_buf`
	Number of message buffers between the components (per channel) to deliver event and receive response.
	Valid values: 1 to 16
	
	`buf_size`
	Size of each data buffer in the channel. Application programmer must ensure to provide right buffer size with regard to `src_config_param` configuration parameter

Restrictions	Should not be called before `xaf_comp_create` API

Example

1. Channel between two components

```
ret = xaf_create_event_channel(p_audioComp1,
                               XA_COMP1_CONFIG_PARAM,
                               p_audioComp2,
                               XA_COMP2_CONFIG_PARAM,
                               4, 4);
```

2. Channel between a component and application

```
ret = xaf_create_event_channel(p_audioComp1,
                               XA_COMP1_CONFIG_PARAM,
                               NULL, NULL,
                               4, 4);
```

Errors

- Common API Errors

Table 3-21 `xaf_delete_event_channel` API

API	`XAF_ERR_CODE xaf_delete_event_channel(pVOID p_src,UWORD32 src_config_param, pVOID p_dest, UWORD32 dst_config_param)`
Description	This API deletes an event communication channel.

Actual Parameters	`p_src`
	Pointer to the source audio component
	
	`src_config_param`
	Configuration parameter ID of the source component
	
	`p_dest`
	Pointer to the destination audio component
	
	`dst_config_param`
	Configuration parameter ID of the destination(sink) component

Restrictions	Should not be called before channel (which is to be deleted) is created using `xaf_create_event_channel`. Should not be called before `xaf_comp_create` API Note, If component deletion is attempted before calling this API, then the associated event channels would be deleted automatically.

Example

```
ret = xaf_delete_event_channel(p_audioComp1,
                               XA_COMP1_CONFIG_PARAM,
                               p_audioComp2,
                               XA_COMP2_CONFIG_PARAM);
```

Errors

- Common API Errors

Table 3-22 `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, UWORD32 core)`
Description	This API enables preemptive scheduling of audio components on the DSP Interface Layer. By default, DSP Interface Layer 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 the `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 that the 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`
	background priority level
	
	`core`
	core ID on which DSP worker threads are to be created if NCORES>1.

Restrictions	Should not be called before `xaf_adev_open` API. Should be called only once after `xaf_adev_open` API. Priority of DSP worker threads should not exceed the priority of DSP thread. That is, (`rt_priority_base` + `n_rt_priorities -1`) should be less than or equal to DSP thread priority. `rt_priority_base` should be at-most DSP-thread priority. `bg_priority` should be at-most DSP-thread priority.

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
 * on a core with core ID 1.
 */
ret = xaf_adev_set_priorities(p_adev, 2, 3, 1, 1);
```

Errors

- Common API errors

Table 3-23 `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 3-24 `xaf_get_mem_stats` API

<table>
<tr><td>API</td><td><pre>XAF_ERR_CODE xaf_get_mem_stats(pVOID p_adev,
                               UWORD32 core,
                               WORD32 *p_mem_stats)</pre></td></tr>
<tr><td>Description</td><td>This API returns the information about the memory usage statistics of the audio components, framework and XAF. <code>p_adev</code> should point to the valid audio device structure. This API will update the pointer contents with memory usage statistics.</td></tr>
<tr><td>Actual Parameters</td><td><code>p_adev</code>
Pointer to the audio device structure

<code>core</code>
core ID number

<code>p_mem_stats</code>
Pointer to an array of five WORD32 data types to get information from the API about the memory usage statistics in bytes.

<table>
<tr><td>Array values</td><td>Description</td></tr>
<tr><td><code>p_mem_stats[0]</code></td><td>Peak usage of local Memory by Audio Components</td></tr>
<tr><td><code>p_mem_stats[1]</code></td><td>Peak usage of shared Memory by Audio Components and Framework</td></tr>
<tr><td><code>p_mem_stats[2]</code></td><td>Local Memory used by Framework structures</td></tr>
<tr><td><code>p_mem_stats[3]</code></td><td>Peak usage of shared Memory by Audio Components if <code>NCORES>1</code>
Current usage of local memory by Audio Components if <code>NCORES=1</code></td></tr>
<tr><td><code>p_mem_stats[4]</code></td><td>Current usage of shared memory by Audio Components and Framework
if <code>NCORES=1</code></td></tr>
</table></td></tr>
<tr><td>Restrictions</td><td>The API is recommended to be used at the very end of application execution and before closing the device (using <code>xaf_adev_close</code> API call) for the memory statistics to be reliable.
Can be called from Master DSP only.</td></tr>
</table>

Example

```
WORD32 mem_stats[5];
ret = xaf_get_mem_stats(p_adev, 0,
                        &mem_stats[0]);
```

Errors

- Common API Errors

Table 3-25 `xaf_dsp_open` API

API	`XAF_ERR_CODE xaf_dsp_open(pVOID *pp_adev, xaf_adev_config_t *pconfig)`
Description	This API opens and initializes the audio device structure on worker cores. It starts the DSP thread that performs all audio processing on DSP Interface Layer on worker core. It also allocates local memory to be used by the audio components.. It passes dsp_shared_memory pointer which houses the common shared structure across DSPs.

Actual Parameters	`pp_adev`
	Address of pointer to audio device. This API call allocates memory for audio device and updates this pointer with it.
	
	`pconfig`
	Pointer to an initialized structure that contains the necessary parameters for this API. Refer to Table 3-2 `xaf_adev_open` API for description of `xaf_adev_config_t` structure variables.

Restrictions	■ **Prerequisite:** The RTOS startup procedure must be invoked before calling this function. Procedures for XOS and FreeRTOS are as follows. ■ 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. This API should not be called from Master core testbench.

Example

```
ret = xaf_dsp_open(&p_adev,
                    &adev_config);
```

Errors

- Common API Errors

Table 3-26 `xaf_dsp_close` API

API	`XAF_ERR_CODE xaf_dsp_close(pVOID p_adev)`
Description	This API waits on the worker core for DSP-thread to finish. It populates `cb_stats` structure with memory stats and also calls cb_compute_cycles which updates the execution cycles. It frees the memory allocated during `xaf_dsp_open`.
Actual Parameters	`p_adev` Address of pointer to audio device.
Restrictions	■ Should not be called before xaf_dsp_open API. ■ This API should not be called from Master core testbench.

Example

```
ret = xaf_dsp_close(p_adev);
```

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 3-8.

Table 3-27 `XAF_COMP_CONFIG_PARAM_PROBE_ENABLE` Configuration 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 configuration parameter is only supported during audio component initialization (as it results in one-time probe buffer allocation during initialization); that is, probe specification cannot be changed at runtime. For an input port with input-bypass mode active, the `xaf_comp_set_config` API with this parameter will return `XAF_INVALIDVAL_ERR` fatal error.

Table 3-28 XAF_COMP_CONFIG_PARAM_RELAX_SCHED Configuration Parameter

Configuration Parameter	XAF_COMP_CONFIG_PARAM_RELAX_SCHED
Description	By default, each processing or execution call of MIMO Class component requires that all the necessary ports are ready; that is, at least one of the active input ports has 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 the responsibility of respective component plugin implementation to manage execution without readiness of specified ports.
Values	Value Type: UWORD32 Default Value: 0 (All ports disabled) Example value: 0x3 (port 0 and port 1 readiness checks are relaxed)
Restrictions	This configuration parameter is only supported for MIMO Class components and it can be used at component initialization as well as at runtime.

Table 3-29 XAF_COMP_CONFIG_PARAM_PRIORITY Configuration Parameter

Configuration Parameter	XAF_COMP_CONFIG_PARAM_PRIORITY
Description	By default, DSP Interface Layer 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 relative priority of audio component w.r.t base_priority. It accepts values from 0 to (max(UWORD32)-1). Note, higher number indicates higher priority and vice versa. A value higher than the highest possible priority, which is determined from set_priority API parameters, results in fatal error.
Values	Value Type: UWORD32 Example value: 0x3 (audio component runs at priority base priority + 3)
Restrictions	This configuration parameter 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, otherwise this parameter would be ignored.

Table 3-30 XAF_COMP_CONFIG_PARAM_DEC_INIT_WO_INP Configuration Parameter

Configuration Parameter	XAF_COMP_CONFIG_PARAM_DEC_INIT_WO_INP
Description	Generally, decoders which use speech APIs do not require input data for initialization but those which use audio APIs require input data for initialization. By setting this configuration parameter, a decoder can attempt initialization without input data. If initialization without input data succeeds, XAF_INIT_DONE status is returned to application. If initialization without input fails, then XAF_INIT_NEED_INPUT status is returned after which the application can re-attempt initialization by providing input data. Note, since the output buffer would have returned to the application after the first initialization attempt, the same needs to be sent back again using XAF_START_FLAG.

Values	<table><tr><td>Value Type</td><td>UWORD32</td></tr><tr><td>Example value</td><td>1 (To allow attempt initialization without providing input data)</td></tr></table>
Restrictions	This configuration parameter is supported for 'XAF_DECODER' type components. For other components 'XAF_INVALIDVAL_ERR' error is returned.

4. Xtensa Audio Framework Package

The XAF package is released in the 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 following diagrams.

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

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

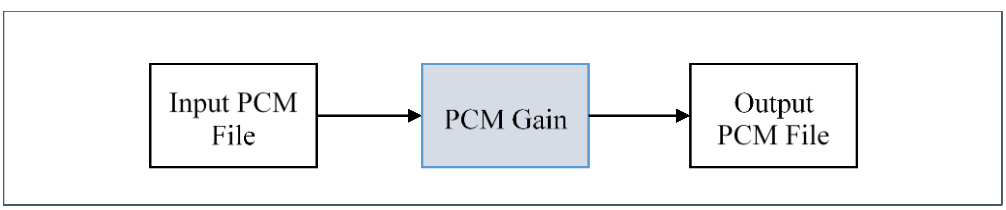


Figure 4-1 Testbench 1 (pcm-gain) Block Diagram

Testbench 2 (`xa_af_dec_test`) decodes MP3 streams.

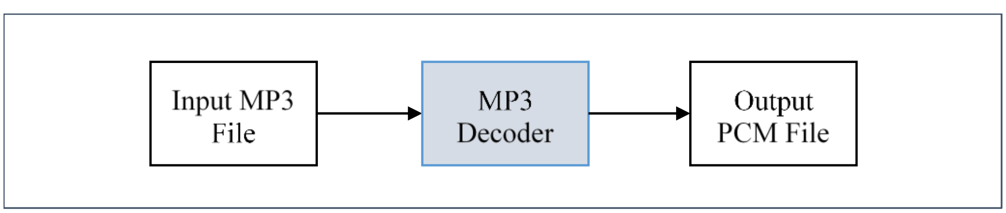


Figure 4-2 Testbench 2 (mp3-dec) Block Diagram

Testbench 3 (`xa_af_dec_mix_test`) decodes two MP3 streams and mixes the output. The mixer used in this testbench is a MIXER class component with 4 input ports and 1 output port.

Note Mixer component used in this testbench allows start of processing (schedule for execution) when at least one of the input ports is connected and valid input is available (among the 4 input ports). The connections and data arrival instances on input ports can vary between single core and multicore execution, which means the output of the mixer can differ.

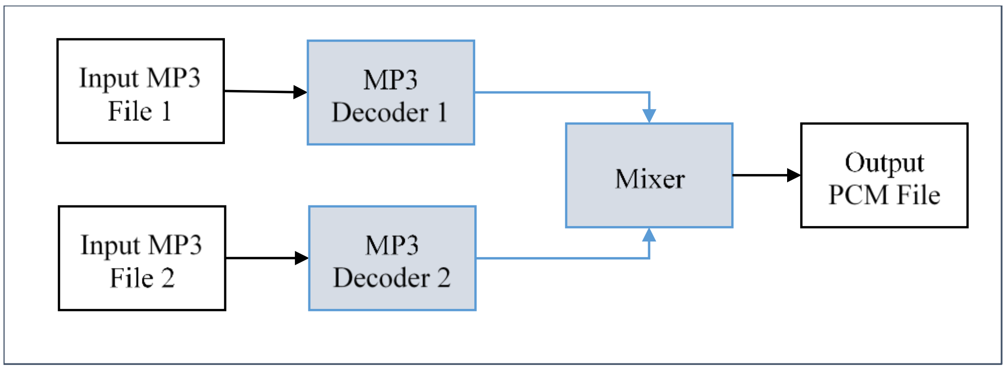


Figure 4-3 Testbench 3 (dec-mix) Block Diagram

Testbench 4 (`xa_af_full_duplex_opus_test`) encodes an OPUS stream and simultaneously decodes an OPUS stream. The Opus decoder supports both OGG and RAW encoded input data. OGG and RAW mode can be altered by enabling/disabling `#define ENABLE_RAW_OPUS_SET_CONFIG` in the testbench.

This testbench demonstrates usage of extended set config (`xaf_set_config_ext`) and get config (`xaf_get_config_ext`) APIs.

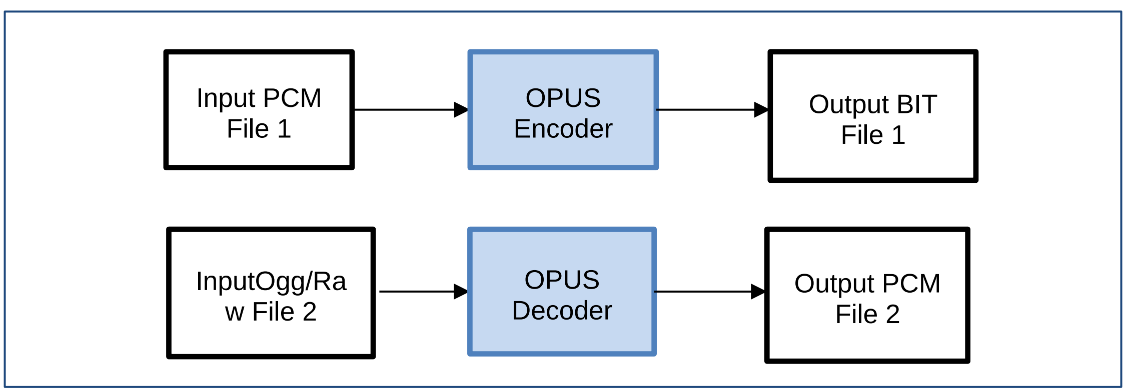


Figure 4-4 Testbench 4 (full-duplex-opus) Block Diagram

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

This testbench demonstrates decoder initialization without input functionality by using the configuration parameter `XAF_COMP_CONFIG_PARAM_DEC_INIT_WO_INP`.

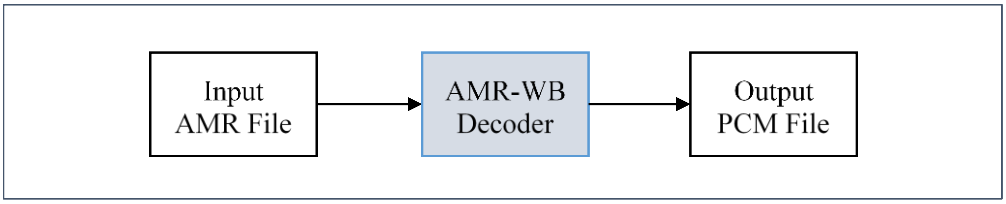


Figure 4-5 Testbench 5 (amr-wb-dec) Block Diagram

Testbench 6 (`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.

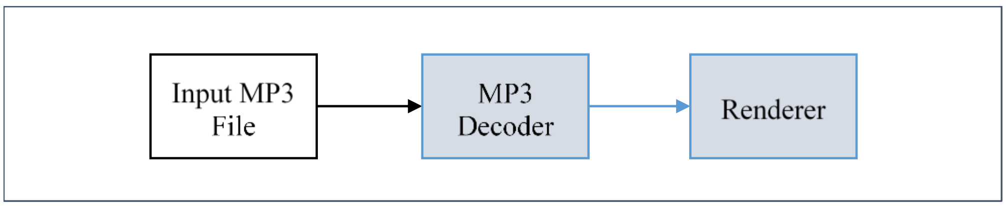


Figure 4-6 Testbench 6 (mp3-dec-renderer) Block Diagram

Testbench 7 (`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.

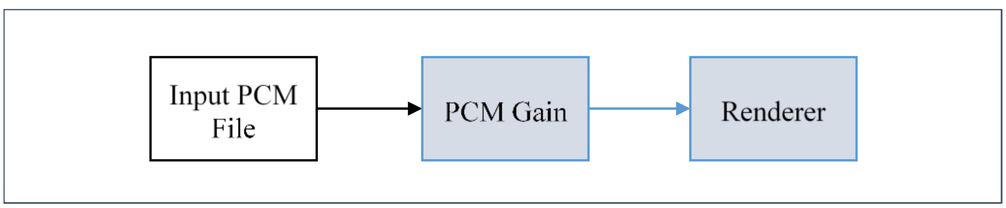


Figure 4-7 Testbench 7 (pcm-gain-renderer) Block Diagram

Testbench 8 (`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.

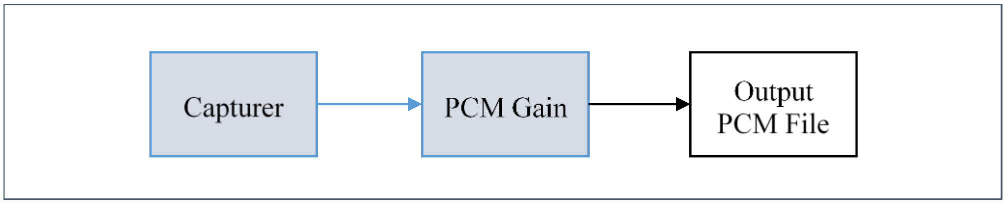


Figure 4-8 Testbench 8 (capturer-pcm-gain) Block Diagram

Testbench 9 (`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.

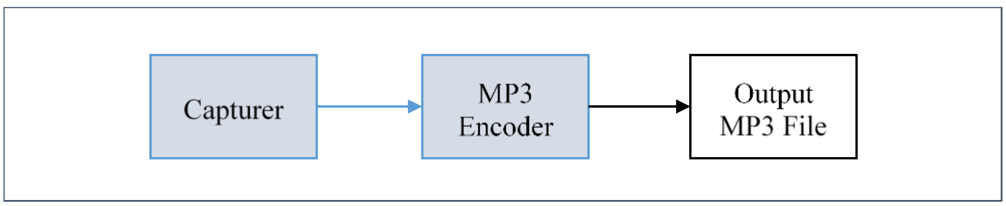


Figure 4-9 Testbench 9 (capturer-mp3-enc) Block Diagram

Testbench 10 (`xa_af_mimo_mix_test`) applies gain to two PCM streams and mixes them to produce the output. For this testbench, the mixer is a MIMO class component with 2 input ports and 1 output port.

Note that 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.

This testbench also demonstrates event communication functionality using the `xaf_create_event_channel` and `xaf_delete_event_channel` APIs. Here, the MIMO Mixer component communicates with PCM Gain components to change their gain factor after producing certain amount of data. Here the orange arrows represent event communication channel.

Note MIMO-Mixer component used in this testbench has two input-ports. The component waits for inputs to be available on both the ports before consuming.

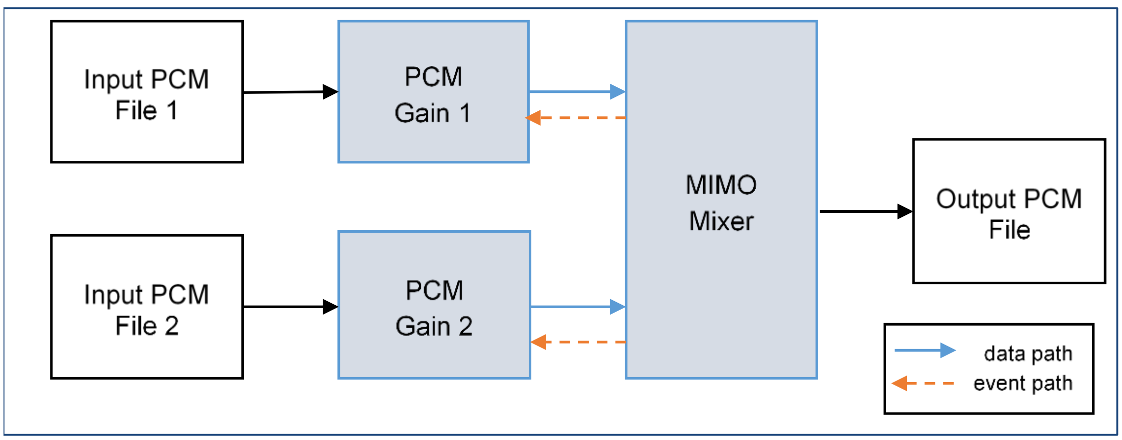


Figure 4-10 Testbench 10 (mimo-mix) Block Diagram

Testbench 11 (`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. The mixer in this testbench is a mixer class component with 4 input ports and 1 output port.

Note that 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.

This testbench also demonstrates propagation and handling of component execution errors to the application. This is enabled using the component configuration parameter `error_channel_ctl` during component creation, which creates an error channel between framework and the application. The errors received, if any, are handled gracefully in the testbench. Refer to the error handler example implementation in the testbench code for more details.

Note	Mixer component used in this testbench allows start of processing (schedule for execution) when at least one of the input ports is connected and valid input is available (among the 4 input ports). The connections and data arrival instances on input ports can vary between single core and multicore execution, which means the output of the mixer can differ, and so will the final outputs.

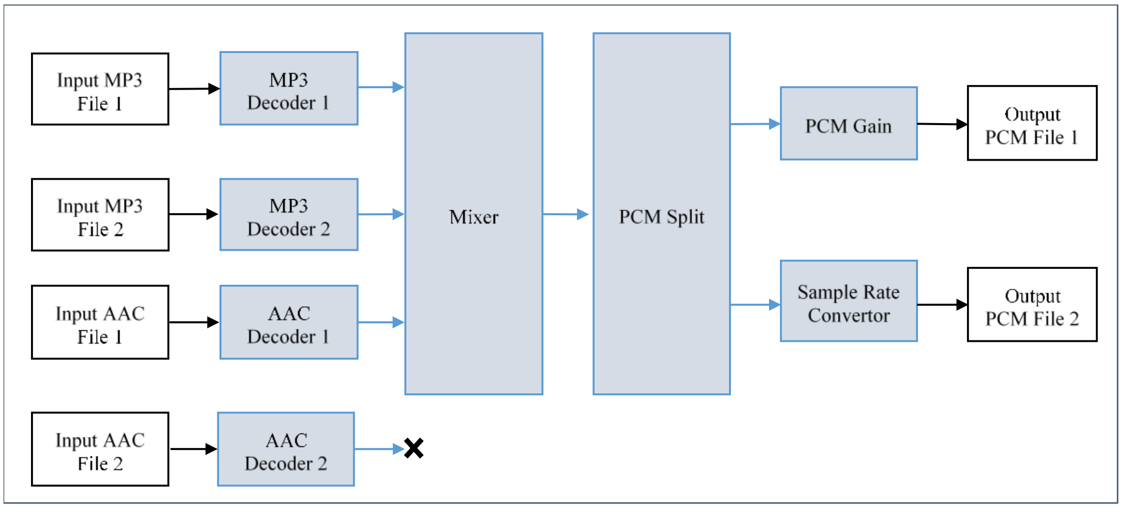


Figure 4-11 Testbench 11 (playback-usecase) Block Diagram

Testbench 12 (`xa_af_renderer_ref_port_test`) demonstrates use of renderer optional port as feedback or reference path for echo cancellation type of applications. It demonstrates the connection between two independent audio-processing chains. One chain is PCM-Gain1, RENDERER, the other being PCM-Gain2, AEC23, PCM-Gain3 and PCM-Gain 4.

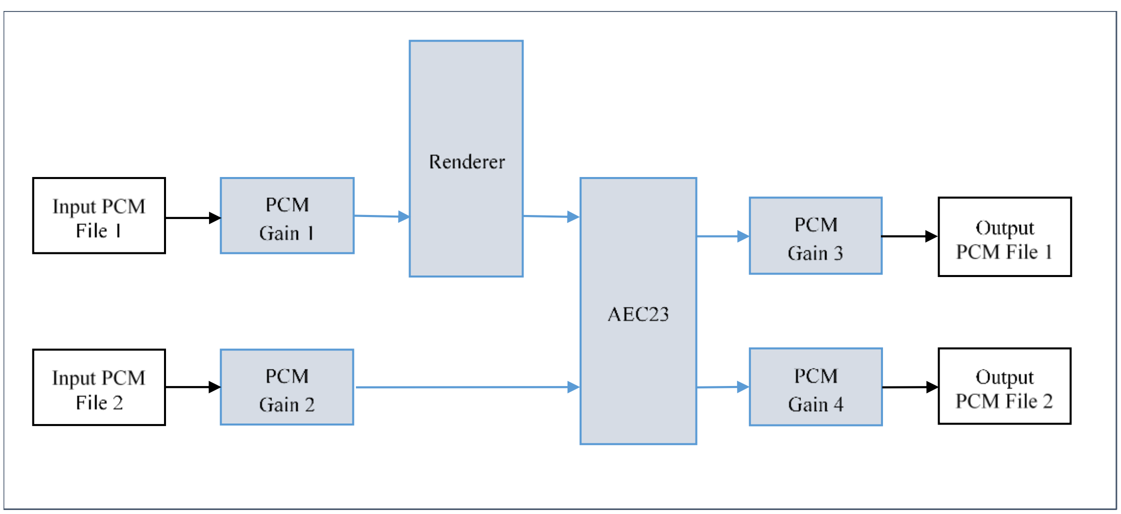


Figure 4-12 Testbench 12 (renderer-ref-port) Block Diagram

Testbench 13 (`xa_af_tflite_microspeech_test`) captures a PCM stream from the audio input device (in case of a hardware platform) and detects Yes/No keyword and outputs the corresponding Yes/No score in the cases where Yes/No keyword is recognized. For the simulator case, the input is read from a file.

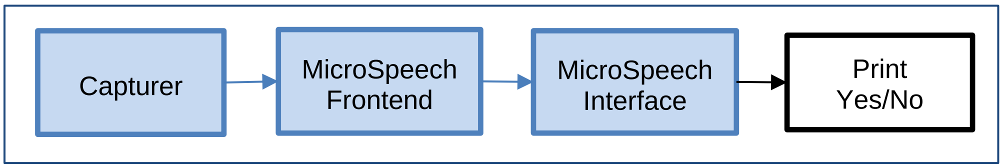


Figure 4-13 Testbench 13 (capturer-tflite-microspeech) Block Diagram

Testbench 14 (`xa_af_tflite_person_detect_test`) detects the presence or absence of a person as person/no person for the given input data. It prints the person/no person inference score.

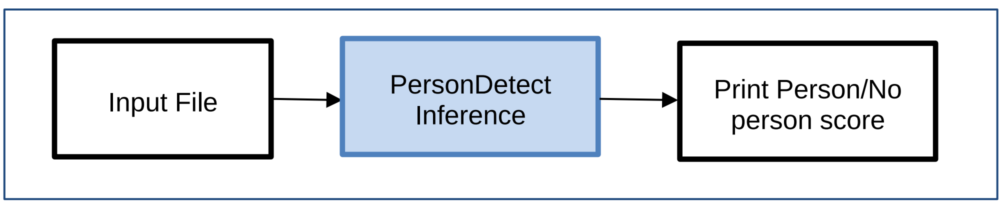


Figure 4-14 Testbench 14 (tflite-person-detect) Block Diagram

Testbench 15 (`xa_af_person_detect_microspeech_test`) detects and prints Yes/No keyword for the capturer input and simultaneously detects person/no person and provides inference score for the given input.

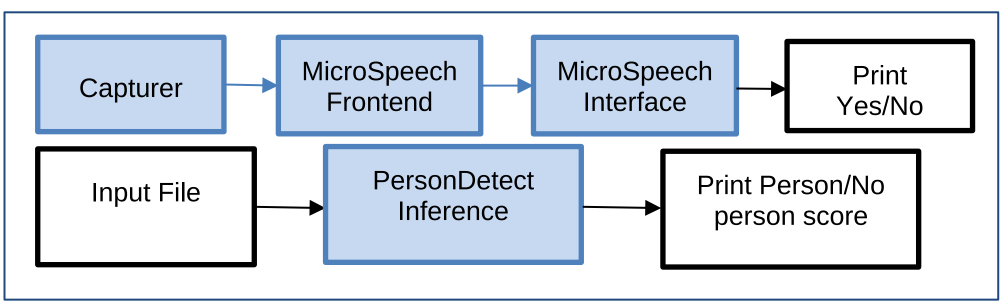


Figure 4-15 Testbench 15 (person-detect-microspeech) Block Diagram

Table 4-1 summarizes component header file, component wrapper file, and component library dependencies for each of fifteen testbenches included in XAF package. The testbench sources use a set of preprocessor symbols (refer to section 4.4.1) to enable inclusion of respective component plugins into compilation.

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-opus-test.c	xa-opus-decoder.c xa-opus-encoder.c	xa_opus_codec_api.h xa_opus_encoder_api.h xa_opus_decoder_api.h xa_ogg_lib_api.h opus_header.h	xa_opus_codec.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-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
7	xaf-gain-renderer-test.c	xa-pcm-gain.c xa-renderer.c	xa-pcm-gain-api.h xa-renderer-api.h	-
8	xaf-capturer-pcm-gain-test.c	xa-capturer.c xa-pcm-gain.c	xa-capturer-api.h xa-pcm-gain-api.h	-
9	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
10	xaf-mimo-mix-test.c	xa-pcm-gain.c xa-mimo-mix.c	xa-pcm-gain-api.h xa-mimo-mix-api.h	-
11	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_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

No	Testbench source file	Component wrapper files	Component header files	Component libraries
		xa-src-pp.c		
12	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	-
13	xaf-capturer-tflite-microspeech-test.c	xa-capturer.c tflm-inference-api.cpp xa-tflm-inference-api.c xa-microspeech-frontend.c microspeech_model_data.c microspeech-frontend-wrapper-api.cpp microspeech-inference-wrapper-api.cpp	xa-capturer-api.h xa-microspeech-frontend-api.h xa-microspeech-inference-api.h microspeech_model_data.h tflm-inference-api.h	libtensorflow-microlite.a libmicro_speech_frontend.a
14	xaf-tflite-person-detect-test.c	tflm-inference-api.cpp xa-tflm-inference-api.c person_detect_model_data.c person-detect-wrapper-api.cpp	person_detect_model_data.h xa-person-detect-inference-api.h tflm-inference-api.h	libtensorflow-microlite.a
15	xaf-person-detect-microspeech-test.c	xa-capturer.c tflm-inference-api.cpp xa-tflm-inference-api.c xa-microspeech-frontend.c microspeech_model_data.c microspeech-frontend-wrapper-api.cpp microspeech-inference-wrapper-api.cpp person_detect_model_data.c person-detect-wrapper-api.cpp	xa-capturer-api.h xa-microspeech-frontend-api.h xa-microspeech-inference-api.h microspeech_model_data.h tflm-inference-api.h person_detect_model_data.h xa-person-detect-inference-api.h	libtensorflow-microlite.a libmicro_speech_frontend.a

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-opus-test.c
- xaf- amr-wb-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-mimo-mix-test.c
- xaf-playback-usecase-test.c
- xaf-renderer-ref-port-test.c
- xaf-capturer-tflite-microspeech-test.c
- xaf-tflite-person-detect-test.c
- xaf- person-detect-microspeech-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 Interface Layer source and include files.
- `host-apf/` – App Interface Layer 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.
- `sysdeps/mc_ipc` – Multicore-XAF IPC lock, interrupt and reset-sync definition header files.
- `xaf-api.h` – XAF Developer APIs header file.
- `xf-debug.h` – XAF debug trace support header file.

XAF shared memory directory (`/xf_shared/`)

- `src/xf-shared.c` – IPC shared memory buffer definitions (for NCORES>1).
- `include/xf-shared.h` – IPC shared memory buffer macro definition and references.

XAF system file directories (`/xtsc/`)

- `xaf_xtsc_sys_2c.xtsys, xaf_xtsc_sys_2c.yml` – System specification files for 2 core system (NCORES=2)
- `xaf_xtsc_sys_3c.xtsys, xaf_xtsc_sys_3c.yml` – System specification files for 3 core system (NCORES=3)
- `xaf_xtsc_sys_4c.xtsys, xaf_xtsc_sys_4c.yml` – System specification files for 4 core system (NCORES=4)

4.3 Build and Execute using tgz Package

4.3.1 Making the Executable

Unpack the source tgz package which generates “xa_af_hostless”. Call it <BASE_DIR>.

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

To build XAF Library and testbenches with FreeRTOS as RTOS:

1. Follow the 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 must be built before building the testbench application. To build the XAF library, follow these steps:

For NCORES=1:

1. Go to <BASE_DIR>/`build/`
2. At the prompt, enter

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

For NCORES>1: (Example commands for NCORES=2)

1. Go to <BASE_DIR>/`build/`
2. At the prompt, enter

```
$ xt-make sysbuild NCORES=2
```

Note: At this point the environment variable XTENSA_SYSTEM should be set to the following:

```
<Absolute path of the BASE_DIR>/xtsc/mbuild/package/config
```

3. At the prompt, enter

```
$ xt-make clean all install NCORES=2 [XA_RTOS=freertos FREERTOS_BASE=<dir>]
```

This 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 <BASE_DIR>/test/build.
2. At the prompt, enter:

```
$ xt-make -f makefile_testbench_sample clean af_hostless [NCORES=<num cores> XA_RTOS=<freertos> FREERTOS_BASE=<dir>]
```

Note: NCORES parameter is optional for NCORES=1.

This will build the pcm-gain example test application.

All Testbenches:

To build the other testbenches, the Cadence MP3 decoder [4], MP3 encoder [5], AMR-WB decoder [6] [7], Sample rate converter [8], AAC decoder [9], 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/opus_enc/lib/xa_opus_codec.a
/test/plugins/cadence/opus_dec/lib/xa_opus_codec.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_opus_codec_api.h
```

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

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

This will build all the testbench applications except TFLM testbench applications, which can be generated independently with the following command:

```
$ xt-make -f makefile_testbench_sample clean <target> [NCORES=<num cores>
XA_RTOS=freertos FREERTOS_BASE=<rtos_dir> TFLM_BASE=<dir>/tensorflow]
```

Where, the `target` can be one of the following: `af_tflm_microspeech`, `af_tflm_pd`, `af_tflm_microspeech_pd`.

Note: TFLM libraries must be built before building the TFLM testbenches. Refer to section 4.6 for steps to build the TFLM library and for additional build settings for TFLM test examples refer `libxa_af_hostless/build/readme_tflm.txt`.

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 "`XF_TRACE=<TRACE_LEVEL>`" to both XAF library and testbench compilation command lines described above. For all trace prints, set `TRACE_LEVEL` as 1. For trace prints related to command, response transactions, set `TRACE_LEVEL` as 2. Any trace tag can be enabled or disabled by setting or resetting `TRACE_TAG` listed in include/xf-debug.h. For example,

  `TRACE_TAG(PROCESS, 1);` /*… PROCESS trace tag is enabled */

  `TRACE_TAG(PROCESS, 0);` /*… PROCESS trace tag is disabled */

  Note: With more trace tags enabled, size of the executable is increased. A "CRITICAL" trace tag is provided to print only minimal and important trace logs of errors/warnings during component execution and configuration without significant increase in the executable size. For only critical error/warnings trace prints, disable all other tags except the CRITICAL tag.

- To build without event communication support, add "XA_DISABLE_EVENT=1" to both XAF library and testbench compilation command lines described above which can save the corresponding code memory. Note, event communication support is enabled by default.

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) for NCORES=1 and using cycle accurate Xtensa System C (XTSC) simulator for NCORES>1. 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 that 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_af_hostless
[NCORES=<num cores>]
```

All Testbenches:

To run all the testbenches (except the TFLM testbenches), at the prompt (in `test/build)`, use the following command. The TFLM testbenches can be run independently.

```
$ xt-make –f makefile_testbench_sample run-dec [NCORES=<num
cores>]
```

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.
Note	NCORES parameter is optional for NCORES=1.

Individual testcase:

The script `xaf_xtsc_run.sh` accepts the identical command line to that of Hostless XAF and renders it to the multicore simulator xtsc-run.

```
./xaf_xtsc_run.sh NCORES [1,2(default),3,4,..,N] xt-run
xa_af_playback_usecase_test -
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3 -
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3 -
infile:../test_inp/hihat_1ch_16b_44.1kHz.adts -
infile:../test_inp/hihat_1ch_16b_44.1kHz.adts -outfile:out0.pcm -
outfile:out1.pcm -core-cfg:1,1 -core-cfg:2,2 -core-cfg:3,4
```

1. The binary names provided should be without the name `_coreX` (the binaries get generated with <test-name>`_core0, _core1` -etc.)
   ```
   Example: if the testcase binaries generated are
   xa_af_playback_usecase_test_core0,
   xa_af_playback_usecase_test_core1 then provide
   './xaf_xtsc_run.sh xt-run xa_af_playback_usecase_test -
   infile:<> -infile:<> -outfile:<>
   ```

2. One can also use the command directly without the script `xaf_xtsc_run.sh` with appropriate comma separated arguments
   ```
   Example: xtsc-run --
   define=core0_BINARY=xa_af_playback_usecase_test_core0 --
   ```

```
define=core0_BINARY_ARGS=-
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3,-
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3,-
infile:../test_inp/hihat_1ch_16b_44.1kHz.adts,-
outfile:out0.pcm,-outfile:out1.pcm,-core-cfg:1,1 --
define=core1_BINARY=xa_af_playback_usecase_test_core1 --
define=XTSC_LOG=0 --include=../../xtsc/sysbuilder/xtsc-
run/multicore2c.inc
```

4.3.3 Component Creation on a Worker-Core

By default, all the components are created on core-0 which is the master core with XF_CORE_ID=0.

To create a component on a different core `-core-cfg:<core>, <component id or comp_id>` command-line option should be used.

For example:

```
./xaf_xtsc_run.sh NCORES 4 xt-run xa_af_playback_usecase_test -
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3 -
infile:../test_inp/hihat_1ch_16b_192kbps_cbr.mp3 -
infile:../test_inp/hihat_1ch_16b_44.1kHz.adts -
infile:../test_inp/hihat_1ch_16b_44.1kHz.adts -outfile:out0.pcm -
outfile:out1.pcm -core-cfg:1,1 -core-cfg:2,2 -core-cfg:3,4
```

In the above example, there are seven components with id ranging from 0 to 6.

`-core-cfg:1,1` component 1 is created on worker core 1

`-core-cfg:2,2` component 2 is created on worker core 2

`-core-cfg:3,4` component 4 is created on worker core 3

Rest of the components 0,3,5 and 6 are created on core 0 (master DSP)

Note In System C Simulator (XTSC) 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.

Note The xt-run command token is parsed by the shell script **xaf_xtsc_run.sh** in the test/build folder into comma separated tokens as required by the SystemC simulator **xtsc-run** for the execution of the testcase.

Note The command argument parsing of "`-core-cfg:<core-id X>,<<comp-id I>,<comp-id J>,<comp-id K>>`" is supported by the projects `testxa_af_gain_renderer_test`, `testxa_af_playback_usecase_test`, `testxa_af_mimo_mix_test`, `testxa_af_renderer_ref_port_test`. Other project testbenches can be updated in a similar way as required.

4.4 Build and Execute using xws Package

4.4.1 Working with XAF xws Package

The XAF xws package can be used in both single core(NCORES=1) and multicore(NCORES>1) subsystems. The xws contains the XAF library project "libxa_af_hostless" and 15 testbench projects as shown in Table-4.2.

Table 4-2 XWS Test Project List

S No	Test project	Testbench
1	testxa_af_hostless	`xa_af_hostless_test`
2	testxa_af_mimo_mix	`xa_af_mimo_mix_test`
3	testxa_af_gain_renderer	`xa_af_gain_rend_test`
4	testxa_af_capturer_gain	`xa_af_capturer_pcm_gain_test`
5	testxa_af_renderer_ref_port	`xa_af_renderer_ref_port_test`
6	testxa_af_dec*	`xa_af_dec_test`
7	testxa_af_dec_mix*	`xa_af_dec_mix_test`
8	testxa_af_amr_wb_dec*	`xa_af_amr_wb_dec_test`
9	testxa_af_mp3_dec_rend*	`xa_af_mp3_dec_rend_test`
10	testxa_af_capturer_mp3_enc*	`xa_af_capturer_mp3_enc_test`
11	testxa_af_playback_usecase*	`xa_af_playback_usecase_test`
12	testxa_af_full_duplex_opus*	`xa_af_full_duplex_opus_test`
13	testxa_af_tflm_microspeech*	`xa_af_tflite_microspeech_test`
14	testxa_af_tflm_pd*	`xa_af_tflite_person_detect_test`
15	testxa_af_tflm_microspeech_pd*	`xa_af_person_detect_microspeech_test`

(* These test projects have library dependencies, hence will not build and run out-of-the-box. Refer to step 5 to build these test-projects.)

Note	The above testbenches require Xtensa Xplorer version 8.0.16 or later.

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. Xtensa Xplorer supports two build modes “Release” and “Debug”, which can be selected with “Target (T:)”. “Release” mode uses default build options whereas “Debug” mode uses build options defined under “DEBUG=1” in the Makefiles.

Single core XAF (NCORES=1):

1. To import the HiFi Audio Framework Xtensa Workspace file (extension xws) into Xplorer, click **File → Import…**. The Import wizard opens. Select **Import Xtensa Xplorer Workspace**. Click **Next >**. Browse for the Xtensa workspace file and click **Next >**

   select project ‘`libxa_af_hostless`’

   select project ‘`testxa_af_hostless`’

   select any other or all projects among the 15 available test-projects

   click **Finish** (Ignore the warning: “There are unimported items..”)
2. Select a test project from 2 to 14 from Table 4-2 XWS Test Project List as the active project. For example, select “testxa_af_hostless” (PCM-gain) as the active project and any of the compatible HiFi cores as the configuration.
3. Build by clicking the **Build Active** button.
4. To run the selected Testbench (example: `testxa_af_hostless_test` i.e. PCM gain), from the “Run configurations” menu, select the launch corresponding to the active project available under “Xtensa Single Core Launch” and click the **Run** button.

   Note: One should choose the cycle-accurate simulation launch `<test project>_cycle` (refer to Known Issues) to run the test.

   The default input or output file settings can be changed, from the “Run configurations” menu under “Arguments” tab in “Program Arguments” text box by modifying the command text.

   For example, in `testxa_af_hostless` modify the following as required:

   `-infile:<input PCM file> -outfile:<output PCM file>`
5. To build and run other testbenches with library dependencies, 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. 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`).
   c. Follow steps 2 to 4 as given above, with appropriate command-line arguments.

d. For any custom testbenches other than those mentioned in Table 4-2 XWS Test Project List, ensure that the required symbols among the following are defined in “Build Properties” under “Symbols” tab.

```
XA_PCM_GAIN=1
XA_MP3_DECODER=1
XA_MP3_ENCODER=1
XA_SRC_PP_FX=1
XA_AAC_DECODER=1
XA_MIXER=1
XA_AMR_WB_DEC=1
XA_RENDERER=1
XA_CAPTURER=1
XA_AEC22=1
XA_AEC23=1
XA_PCM_SPLIT=1
XA_MIMO_MIX=1
XA_OPUS_ENCODER=1
XA_OPUS_DECODER=1
XA_TFLM_MICROSPEECH=1
XA_TFLM_PERSON_DETECT=1
```

These symbols enable inclusion of respective component plugins into compilation. While most of the symbol names are self-explanatory, following is a brief list of some of these symbols and their respective component plugin.

XA_MIXER	PCM mixer, 4 in 1 out
XA_SRC_PP_FX	Sample rate converter
XA_AEC22	Dummy acoustic echo canceler, 2 in 2 out MIMO component
XA_AEC23	Dummy acoustic echo canceler, 2 in 3 out MIMO component
XA_PCM_SPLIT	PCM splitter, 1 in 2 out MIMO component
XA_MIMO_MIX	MIMO class mixer component, 2 in 1 out

e. 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 *d* 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 in the testbench.

```
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 = <TRACE_LEVEL>` in the “Symbols” tab for both 'libxa_af_hostless' and 'testxa_af_hostless' projects. Refer to Special Build Settings for details about available TRACE levels.

Notes:

1. The project `testxa_af_hostless` has a common `test_inp` directory that contains the test input files required for all the test projects in the package and a common `test_out` directory containing any output files generated for all the test projects. Hence one must also import this project into the workspace.
2. For `testxa_af_full_duplex_opus`, in step 5.b it is required to provide the path of `xa_opus_codec.a` of either of the opus_enc or opus_dec, but not both.
3. Refer to section 4.6 for steps to build the TFLM library and for additional build settings for TFLM test examples refer `libxa_af_hostless/build/readme_tflm.txt`.

Multicore XAF (NCORES>1):

Note that the previous section about importing and building testbench projects into Xplorer is same for multicore also. Only the additional steps specific to multicore are mentioned here.

1. Select the 'project xws' (ex: xa_hifi_af_hostless_lib_3_1_Beta_api_3_0_src.xws), select project ‘`libxa_af_hostless`’

   select project ‘`testxa_af_hostless`’

   select project ‘`xf_shared`’

   select any other or all projects among the 15 test-projects available

   click **Next**

   Select required project launch configuration (ex: BMap0_af_hostless_2c) for one or more projects selected above

   click **Next**

   select required subsystem from the package multicore2c for NCORES=2, multicore3c for NCORES=3, multicore4c for NCORES=4

   click **Finish**

2. The test projects need to be re-imported with different names, for as many cores in the system (NCORES). For example, if NCORES=2, the testbench project `testxa_af_hostless` needs to be imported twice. Since Xplorer would not allow importing the same project with same name again, it needs to be renamed (say testxa_af_hostless2) before importing.

   Click **File → Import…**. The Import wizard opens. Select **Import Xtensa Xplorer Workspace**. Click **Next >**. Browse for the Xtensa workspace file and click **Next >**. Select one of the test project from the available project checkboxes, on the right side there is option to rename. Rename the project and click “apply”. Repeat the same for all the test projects that need to be re-imported and click **Next-> Finish.**

   Note: The library project `libxa_af_hostless`, launch configurations, subsystems and the xf_shared project should not be re-imported.

3. In the workspace window, select core (Example: AE_HiFi4_LE5_XC_MOD_XTSC)
   Select test project: (Example: `testxa_af_hostless`)
   Select target: Release

4. In the **System overview** window, expand to see **Subsystems**. Right click on the named <`subsystem`> (Example: multicore2c) and choose **build subsystem** which builds the subsystem into `${workspace_loc}/<subsystem>/bin/sysBuild`

5. Expand the named <subsystem> (Example: multicore2c) and right click on **MMap0** and chose **build Memory map** which builds the memory-map and required include header into the mbuild location `${workspace_loc}/<subsystem>/bin/mBuild`

   Note: MMap0 build can result into error like "Binary map "BMap10_<unimported project name> of memory map 'MMAP0' has inaccessible project 'testxa_<unimported project name>'". This is error is due to MMap0 being linked to all the project in the xws package. If all the projects are imported, then this error will not occur.

   To avoid the error: Double click on MMap0.

   In the window "multicore2c"→ Memory Maps → Memory and Binary maps → drop-down arrow of MMap0, select a project with X-mark in red which indicates that project is not imported, and click '**Remove**' button on the right side of the window.

   Do this for all other unimported projects.

   Ctrl+S to save the state.

   Re-attempt building MMap0 as mentioned at the start of this step.

6. Set required include paths (note, these are only additional include paths required for multicore build).

   For `libxa_af_hostless` project:

   a. `${workspace_loc}/<subsystem>/bin/sysBuild/include`
   b. `${workspace_loc}/<subsystem>/bin/mBuild/MMap0/package/xtensa-elf/include`

      where <subsystem> names in the package are among: multicore2c, multicore3c, multicore4c

      Example include path if <subsystem> is multicore2c:

      `${workspace_loc}/multicore2c/bin/sysBuild/include`

      `${workspace_loc}/multicore2c/bin/mBuild/MMap0/package/xtensa-elf/include`
   c. `${workspace_loc:libxa_af_hostless/include/sysdeps/mc_ipc}`

   Include path required for all the projects:

   d. `${workspace_loc:xf_shared/include}`

7. Add/Edit symbols:
   For all projects:
   a. To enable cache set symbol `XF_LOCAL_IPC_NON_COHERENT = 1`
   b. Set the symbol `XF_CFG_CORES_NUM` to appropriate number.

Note: The value of `XF_CFG_CORES_NUM` is number-of-cores/NCORES in the subsystem. (Example: `XF_CFG_CORES_NUM=2` for 2-core subsystem)

Symbols for test projects:

a. For testxa_*, each testxa_ project should have unique XF_CORE_ID. XF_CORE_ID varies from 0 to NCORES-1
   Example: for 2 core subsystem, `XF_CFG_CORES_NUM=2`

```
testxa_af_hostless->build-properties->common->symbols->XF_CORE_ID = 0
testxa_af_hostless2->build-properties->common->symbols->XF_CORE_ID= 1
```

8. Link `xf_shared` (the shared library) project as library dependencies to all testxa_* projects using **Library dependencies** option.

   (Right click on all testxa_* the project->select **Library dependencies** option then, double click on xf_shared which should appear in lower-box)

9. In System Overview window: subsystem->MMap0->BMAP0 (Ex: BMap0_af_hostless), attach binaries to cores under **core/project mappings window** double click,

   Select project → Select correct project from the dropdown list. The association should be unique. For example, it is recommended suggested to associate testxa_af_hostless to core0, testxa_af_hostless2 to core1-etc. Do this for all cores under that BMap0.

   Select Build Target → <Active Set> (inherits the Active Build Target of the Active Project).

   Select LSP → sim.

   Arguments for the Xtensa Program → provide the necessary command argument. The test_inp directory can be referenced with testxa_af_hostless/test/test_inp for a input file, similarly testxa_af_hostless/test/test_out for test_out directory. This is necessary for only the master core or core0 in this package.

   Note, by default the example testbench projects have the necessary mappings in place. But it is suggested to verify that the mappings are correct. Do all of the above Selects, except 'Arguments for the Xtensa Program' for all the other worker cores under this BMapX.

10. Build the test-project: Right click on BMap0 (Ex: BMap0_af_hostless) and chose 'build all projects' to build the corresponding test-project. Do the same for all BMap1, BMap2.. BMapN test projects.

11. To run a test project after build, go to **Run Configurations** > Select **MP Launch** and select one of the launch targets in cycle accurate mode.

    Note, Though the default settings should work, it is suggested to check the following are selected:

    Select MP Simulator Launch Type → 'Managed Subsystem'

    Subsystem Launch Options→ Subsystem → Select the project's BMap in the dropdown

    Working Directory → ${workspace_loc}

Debug Options → sync
Debugger Attach Options → Stop All Cores

Notes:

1. The project `testxa_af_hostless` has a common `test_inp` directory that contains the test input files required for all the test projects in the package and a common `test_out` directory containing any output files generated for all the test projects. Hence one must also import this project into the workspace.
2. ${workspace_loc} directory is parent directory to all the projects, and can be accessed from commandline
3. test_inp: All test inputs are available in ${workspace_loc}/testxa_af_hostless/test/test_inp.
4. test_out: All test outputs are to be written to ${workspace_loc}/testxa_af_hostless/test/test_out.
5. The capturer input file 'capturer_in.pcm' is to be copied to the directory ${workspace_loc}.
6. The renderer output file 'renderer_out.pcm' is generated in the directory ${workspace_loc}.
7. To copy capturer_in.pcm to the set location in xws which is ${workspace_loc}, either use commandline OR change the "Working-Directory" in the 'Run-Config' or launch followed by modifying the input/output file paths of the "Argument for the Xtensa Program" associated with the binary of the BMap.

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 Target” in the new window that opens, and 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 the following include paths for common target.

   (Go to **T:Debug**, select **Modify**, select Target as “Common Target” in the new window that opens, and select 'Include Paths' tab).

   `<BASE_DIR>/FreeRTOS/include`

   `<BASE_DIR>/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 Target" in the new window that opens, and 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 Target" in the new window that opens, and 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 the following include path for common target.

   (Go to **T:Debug**, select **Modify**, select Target as "Common Target" in the new window that opens, and select 'Include Paths' tab).

   `<BASE_DIR>/FreeRTOS/include`

   `<BASE_DIR>/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 Target" in the new window that opens, and 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 Target" in the new window that opens, and 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 that the FreeRTOS compilation is only supported under Linux environment.

1. Copy `libxa_af_hostless/build/getFreeRTOS.sh` from XAF Package to the directory of choice outside XAF Package under Linux environment. This directory is referred to as `<BASE_DIR>` in the 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 downloads and builds 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 can copy `<FreeRTOS>` directory from Linux to Windows for building XAF Library and testbenches. In that case, the destination directory on Windows is your new `<BASE_DIR>`.

4.6 Building TFLM for XAF

This section describes how to build the required version of TFLM [13] library to be used with XAF. Note that this TFLM compilation method is only supported under Linux environment.

1. Copy `libxa_af_hostless/build/getTFLM.sh` from XAF Package to the directory of choice outside XAF Package under Linux environment. This directory is referred to as `<BASE_DIR>` in the following steps.
2. Set up environment variables to have Xtensa Tools in `$PATH` and `$XTENSA_CORE` defined to your HiFi core.
3. Execute `getTFLM.sh <target>` as below. This downloads and builds the tensorflow TFLM libraries in the directory `<BASE_DIR>/tensorflow`

   `$ ./getTFLM.sh hifi3/hifi3z/hifi4/hifi5/fusion_f1`
4. The following libraries will be created in directory:

   `libtensorflow-microlite.a` - TFLM Library

   `libmicro_speech_frontend.a` - Frontend lib for Microspeech Application

5. Path for HiFi 5 core:
   `<BASE_DIR>/tensorflow/tensorflow/lite/micro/tools/make/gen/xtensa_hifi5_default/lib/`

6. Path for other cores:

   `<BASE_DIR>/tensorflow/tensorflow/lite/micro/tools/make/gen/xtensa_fusion_f1_default/lib/`

You can copy `<tensorflow>` directory from Linux to Windows for building XAF Library and testbenches. In that case, the destination directory on Windows is your new `<BASE_DIR>`.

4.7 Building Multicore Subsystem

Multicore XAF tests require a multicore-subsystem to compile and run, so they would not compile and run OOB. A subsystem is required to build execute the software.

1. In the file `include/xaf-api.h`, `XF_EXTERNAL_INTERRUPT_NUMBER` is the BInterrupt number or external interrupt number, which is internally mapped to the Processor interrupt number in the file `include/sysdeps/mc_ipc/xf-mc-ipc.h`
2. The interrupt can be EDGE or LEVEL triggered.
3. For configurations with `XCHAL_HAVE_EXCLUSIVE`, the memory region used for global mutex lock object is required to be located in the shared memory region of `NON_CACHEABLE` attributes.

   The attributes can be set using `xthal_mpu_set_region_attribute`() with the following flags for all the DSPs in the subsystem before calling the APIs `xaf_adev_open` or `xaf_dsp_open`:

   ```
   XTHAL_MEM_NON_CACHEABLE,
   XTHAL_MEM_SYSTEM_SHAREABLE,
   XTHAL_MEM_BUFFERABLE.
   ```

   Example:

   ```
   err = xthal_mpu_set_region_attribute (addr_pointer, size_in_bytes,
               XTHAL_MPU_USE_EXISTING_ACCESS_RIGHTS,
               (XTHAL_MEM_NON_CACHEABLE   |
                XTHAL_MEM_SYSTEM_SHAREABLE |
                XTHAL_MEM_BUFFERABLE),
               0);
   ```
4. Multicore subsystem consists of the following sections with directory structure:

   Shared library: xf_shared

   ```
   xf_shared/include/xf-shared.h
   xf_shared/src/xf-shared.c
   ```

   Subsystem: example for 2 core 'multicore2c'

   ```
   multicore2c/spec.yml
   multicore2c/cluster.yml
   multicore2c/MMap0/MMap0.xld
   multicore2c/MMap0/BMap....yml
   multicore2c/bin (empty)
   multicore2c/params (empty)
   ```

   The following will be populated when 'subsystem' is built

   ```
   multicore2c/params
   multicore2c/bin/sysBuild
   ```

   The following will be populated when 'memory map (MMap)' is built

   ```
   multicore2c/bin/mBuild
   ```

4.7.1 Core Configuration Requirements

This section describes the core-configuration requirements for using cores to build multicore subsystem for multicore-XAF. The system files are available under $(ROOTDIR)/xtsc folder as described in the package directory structure (ROOTDIR is the base directory of the package).

Core configuration requirements

- Reference or base cores: HiFi 4 or HiFi 5 (current release extensively tested for HiFi 4)
- Additional configuration requirements:
  - PIFWriteResponse = 1 (`XCHAL_HAVE_PIF_WR_RESP` = 1)
  - `XCHAL_HAVE_S32C1I` = 1 (IsaUseSynchronization = 1) or `XCHAL_HAVE_EXCLUSIVE` = 1 (all the DSPs in the subsystem should have the same option)
  - PIFInbound = 1
  - `XCHAL_NUM_INTERRUPTS` > 0 (at least one edge or level triggered interrupt)

The system definition is provided in `yml` and `xtsys` file pairs in $(ROOTDIR)/xtsc folder

```
xtsc/xaf_xtsc_sys_2[3,4]c.yml
xtsc/xaf_xtsc_sys_2[3,4]c.xtsys
```

Note, the above files are for example usage and user should carefully review and update these for their multicore subsystem.

Note: `yml` and `xtsys` files mention core config as: `AE_HiFi4_LE5_XC_MOD_XTSC`. User must update it with their 'core config name' in both files.

User must update `yml` file for parameters like PIF width, local memory access widths, cache access widths, memory configurations etc.to exactly match their core configuration (Refer to section Custom Core-Configuration).

For memory size and partition updates, user must update respective details in `xtsys` file (Refer to section Custom Core-Configuration).

4.7.2 Updating the Shared Memory

The shared memory buffer and the buffer size are defined in xf_shared/src/xf-shared.c and xf_shared/include/xf-shared.h respectively. User should update the size of the shared buffer as required. The size should be within the allocated partition as specified in `xtsys` file.

The buffers required for global lock-objects of IPC and shared-memory management should also be allocated in the shared memory with additional attributes set using `xthal_mpu_set_region_attribute` as mentioned before.

Note: For cores with `XCHAL_HAVE_EXCLUSIVE` option enabled, the locks are required to be placed in a non-cached, shared memory segment. Such a memory segment can be created by appropriate modifications in `xtsys` file.

For example, we create `section(".sysram_uncached.data")` in our subsystems for which the following entries are required in `xtsys` file, when creating a subsystem on the command-line.

```
<hash n="memories">
<hash n="sysram_uncached" paddr="0x24fd8000" size="0x20000" writable="1"/>
<array n="partitions">
<hash corename="*" memname="sysram_uncached" name="shared_uncached_sram" offset="0x0" size="0x20000"/>
```

4.7.3 Custom Core-Configuration

The necessary sub system parameters definitions are provided in the files `.yml` and `.xtsys` .

For a custom HiFi-core, update the subsystem parameters to meet the requirements of Multicore-XAF, viz. core config name, PIF width, local memory access widths, cache access widths, memory configurations -etc. in the two subsystem files mentioned above.

yml:

- The file is used to build memory map/lsp using $XTENSA_TOOLS/libexec/xt-mbuild which generates the subsystem files in the folder `mbuild`
- SubSystemName: The subsystem name should match that in the xtsys file
- Interrupt number: `XF_EXTERNAL_INTERRUPT_NUMBER` in `xaf-api.h` should match the `BInterrupt` number in `.yml` file and should be same for all the cores.
- It is required to setup correct environment variables required for building the subsystem: viz. system paths(`XTENSA_SYSTEM`), tools paths(`XTTOOLS, XTENSA_TOOLS`), core-config (`XTENSA_CORE`).

A sample `yml` file which is available with the release package, is shown in the below table, along with the required parameters settings.

Table-1 Custom Core-Config Parameter List

Yml sample file parameters	Parameters to update (also available in config-params file)
`SubSystemName: multicore2c`	SubSystemName should match with the one in `xtsys`
`Processors:`	
`- Name: core0`	Name of the core should match with the one in `xtsys`. (e.g. "`core0`").
`Config: AE_HiFi4_LE5_XC_MOD_XTSC`	Configuration of reference core used to build core (`core0`)
`ProcId: 0`	Processor ID is a numeric constant between `0..NCORES-1`
`Master: true`	Master `True` for Master core (1 per subsystem), `False` for worker cores and each independent core is a master-core.
`Pipeline: LX`	
`StaticVectorSel: 0`	
`PifWidth: 8b`	PIF width in bytes
`DataRamFetchWidth: 8b`	DRAM width in bytes
`InstRamFetchWidth: 16b`	IRAM width in bytes
`InstRam0:`	
`LocalAddr: '0x58000000'`	IRAM address
`GlobalAddr: '0x88000000'`	
`Size: 128Mb`	IRAM size in Kb or Mb
`DataRam0:`	
`LocalAddr: '0x70000000'`	DRAM address
`GlobalAddr: '0x98000000'`	
`Size: 128Mb`	DRAM size in Kb or Mb
`NumBanks: 4`	Number of memory banks
`LocalMMIO:`	
`GlobalAddr: '0x80010000'`	
`Size: 4b`	
`InterruptRegister:`	
`GlobalAddr: '0x80010000'`	
`InterruptMap:`	
`- BInterrupt: 7`	External interrupt number
`ProcInterrupt: 7`	Processor interrupt number
`BitOffset: 0`	External Interrupt bit offset in the interrupt mask
`Type: Edge`	Interrupt type: Level or Edge
`Dcache:`	
`LineSize: 128`	DCACHE line size in bytes

`SystemRAM:`	
`  GlobalAddr: '0x20000000'`	Update SRAM address
`  Size: 80Mb`	SRAM size in Kb or Mb
`  ReadDelay: 1`	SRAM read memory latency in cycles
`  ReadRepeat: 1`	
`  WriteDelay: 1`	SRAM write memory latency in cycles
`  WriteRepeat: 1`	
`  RequestFIFODepth: 255`	
`SystemROM:`	
`  GlobalAddr: '0x50000000'`	SROM address
`  Size: 12Mb`	SROM size in Kb or Mb
`SubSystemInterconnect:`	
`- Src: CommonBus`	
`  Dests:`	
`  - core0`	
`  - core1`	
`DeviceMemories:`	
`- Name: DeviceMemory0`	
`  GlobalAddr: '0x80000000'`	
`  Size: 64Kb`	

xtsys:

The file is used to build xtsc execution environment using $XTENSA_TOOLS/libexec/xt-sysbuilder which generates the subsystem files into `sysbuilder` folder

- The names in both the files `yml` and `xtsys` should match

  `<hash n="system" name="multicore2c" t="MultiCoreSystem">`.

- Update the name of the core with name and the reference configuration config

  ```
  <hash config="AE_HiFi4_LE5_XC_MOD_XTSC" name="core0"
  vecselect="0"/>
  ```

- The shared memory buffer and buffer size are defined in a separate `xf-shared.c` and `xf-shared.h` files. User should update the size of the shared buffer as required. The size should be within the allocated partition of system-RAM as specified in the file `xtsys` as follows:

```
<hash n="sysram" paddr="0x20000000" size="0x4fd8000" writable="1"/>
<array n="partitions">
<hash corename="*" memname="sysram" name="shared_sram"
offset="0x1800000" size="0x37d8000"/>
```

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 uses 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 additional 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 example, 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-mp3-dec-rend-test.c`, `xaf-playback-usecase-test.c`
Cadence MP3 encoder [5]	Audio [2]	Encodes MP3 data	Folder: `mp3_enc` Application: `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 converter [8]	Audio [2]	Converts sampling rate	Folder: `src-pp` Application: `xaf-playback-usecase-test.c`
Cadence AAC decoder [9]	Audio [2]	Decodes AAC data	Folder: `aac_dec` Application: `xaf-playback-usecase-test.c`
Cadence Opus encoder [11]	Speech [3]	Encodes Opus data	Folder: `opus_enc` Application: `xaf-full-duplex-opus-test.c`
Cadence Opus decoder [11]	Speech [3]	Decodes Opus data	Folder: `opus_dec` Application: `xaf-full-duplex-opus-test.c`

6. Known Issues

The current version of XAF has only been tested with Version RI-2021.6 of the Xtensa tool chain with XT-CLANG compiler. The Instruction Set Simulator (ISS) and Xtensa System C (XTSC) 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` and `xaf_dsp_open` APIs and depends on the three parameters `audio_component_buffer_size`, `audio_framework_buffer_size` and `audio_shmem_buffer_size` passed to the above functions.

1. `audio_component_buffer_size`: This is the memory allocated by XAF for usage by audio components. Local buffers required by audio components such as connect buffers between components, persist buffers, or scratch buffer are allocated from this memory. Also, if pre-emptive scheduling is enabled, memory required for the worker threads is allocated from this memory. Buffers required for event communication are also allocated from this memory. Note, if error channel is enabled, additional memory of 96 bytes per component is required.

2. `audio_framework_buffer_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. Note, if error channel is enabled then `num_err_msg_buf` of size 4 bytes each, aligned to 64 bytes are created. This requires additional memory of 64 bytes per error message buffer.

   When NCORES>1 this buffer is allocated from global shared memory. In “non zero-copy mode” of `xaf_get_config_ext` and `xaf_set_config_ext` APIs the required buffers (whose size is determined by the variable `cfg_param_ext_buf_size_max` of `xaf_comp_config_t` structure, and an additional 256 bytes) are allocated from this memory.

   Note: This buffer is only allocated by master core. Thus, `audio_framework_buffer_size` should be zero on worker core application.

3. `audio_shmem_buffer_size`: This is the pre-allocated global shared memory required only when framework is built with NCORES>1. It is used for allocating connect buffers and event buffers between components from two different DSPs.

Table 7-1 List of Buffers

Sr No.	Type of Buffer	Type of Memory (Memory Pool)	
		NCORES = 1	NCORES > 1
1.	Connect buffers	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size) Global shared Memory (audio_shmem_buffer_size)
2.	Input buffer	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size)
3.	Output buffer	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size)
4.	Persist buffers	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size)
5.	Scratch buffer	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size)
6.	Stack for worker threads	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size)
7.	Buffers for `xaf_get_config_ext` and `xaf_set_config_ext`	Local Memory (audio_framework_buffer_size)	Global shared Memory (audio_framework_buffer_size)
8.	Event buffers (Events between Application and Component, Framework E.g. Error channel buffers)	Local Memory (audio_framework_buffer_size)	Global shared Memory (audio_framework_buffer_size)
9	Event buffers (Events between Components)	Local Memory (audio_component_buffer_size)	Local Memory (audio_component_buffer_size) Global shared Memory (audio_shmem_buffer_size)
10.	Message buffers for communication between	Local Memory (audio_framework_buffer_size)	Global shared Memory (audio_framework_buffer_size)

	application and audio components		
11.	Message pool on DSP	Local Memory (audio_component_buffer_size)	Global shared Memory (audio_shmem_buffer_size)

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).

D is size of message pool that needs to be allocated on the master DSP. Size of this pool is 256 * cache line size bytes. An additional 1KB per core and 2 KB, independent of number of cores is required.

$$D = D_1 + D_2,$$

$$D_1 = \begin{cases} 256 * \mathit{cache\ line\ size} & \mathit{if\ NCORES} > 1 \\ 16\,KB & \mathit{if\ NCORES} = 1 \end{cases}$$

$$D_2 = \begin{cases} 1\,KB * (\mathit{Number\ of\ cores}) + 2\,KB & \mathit{if\ NCORES} > 1 \\ 0 & \mathit{if\ NCORES} = 1 \end{cases}$$

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

- Audio component buffer of size `audio_component_buffer_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:

$$T = T_1 + T_2 + T_3, \qquad T_1 = \sum_{n=1}^{N} (P_n + A_n I_n + B_n O_n Z_n + 0.25 * Z_n), \qquad T_2 = \max_n S_n$$

$$T_3 = \sum_{n=1}^{N} \begin{cases} B_n O_n Y_n & \mathit{for\ audio\text{–}codec\text{–}class} \\ 0 & \mathit{otherwise} \end{cases}$$

T_1 is the sum of the persistent, input, output sizes and overhead memory required for connect buffer 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 in `xaf_adev_config_default_init` via the compile time constant `XF_CFG_CODEC_SCRATCHMEM_SIZE` and T_2 is user-configurable. T_3 is the additional memory required by audio-codec-class components for initialization. Furthermore, some memory is required by XAF itself. The size of the memory required by XAF is (2N + D) KB, where N is the number of components. Note that, this 2 KB per component includes each component's API-structure, memory table, and miscellaneous audio-framework data structures for the component.

Thus, `audio_component_buffer_size` should be set to a value greater than (T + T_2 + 2N) KB + D KB if NCORES = 1.

Notes on `audio_component_buffer_size`:

i. An additional 32 bytes per allocation are required each time a memory allocation is done for a component to provide the aligned pointer. This is absorbed in 2N KB of extra memory per component as mentioned above. Thus, for every additional 32 memory allocations, 1 KB of extra memory is required (for example, 2N KB in the above formula would become 3N KB).

ii. Additional memory required when pre-emption enabled:

(1) XOS: 1240 bytes for thread-structure and 8192 bytes for thread-stack for each of the priority (`n_rt_priorities)` and non-priority (`bg_priority`) threads.

Example: xaf_adev_set_priorities (p_adev, 2, 3, 2) requires 3*1240 + 3*8192 bytes.

(2) FreeRTOS: 32 bytes each for thread-structure for all priority (`n_rt_priorities)` and non-priority (`bg_priority`) threads.

Example: xaf_adev_set_priorities(p_adev, 2, 3, 2) requires 3*32 bytes.

(3) T_2 bytes of scratch memory (of size `XF_CFG_CODEC_SCRATCHMEM_SIZE)` per priority thread.

XAF buffer of size `audio_framework_buffer_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. Note, all buffer allocations have a cache line size overhead and minimum alignment value is 1 (for NCORES=1) and maximum supported alignment value is 4096.

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 24 KB, independent of the number of components.

Thus, `audio_framework_buffer_size` should be set to a value greater than (S + 24) KB.

XAF buffer of size `audio_shmem_buffer_size` (NCORES>1): If source and destination component of the connect buffer are on different cores, then connect buffer is allocated from the corresponding buffer pool. Additional 0.25 KB overhead is required per connect buffer.

$$U = U_1, \; U_1 = \sum_{n=1}^{N} (B_n O_n Z_n + 0.25 * Z_n),$$

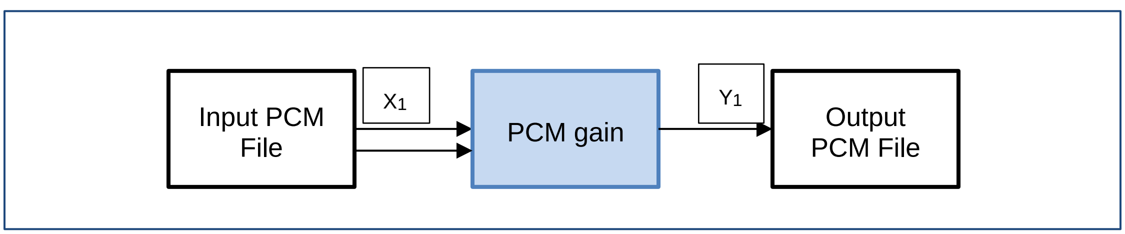
Message pool of size D is also allocated from this buffer.

Thus, `audio_shmem_buffer_size` should be set to a value greater than (U + D) 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_HiFi4_LE5 core.

- Example 1: “PCM_Gain”(xa_af_hostless_test) with NCORES= 1

  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_component_buffer_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_2 = 56 KB

  T_3 = 1(B_1) * 4(O_1) = 4 KB

  D = 16(D_1) + 0(D_2) = 16 KB

$T = 4\ (T_1) + 56\ (T_2) + 2\ (N) + 16\ (D) + 4\ (T_3) = 82$ KB is the required `audio_component_buffer_size`.

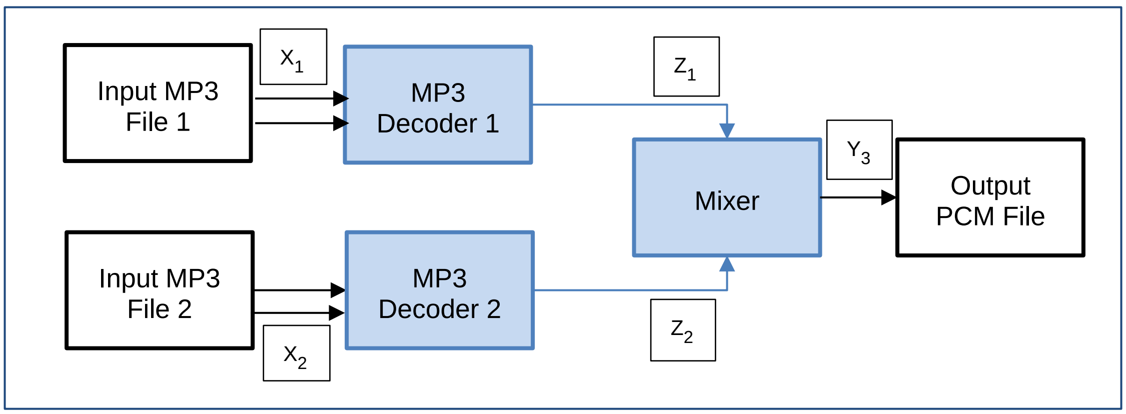
- audio_framework_buffer_size Computation:

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

  $S + 20 = 12 + 24 = 36$ KB is the required `audio_framework_buffer_size`.

- Example 2: “2 MP3 Decoder + Mixer” (xaf-dec-mix-test) with NCORES= 1

  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_component_buffer_size Computation:

  sum1 = $12.125\ (P_1) + 1\ (A_1) * 2\ (I_1) + 1\ (B_1) * 4.5\ (O_1) * 4\ (Z_1) + 0.25 * 4(Z_1) = 33.125$ KB

  sum2 = $12.125\ (P_2) + 1\ (A_2) * 2\ (I_2) + 1\ (B_2) * 4.5\ (O_2) * 4\ (Z_2) + 0.25 * 4(Z_2) = 33.125$ KB

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

T_1 = 33.125 + 33.125 + 8 = 74.25 KB

T3 = 1 (B_1) * 4.5 (O_1) = 4.5 KB

D = 16 (D_1) + 0 (D_2) = 16 KB

T = 74.25 (T_1) + 56 (T_2) + 2*3(N) + 4.5 (T_3)+ 16 (D) = 156.75 KB is the required `audio_component_buffer_size`.

- audio_framework_buffer_size Computation:

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

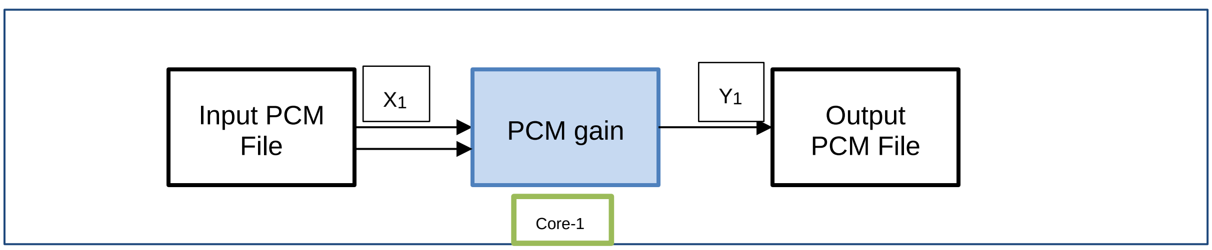
  sum2 = 4 * 1 (A_2) * 2 (X_2) + 4.5 (O_2) * 1 (B_2) * 0 (Y_2) = 8 KB

  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 + 24 = 42 KB is the required `audio_framework_buffer_size`.
- Example 3: “PCM_Gain”(xa_af_hostless_test) with NCORES = 2

  In this example, PCM Gain is on core-1 (Nc_0 = 0 Nc_1 = 1 where Nc_0 is the number of components on core-0)



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_component_buffer_size Computation:

  Core-0 (Master core):

  Nc_0 = 0

  Thus 0 KB is the required `audio_component_buffer_size`.

  Core-1(Worker core):

  Nc_1 = 1

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

$T_2 = 56$ KB

$T_3 = 1(B_1) * 4(O_1) = 4$ KB

$T = 4\ (T_1) + 56(T_2) + 2 *1(N_{c1}) + 4\ (T_3) = 66$ KB is the required `audio_component_buffer_size`.

- audio_framework_buffer_size Computation:

  Core-0 (Master core):

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

  S + 20 = 12 + 24 = 36 KB is the required `audio_framework_buffer_size`.

  Core-1 (Worker core):

  0 KB is the required `audio_framework_buffer_size`.

- audio_shmem_buffer_size Computation:

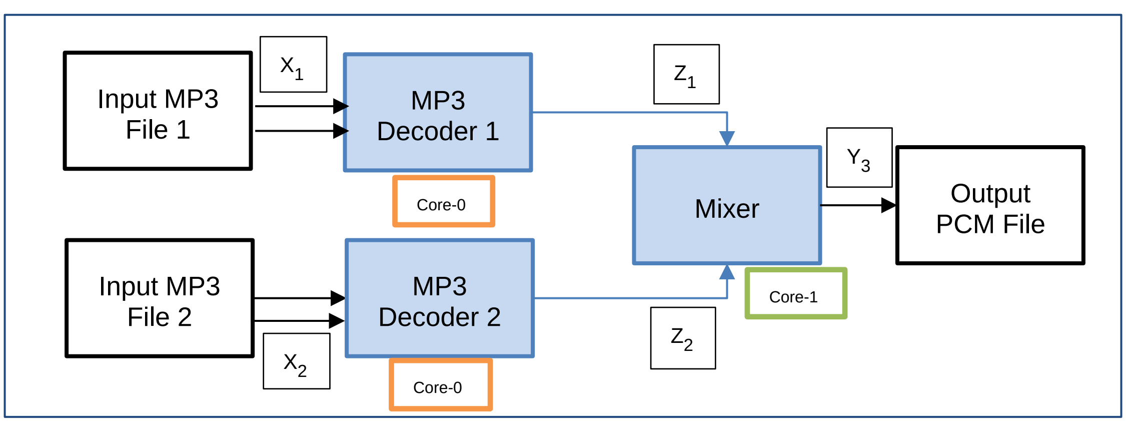
  D = 256 (Number of messages) * 0.125 KB (Size of a cache aligned message) + 1 KB * 2 (Number of cores) + 2 KB = 36 KB

  U = 0

  D + U = 36 + 0 = 36 KB is the required `audio_shmem_buffer_size`.

- Example 4: "2 MP3 Decoder + Mixer" (xaf-dec-mix-test)

  Number of components, N =3 (MP3 Decoder1, MP3 Decoder2, Mixer). In this example, MP3 Decoder1 and MP3 Decoder2 are on Core-0 while Mixer is on Core-1 (N_{c0} =2, N_{c1} =1).



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.

Note, if both source and destination component are on different cores, connect buffers are allocated from `audio_shmem_buffer` pool, else, they are allocated from `audio_comp_buffer` pool.

- audio_component_buffer_size Computation:

Computation for core-0 (Master core):

sum1 = 12.125 (P_1) + 1 (A_1) * 2 (I_1) + 1 (B_1) * 4.5 (O_1) * 0 (Z_1) = 14.125 KB

sum2 = 12.125 (P_2) + 1 (A_2) * 2 (I_2) + 1 (B_2) * 4.5 (O_2) * 0 (Z_2) = 14.125 KB

T_1 = 14.125 + 14.125 = 28.25 KB

T_2 = 56 KB

T_3 = 1 (B_1) * 4.5 (O_1) = 4.5 KB

T = 28.25 (T_1) + 56 (T_2) + 2*2(N_{c0}) + 4.5 (T_3) = 92.5 KB is the required `audio_component_buffer_size`.

Computation for core-1(Worker core):

Sum1 = 0 (P_3) + 4 (A_3) * 2 (I_3) + 1 (B_3) * 2 (O_3) * 0 (Z_3) = 8 KB

T_1 = 8 KB

T = 8 (T_1) + 56 (T_2) + 2*1(N) = 66 KB is the required `audio_component_buffer_size`.

- `audio_framework_buffer_size` Computation:

Core-0(Master core):

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 + 24 = 42 KB is the required `audio_framework_buffer_size`.

Core-1 (Worker core):

0KB is the required audio_framework_buffer_size.

- audio_shmem_buffer_size Computation:

  D = 256(Number of messages) * 0.125 (Size of a cache aligned message) + 1 * 2 (Number of cores) + 2 = 36 KB

  U = 1 (B_1) * 4.5 (O_1) * 4 (Z_1) + 1 (B_1) * 4.5 (O_1) * 4 (Z_2) + 0.25 * 4 (Z_1) + 0.25 * 4 (Z_2) = 38 KB

  D + U = 36 + 38 = 74 KB is the required `audio_shmem_buffer_size`.

8. Appendix: OSAL APIs

Operating System Abstraction Layer (OSAL) is defined for all RTOS functionality requirements in XAF. Table 8-1 lists all OSAL APIs that are defined and used in XAF. 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 that the Timer APIs listed in Table 8-1 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.

OSAL APIs List

Table 8-1 OSAL APIs

API Class	OSAL API Defined in XAF
Message Queue APIs	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 APIs	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 APIs	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);

API Class	OSAL API Defined in XAF
Event APIs	void **__xf_event_init** (xf_event_t *event, uint32_t mask); void **__xf_event_destroy** (xf_event_t *event); 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 APIs	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 APIs	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 the 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 the 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
```

Note that while building your test bench example for a particular HiFi DSP configuration, make sure to link the FreeRTOS library that is built for the same HiFi DSP configuration.

Multicore IPC Abstraction API List

Table 8-2 Multicore IPC APIs

API Class	Multicore-IPC abstraction API Defined in XAF
Mutex APIs	uint32_t __xf_ipc_lock(xf_ipc_lock_t *lock) uint32_t __xf_ipc_unlock(xf_ipc_lock_t *lock)
Interrupt APIs	void __xf_ipc_interrupt_notify(uint32_t core) void __xf_ipc_interrupt_clear(uint32_t core)
Reset Sync API	int __xf_ipc_reset_sync(void);

Multicore-IPC APIs are declared in the following header files:

```
/include/sysdeps/mc_ipc/xf-mc-ipc.h
```

Selection of the System Timer in Timer APIs

The system timer selected to generate interrupts for capturer and renderer is, by default, such that the timer has the highest interrupt-priority not exceeding EXCMLEVEL priority.

For XOS, passing argument -1 would select such a timer at the time of execution (`xos_start_system_timer(-1, TICK_CYCLES)`) or by directly specifying a timer number with appropriate priority (`xos_start_system_timer(0, TICK_CYCLES)`).

For FreeRTOS, preprocessor logic selects such a timer during compilations of FreeRTOS library.

Interrupt Handler Implementation with XAF

The interrupt handler for capturer and renderer components must be implemented using the `__xf_set_threaded_irq_handler` API. This threaded interrupt handler splits interrupt processing into two parts. The first part (`irq_handler`) runs in interrupt context and should do minimal, critical work (acknowledge, clear the interrupt etc.). The second part (`threaded_handler`) runs in a high priority background thread, can be context switched, and does the rest of the interrupt processing. Note, the high priority background thread mentioned above is created by XAF during DSP Interface Layer initialization at highest priority available with RTOS only for interrupt processing.

The XAF schedules capturer and renderer processing through callback function upon receiving respective interrupt. This should be implemented in `threaded_handler` as it requires to acquire RTOS lock to access XAF scheduler.

Note that the capturer and renderer in XAF package mimic real time interrupts using the timer interrupts and therefore do not use `__xf_set_threaded_irq_handler` API.

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 for Tensilica HiFi DSPs.

[5] *Cadence MP3 Encoder* – Library version 1.6 for Tensilica HiFi DSPs. The library must be rebuilt from sources for HiFi 4.

[6] *Cadence AMR-WB Decoder* – Library version 2.7 for Tensilica HiFi DSPs.

[7] *Cadence AMR-WB Decoder* – Library version 2.3 for Tensilica HiFi DSPs.

[8] *Cadence Sample Rate Converter* – Library version 1.9 for Tensilica HiFi DSPs.

[9] *Cadence AAC Decoder* – Library version 3.7 for Tensilica HiFi DSPs.

[10] *Cadence Ogg-Vorbis Decoder* – Library version 1.12 for Tensilica HiFi DSPs.

[11] *Cadence Opus Codec* – Library version 1.8 for Tensilica HiFi DSPs.

[12] *Xtensa port of FreeRTOS* – https://github.com/foss-xtensa/amazon-freertos/tree/xtensa-v10.2.1-xaf

[13] *TensorFlow* – https://github.com/tensorflow/tensorflow