SOFTWARE ARCHITECTURE TEMPLATE

BIOSPSP

Audio Interface Driver Design Document

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Sandeep K		14th Mar- 2012	

Table of Contents

1 System Context 4
1.1 Terms and Abbreviations 4
1.2 Disclaimer 4
1.3 Related Documents 4
1.4 Hardware 5
1.5 Software 5
1.6 Design Philosophy 5
1.7 Design Description 5
1.7.1 Operating Environment and dependencies 7
1.7.2 System Architechture 7
1.7.3 Design Goals 8
1.8 Component Interfaces 9
1.8.1 IOM driver Interface 9
1.9 Design Philosophy 10
1.9.1 The Module and Instance Concept 10
2 AUDIO Driver Software Architecture 11
2.1 Static View 11
2.1.1 Functional Decomposition 11
2.1.2 Data Structures 11
2.1.3 Channel Parameters(Audio_ChannelConfig) 13
2.1.4 Audio_IoctlParam 13
2.2 Audio Interface driver data types 14
2.2.1 Audio_DeviceType 14
2.2.2 Audio_IoMode 14
2.2.3 Audio_ModuleSel ..... **Error! Bookmark not defined.**
2.3 Dynamic View 16
2.3.1 Input / Output using Audio driver 16
2.3.2 Functional Decomposition 16
3 IOCTL commands 25

List Of Figures

Figure 1 Sample Hardware configuration..... **Error! Bookmark not defined.**
Figure 2 System Architecture 7
Figure 3 audioMdBindDev() control Flow..... 17
Figure 4 audioMdUnBindDev() control Flow 18
Figure 5 audioMdCreateChan() control flow 20
Figure 6 audioMdDeleteChan() control Flow..... 21
Figure 7 audioMdControlChan() control Flow..... 23
Figure 8 audioMdSubmitChan() control flow 24

1 System Context

The Audio interface driver architecture presented in this document is situated in the context of SYS BIOS 6.x based drivers. But the design is relevant to other architectures also as the audio interface driver is essentially independent of the hardware.

Note: The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

1.1 Terms and Abbreviations

Term	Description
API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System
Audio driver	Audio interface driver
Audio device driver	Driver for audio devices like McASP or McBSP etc
Audio codec driver	Driver for audio codec devices like TLV320AIC31 etc

1.2 Disclaimer

This is a design document for the Audio interface driver for the SYS/BIOS operating system. Although the current design document explain the Audio interface driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation as the BIOS 5.x driver is a direct port of BIOS 6.x driver. The BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IOM driver model. The subsequent section explains how this document can be used to understand and modify the IOM drivers found in this product. Please note that all the flowcharts, structures and functions described here in this document are equally applicable to the Audio interface driver 5.x.

1.3 Related Documents

1.	TBD	SYS/BIOS Driver Developer's Guide
2.	SPRUFM1	Mcasp user guide (draft)

1.4 Hardware

The audio interface driver is essentially independent of the underlying hardware.

1.5 Software

The Audio interface driver discussed here is intended to run in SYS/BIOS™ V6.xx on the C674x DSP.

1.6 Design Philosophy

This device driver is written in conformance to the SYS/BIOS IOM driver model and handles the configuration of the Aic31 ADC and DAC sections independently.

A codec performs conversion of audio streams from digital to analog formats and vice versa. It involves configuring the DAC and ADC sections of the codec. An application might be using multiple instances of AIC31 codecs in a single audio configuration. In such a case the application needs to configure each audio codec independently.

The design of the SYS BIOS based AIC31 IOM driver intends to make the configuration of all AIC31 codecs simple. Using an Aic31 IOM driver interface makes it possible to configure multiple codecs by specifying their control bus and their address on the bus. The Aic31 driver provides the flexibility to configure the required configuration for all the instances during the build time(it is also possible to do so dynamically). It also allows the application to manage multiple instance of the codec with a single interface

1.7 Design Description

This section describes the need for a separate audio interface driver over the audio device driver and audio codec drivers.

In a given hardware platform there are many different types of audio data transport devices (McASP, McBSP etc) are present. Also, there might be multiple instances of each of these devices available. Each of these devices in turn will be interfacing with multiple codecs available on board. Also the codecs can be controlled using different control buses like Audio or Spi etc.). The diagram below shows an imaginary scenario.

The hardware consists of one Mcasp and one McBSP instance. There are one instance of AIC31 codecs on board. All the codecs are configured using the Audio instance 0. The Mcasp instance 2 interfaces with the AIC31 instance 0.

This kind of complex audio configuration requires the sample applications to configure each of the audio device and audio codec individually. In case of multiple audio codecs to be handled, it is required to do bookkeeping of all the codecs to be configured and the channel handles returned by them.

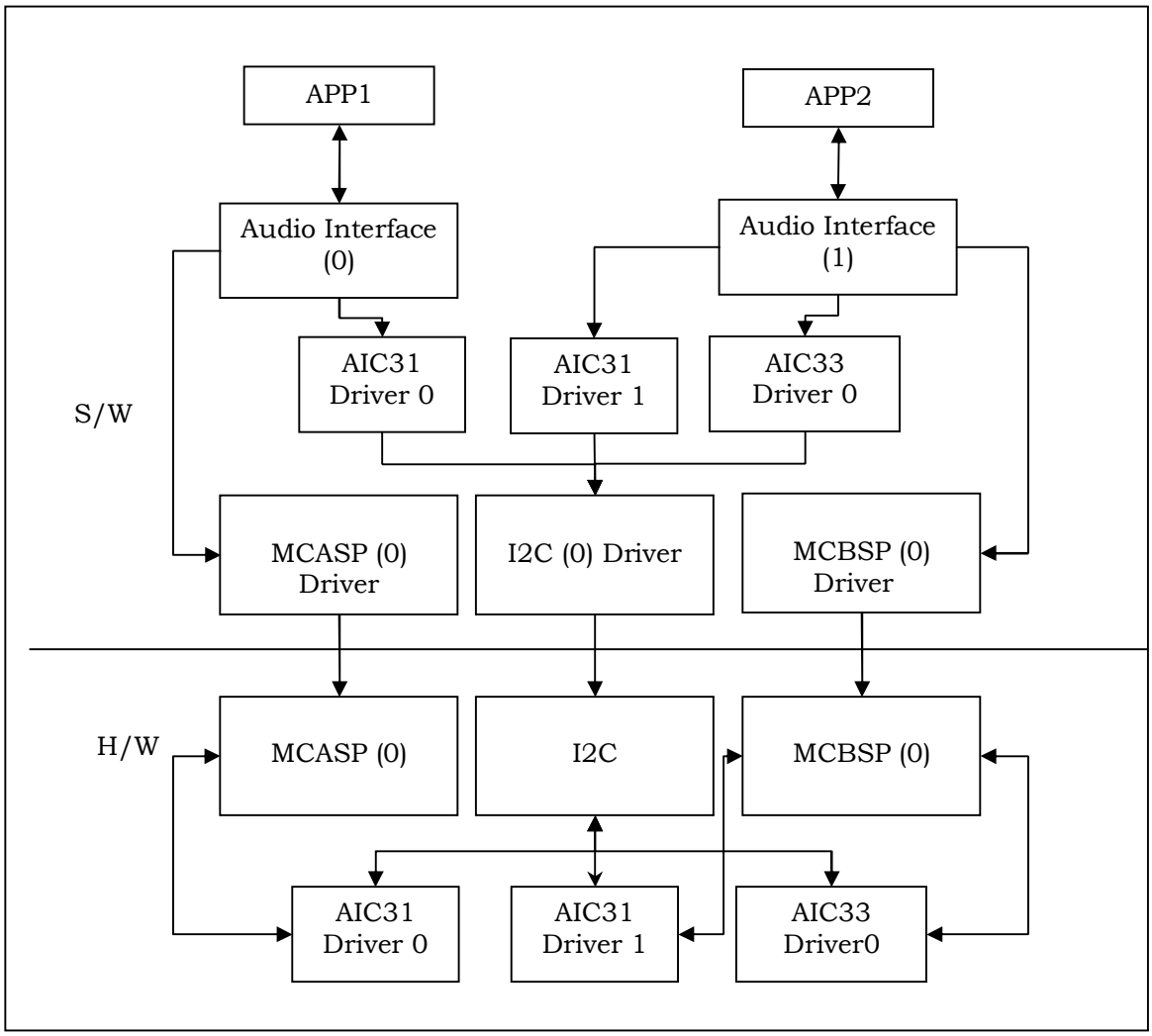

Figure 1 Sample Hardware configuration

In such complex scenarios the audio interface driver provides the application developers to configure the entire audio configuration through a single interface .it maintains all the internal bookkeeping required and thereby providing the application developer with a simplified interface.

The design of audio interface driver is intended to provide the necessary abstraction to the application in interacting with the various audio configurations so that the application can be ported from one platform to another easily with minimum changes. It is also intended that the audio interface driver will also facilitate in easy addition of new audio configurations like addition of new codecs etc, new audio devices etc with minimum code changes.

1.7.1 Operating Environment and dependencies

Details about the tools and SYS/BIOS versions that the driver is compatible with, can be found in the system Release Notes.

1.7.2 System Architechture

The block diagram below shows the overall system architecture.

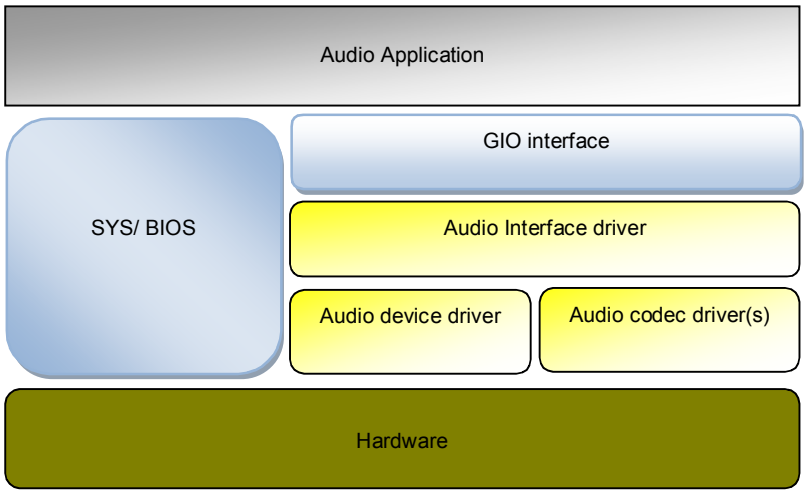

Figure 2 System Architecture

The Audio interface driver is a complete IOM driver comaptible driver.it lies below the stream layer, which is an class driver layer provided by SYS BIOS™ (please note that the stream layer is designed to be OS independent with appropriate abstractions).Depending on the audio configurations it is interfacing with the audio interface driver will interface with an audio device driver (like Mcasp driver or Mcbsp driver) and also with the required number of codec drivers.

The Application can use the stream APIs to interface with the audio interface driver.The driver inturn will interface with the underlying drivers.

Figure 2 shows the overall device driver architecture. For more information about the IOM driver model, see the SYS/BIOS™ documentation. The rest of the document elaborates on the architecture of the Device driver by TI.

1.7.3 Design Goals

The following are the key device driver design goals being factored by proposed audio interface driver:

1. Simple unified interface for the application to access the various audio configurations through a set of well defined APIs.
2. Easy addition of new audio devices and audio codecs with minimum changes in the application.
3. Easy porting of applications using the audio interface drivers across various platforms.
4. Audio interface driver should be platform independent.

Note:

1. The audio interface driver will have a maximum of 1 audio device in a given instance.
2. The audio interface driver will not perform any IO transfers with the audio codec other than configuring the codec and controlling the codec.(no data transfers).
3. The audio interface driver will not perform any module startup checks for the underlying audio device drivers and audio codec drivers.

1.8 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The audio interface driver module is an object of IOM driver class one may need to refer the BIOS IOM driver documentation to access the audio interface driver in raw mode or could refer the GIO APIs to access the driver through GIO layer abstraction.

1.8.1 IOM driver Interface

The IOM Driver constitutes the Device Driver Manifest to application. This adapts the Driver to SYS/BIOS™. This i2c driver is intended to inherit the IOM Driver interfaces. Thus the i2c driver module becomes an object of IOM Driver class. Please note that the terms “Module” and “Driver or IOM Driver” would be used in this document interchangeably.

Audio module specifications file (Audio.h)

The file defines the following in its public section: the data structures, enums, constants, IOCTLS, error codes and module wide config variables that shall be exposed for the user.

These definitions would include

ENUMS: Operation modes, device types etc.

STRUCTURES: channel configuration structures etc.

CONSTANTS: error ids and ioctls and various other constants.

Also this files specifies the list of configurable items which could be configured/specified by the application during instantiation (instance parameters)

In its private section it would contain the data structures, enums, constants and module wide config variables which are internal to the driver. The Instance objects (the driver object) and channel objects which contain all the info related to that particular IO channel. This information might be irrelevant to the User. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application

The driver header file (Audio.h) shall be included by the applications, for referring to any of the driver data structures/components.

A driver which conforms to the IOM driver model exposes a well define set of interfaces -

- Driver initialization function.
- IOM Function pointer table.

Hence the AUDIO driver (because of the virtue of its IOM driver model compliance) exposes the following interfaces.

- Audio_init()
- Audio_IOMFXNS

The Audio_init() is a startup function that needs to be called by the user (application) to initialize the module state structure of the AUDIO instance.

- The working of the Audio driver will be affected if this function is not called by the application prior to accessing the AUDIO driver APIs.

The Model used by the device is identified by the function table. Hence, IOM_Fxns used for IOM model.

The AUDIO driver exposes IOM function pointer table which contains various APIs provided by the AUDIO driver. The IOM mini-driver implements the following API interfaces to the class driver**:**

Function Name	Description
audioMdBindDev	Function to create a new instance of the audio interface dynamically
audioMdCreateChan	Function to open a channel for data communication
audioMdSubmitChan	Function to transfer/receive data using a previously opened channel
audioMdControlChan	Function to pass control commands to the audio device and codecs
audioMdDeleteChan	Function to close a previously opened channel
audioMdUnBindDev	Function to delete the audio interface driver instance.

1.9 Design Philosophy

This device driver is written in conformance to the SYS/BIOS IOM driver model and handles communication to and from an Audio configuration (Audio device plus the codecs it is interfacing with).

1.9.1 The Module and Instance Concept

The IOM driver model, provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation.

The module word usage (Audio module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall and thus shall configure the module behavior, shall be a module variable. However, there can also be module wide constants. For example, whether the function parameters need to be checked can be enabled or not can be selected by setting module wide variable “paramCheckEnable” and can be set by the application.

This instance word usage (Audio instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent an instance of device directly by holding info like the TX/RX channel handles, device configuration settings, hardware configuration etc. This is represented by the Instance_State in the Audio module configuration file. Hence every module shall only support the as many number of instantiations as the number of Audio devices hardware instances on the SOC.

2 AUDIO Driver Software Architecture

This section details the data structures used in the Audio IOM driver module and the interface it presents to the GIO layer. A diagrammatic representation of the IOM driver module functions is presented and then the usage scenario is discussed in some more details.

Following this, we'll discuss the dynamic view of the driver where the driver operational scenarios are presented.

2.1 Static View

2.1.1 Functional Decomposition

The driver is designed keeping a device, also called instance, and channel concept in mind.

This driver uses an internal data structure, called channel, to maintain its state during execution. This channel is created whenever the application calls a gio create call to the Audio IOM driver module. The channel object is held inside the Instance State of the module. The data structures used to maintain the state are explained in greater detail in the following *Data Structures* sub-section.

2.1.2 Data Structures

The IOM driver employs the Instance State (Audio_Object) and Channel Object structures to maintain state of the instance and channel respectively.

In addition, the driver has two other structures defined – the device params and channel params. The device params structure is used to pass on data to initialize the driver during module start up or initialization. The channel params structure is used to specify required characteristics while creating a channel. For current implementation channel parameters only contain the channel parameters required by the individual audio device driver and the audio codec drivers.

The following sections provide major data structures maintained by IOM driver module and the instance.

2.1.2.1 The Instance State (Audio_Object)

The instance state comprises of all data structures and variables that logically represent the actual instance. It preserves the input and output channels for transmit and receive, parameters for the instance etc.

Element Type	Element Name	Description
UInt8	instNum	Instance number of the audio interface driver
Audio_DriverState	devState	Status of the audio interface driver instance (created/deleted)
Audio_DeviceType	adDevType	Audio device type
String	adDevName	Audio device driver name in driver table
GIO_Device	*adDevHandle	Handle to the Audio device IOM Driver
Mcasp_AudioDevData	adAudioDevData	Information related to the audio device
UInt8	acNumCodecs	Number of codecs in this instance
String	acDevName[Audio_NUMCODECINSTANCES]	Handle to the audio codec returned during bind function
GIO_Device	*acDevHandle[Audio_NUMCODECINSTANCES]	Handle to the Audio codec IOM Driver
ICodec_CodecData	acCodecData[Audio_NUMCODECINSTANCES]	Information related to the codec
Audio_Channel_Object	ChanObj[NUM_CHANS]	TX and RX channel objects

2.1.2.2 The Channel Object

The interaction between the application and the device is through the instance object and the channel object. While the instance object represents the actual hardware instance, the channel object represents the logical connection of the application with the driver (and hence the device) for that particular IO direction. It is the channel which represents the characteristic/types of connection the application establishes with the driver/device and hence determines the data transfer capabilities the user gets to do to/from the device. For example, the channel could be input/output channel. This capability provided to the user/application, per channel, is determined by the capabilities of the underlying device. Per instance we have two channels, each for transmit and receive.

Element Type	Element Name	Description
Audio_DriverState	chanState	Channel status variable(opened /closed)
Ptr	aiAudioChanHandle	Channel handle for the audio device channel
UInt32	aiAudioCodecHandle [numCodecInstances]	Array holding the channel handles of all the audio codec driver channels
IOM_TiomCallback	cbFxn	Callback function specified by the stream
Arg	cbArg	Argument to the call back function
Audio_ChannelConfig	aiChannelConfig	Structure holding the channel parameters

Ptr	devHandle	Instance handle

2.1.3 Channel Parameters(Audio_ChannelConfig)

This configuration structure is used to specify the audio interface driver channel initialization parameters. The audio interface driver can be opened in two modes (RX and TX). Hence there will be one channel parameters structures one for each channel (i.e. two channel parameter structures for each instance of audio interface).

The structure internally contains the channel parameters required by the audio device and the audio codecs. This structure needs to be supplied by the user during the opening of the audio interface channel or else the default values will be taken by the driver.

Element Type	Element Name	Description
Ptr	chanParam	Pointer to audio device channel configuration parameters
ICodec_ChannelConfig	acChannelConfig	Audio codec channel configuration data structures for all codecs

2.1.4 Audio_IoctlParam

The audio interface driver interfaces to multiple drivers. Hence an application will need to supply additional information (like the device to which the command is to be passed, the extra information required by the command etc) when using a control command on any device. This structure is used by the application to specify the additional information that will be required during an IOCTL command to the device. This data structure will carry the IOCTL command argument and the information about which device this command is addressed to etc.

Element Type	Element Name	Description
Audio_ModuleSel	aiModule	Device to pass the command to (Audio device / audio codec)
Uint32	codecId	Instance number of the codec(if aiModule is audio codec)
Ptr	ioctlArg	Pointer to command specific data

2.2 Audio Interface driver data types

This section describes the data types used by the audio interface driver.

2.2.1 Audio_DeviceType

The “Audio_DeviceType” specifies the type of audio device supported by the audio interface driver framework. This is used by the driver to identify the type of audio device it has to communicate with.

Enumeration Class	Enum	Description
Audio_DeviceType	Audio_DeviceType_McASP	Audio Device type is McASP.
	Audio_DeviceType_McBSP	Audio Device type is McBSP.
	Audio_DeviceType_VOICE_CODEC	Audio Device type is voice codec
	Audio_DeviceType_UNKNOWN	Audio Device type is unknown

2.2.2 Audio_IoMode

The “Audio_IoMode” enumerated data type specifies the operational mode of the channel i.e. either transmission mode or reception mode.

Enumeration Class	Enum	Description
PSP_audioIfIoMode	Audio_IoMode_RX	Audio Interface channel mode is RX
	Audio_IoMode_TX	Audio Interface channel mode is TX

2.2.3 Audio_Ioctl

The “Audio_Ioctl” enumerated data type specifies the IOCTL . This is used in IOCTL function to change the sample rate.

Enumeration Class	Enum	Description
Audio_Ioctl	Audio_IOCTL_SAMPLE_RATE	Audio interface ioctl to change the sample rate

2.2.4 *Audio_ModuleSel*

The “Audio_ModuleSel” enumerated data type specifies the device type i.e. it is an audio device or an audio codec. This is used in IOCTL function to find to which device the IOCTL is to be sent to (i.e. to the audio device or to the audio codecs).

Enumeration Class	Enum	Description
Audio_ModuleSel	Audio_ModuleSel_AUDIO_DEVICE	Audio Interface module is Audio Device
	Audio_ModuleSel_AUDIO_CODEC	Audio Interface module is Audio Codec

2.3 Dynamic View

2.3.1 Input / Output using Audio driver

In Audio driver, the application can perform IO operation using GIO_read/write() calls (corresponding IOM driver function is audioMdSubmitChan()). The handle to the channel, buffer for data transfer, size of data transfer and timeout for transfer should be provided.

The Audio module receives this information via audioMdSubmitChan. Here some sanity checks on the driver shall be done like valid buffer pointers, etc and then the io request is then routed to the underlying audio device driver which in turn will perform the hardware operations.

2.3.2 Functional Decomposition

The Audio interface driver, has two methods of instantiation, or instance creation – Static and Dynamic. By static instantiation we mean, the invocation of the create call for the module in the configuration file of the application. This is called so because, the creation of the instance is at build time of the application. By dynamic instantiation we mean, the invocation of the create call for the module during runtime of the application.

The two types of instantiation of the module are handled in different ways in the module. The static instantiation of the module is handled in the module script file, and the dynamic instantiation is handled in the C file.

This design concept explained in the sections to follow.

The Aic31 IOM driver needs the global data used by the Aic31 driver to be initialized before the driver can be used. The initialization function for the Aic31 driver is not included in the IOM_Fxns table, which is exported by the Aic31 driver; instead a separate extern is created for use by the SYS/BIOS. The initialization function is responsible for initialization of the following instance specific information

- Initialization of Audio_Instances.
- Sets the “inUse” field of the Audio_module object to FALSE so that the instance can be used by an application which will create it.

2.3.2.1 audioMdBindDev()

The instance MdBindDev function is called when the module is statically/dynamically instantiated. This is the only context available for initialization per instance, when doing dynamic (application C file during run time) or static (application configuration file) instantiation. The return, value of this function represents the extent to which the instance (and hence its resources) were initialized. For example, we could use return value of 0 for complete (successful) initialization done, return value of 1 for failure at the stage of a resource allocation and so on. The audioMdBindDev function is called when the module is

Instance audioMdunBindDev() function, which does a clean up of the driver during instance removal accordingly.

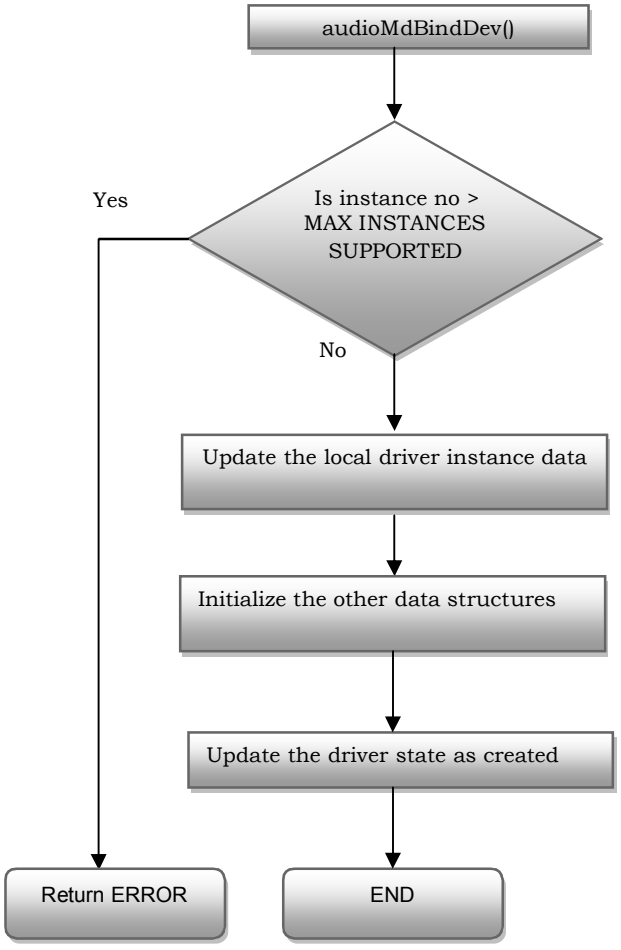

Figure 3 audioMdBindDev() control Flow

2.3.2.2 *audioMdUnBindDev()*

The instance finalize function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance initialization shall be unallocated. After this the instance no more is valid and needs to be reinitialized. Please note that the input parameter for this function is the initialization status returned from the instance) init function. This helps in de-allocation of resources only that were actually allocated during MdBindDev().

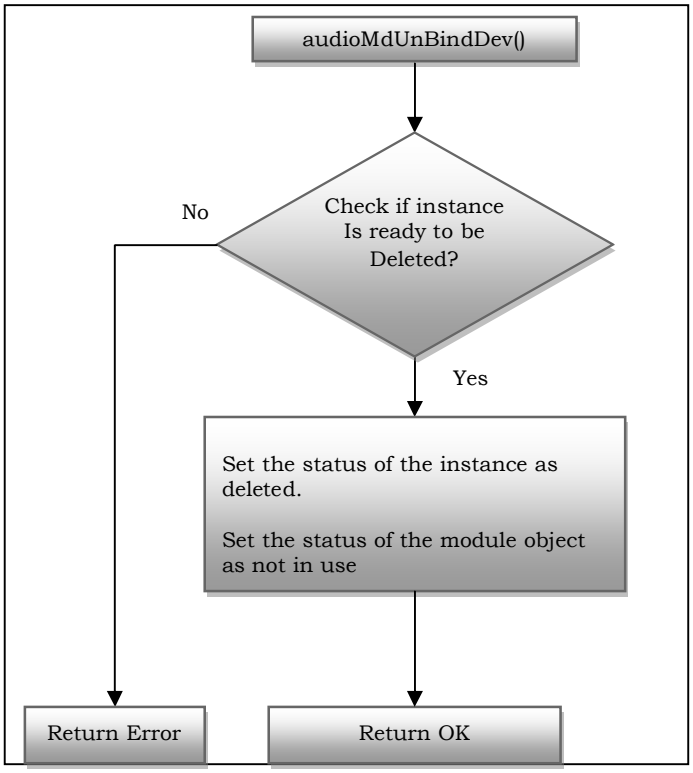

Figure 4 audioMdUnBindDev() control Flow

2.3.2.3 *audioMdCreateChan()*

The audioMdCreateChan function opens a channel which can be used for transmitting or receiving the data. The channel can be opened only in the TX mode or RX mode but not both.

This function is called in response to the “GIO_create” function called by the Application. The IOM driver translates the GIO call to this function.

The open function call tries to open a data transaction channel in the requested mode i.e. (either TX or RX).if it is successful in creating a channel, this API returns a channel handle. This channel Handle is to be used by the channel for all further communications by the channel (e.g. IO request submission, IOCTL submission etc).

Internally the Function performs the following things. It validates the data provided by the application. Once the data is validated the audio driver then loops through all the audio codecs requested to be opened by the application. It uses the name of the audio codec driver to match the driver from the driver table and get the driver handle. If a matching driver is found the audio interface then uses the handle to open the audio codec channel with the data provided by the application in the “chanParams”. if a matching driver is not found or the opening of the channel fails then the audio driver returns an Error code to the stream.

Once all the required codecs are opened successfully then the audio driver then tries to match the audio device name in the driver table. On successful match it uses the driver handle to open the channel to the audio device.

Once the audio device is successfully opened then the audio driver updates the status of the current channel to opened and then returns the handle to the audio channel to the stream which will be used by the application in further communications with the channel.

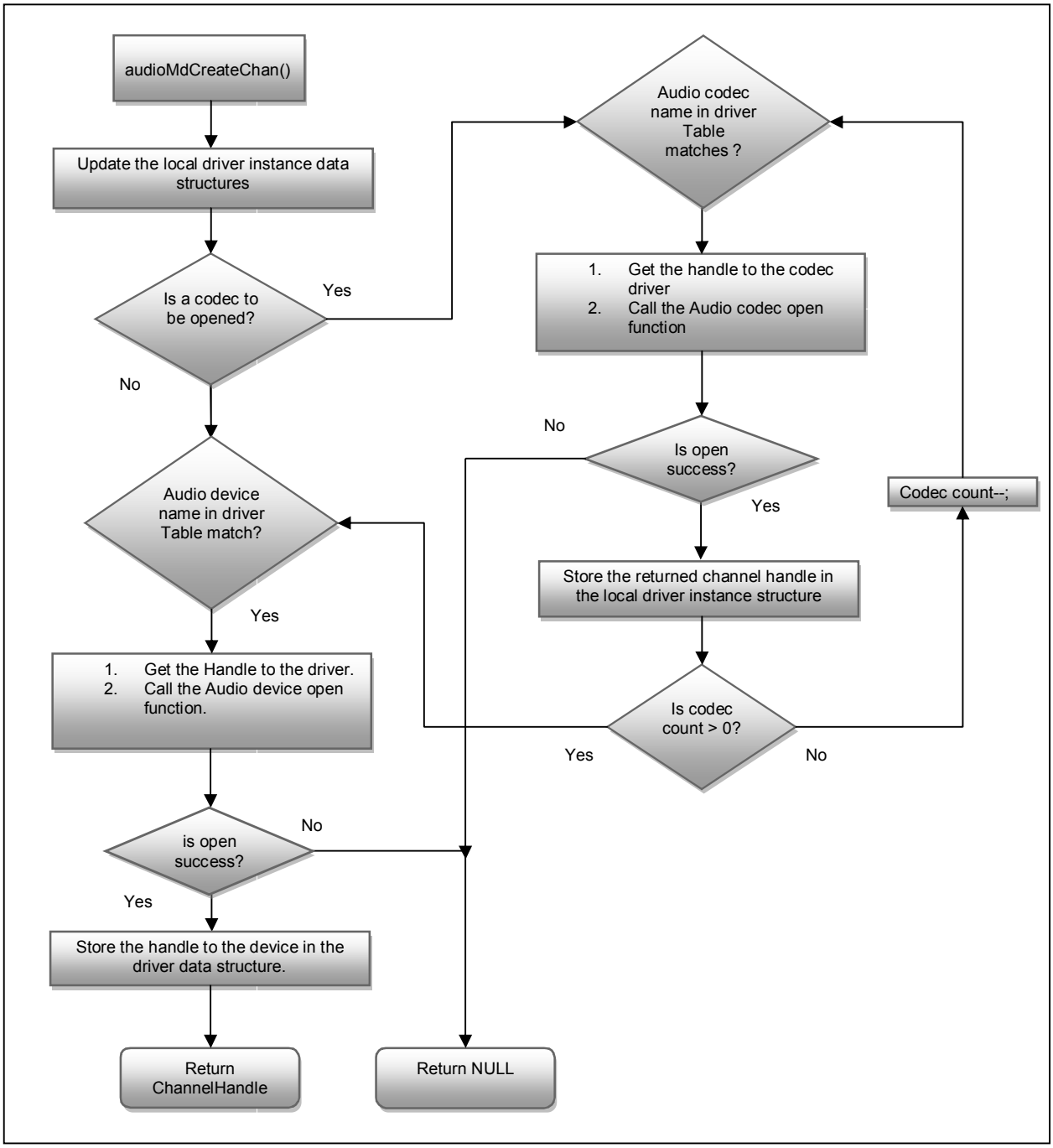

Figure 5 audioMdCreateChan() control flow

2.3.2.4 audioMdDeleteChan()

The application uses a logical channel for transacting with the driver. The application has to open the channel for transaction before using it. Once the application has completed all the transactions and does not want to use the channel for any more transactions, it can close the channel. The application needs to call the function “GIO_delete” so that the channel can be deleted.

Once the channel is closed, it is no longer available for the transactions. If required the channel can be opened by using the “audioMdDeleteChan” function.

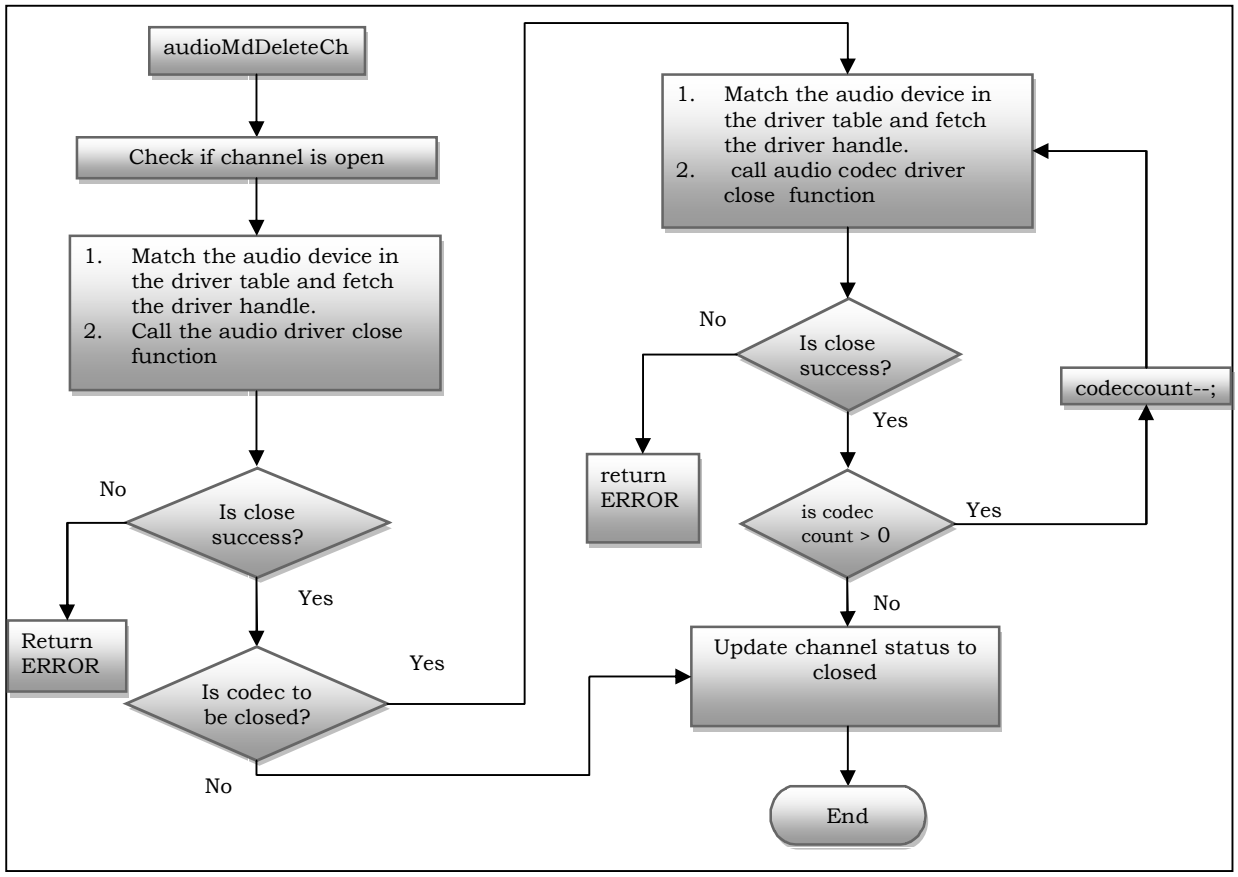

Figure 6 audioMdDeleteChan() control Flow

2.3.2.5 audioMdControlChan()

The audioMdControlChan function provides an interface for the application to pass control commands to the audio device and the audio codecs.

In the audio interface driver, the driver is controlling multiple devices like the audio device (McASP or McBSP) and various number of codecs. The application may be trying to pass the control command to any one of these devices. Hence the control command requires a special structure to be passed along with the command, which will allow the driver to identify the device to which the control command has to be routed.

The “GIO_control” function called by the application translates to the “Audio_control” function in the IOM driver. When a control command is received by the audio interface, it interprets the “Audio_IoctlParam” structure to find the device type to which the command is addressed.

If the device to be controlled is Audio device then the audio driver matches the driver name in the driver table and then retrieves the handle to the driver. Then using the handle, the control function of the audio device driver is called.

If the device to be controlled is an audio codec then the audio driver matches the driver name in the driver table. If the match is not found then an error is raised. If a match is found then the audio driver retrieves the handle to the driver and uses this handle to call the control function of the appropriate audio codec driver.

Note: It should be observed that the user’s IOCTL request completes in the context of calling thread i.e., application thread of control.

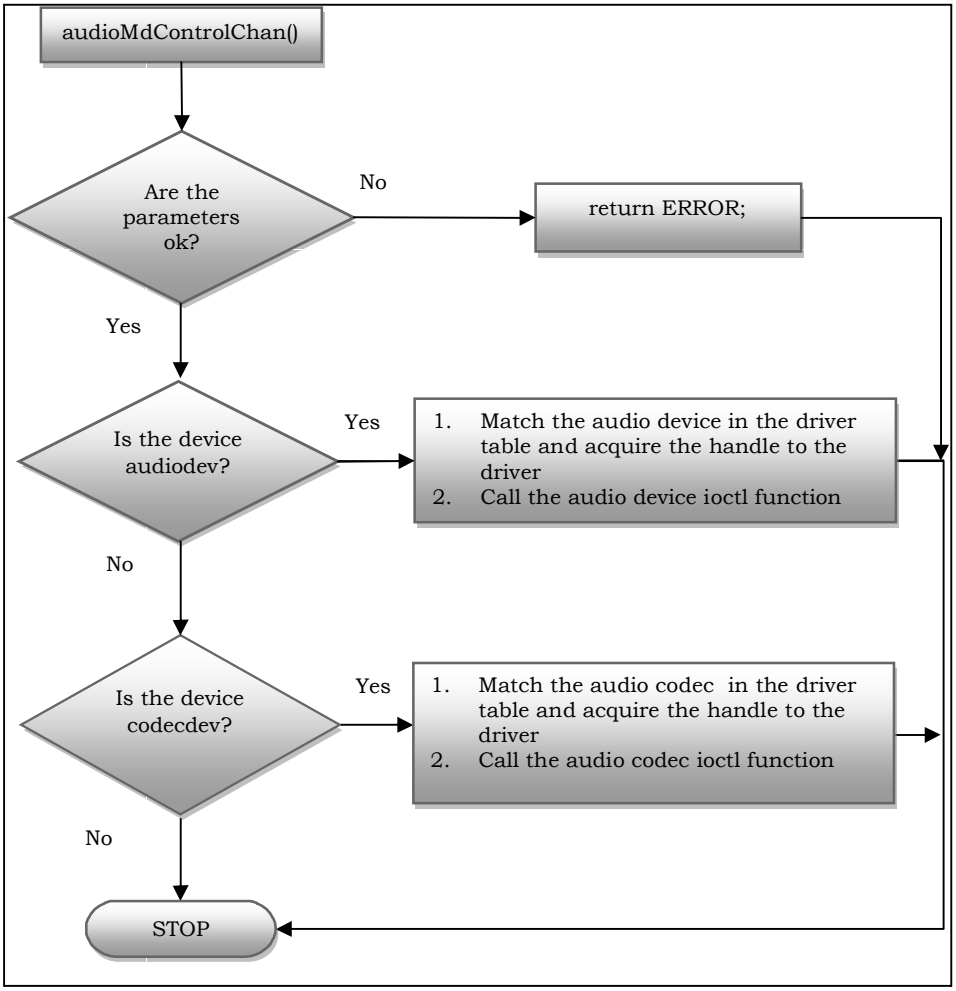

Figure 7 audioMdControlChan() control Flow

2.3.2.6 audioMdSubmitChan()

The application invokes the audioMdSubmitChan to submit a data receive or transmit request to the audio device. Since the audio codec drivers does not support any data transfer request from the user, all the IO transfer request commands will be routed to the audio device only.

Although the audio interface driver by inheritance of the IOM driver module should be an asynchronous driver, the audio driver's mode of working (sync/async) is directly dependent on the underlying drivers it interfaces with. Hence if the underlying audio device driver is working in synchronous mode, then the audio driver will also be in synchronous mode and vice versa.

Note: please note that the audio driver never routes any IO requests to the audio codecs as the audio codec drivers do not handle any data transfer requests from the application.

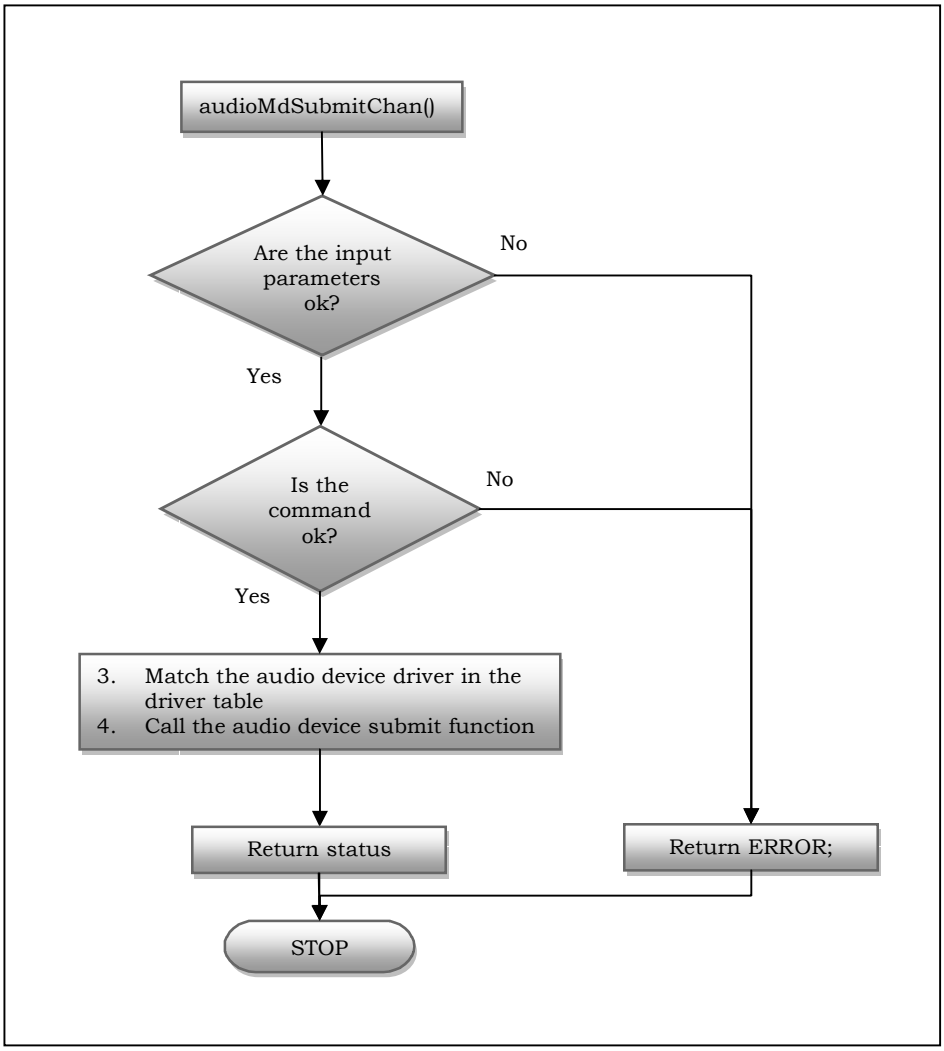

Figure 8 audioMdSubmitChan() control flow

3 IOCTL commands

The application can perform the IOCTLs on any of the devices interfaced by the audio interface driver. Please refer to the individual driver guide for the IOCTLS supported by them. Please refer to the Audio_control () section to send a control command to an underlying device.

S.No	IOCTL Command	Description
1	Audio_IOCTL_SAMPLE_RATE	IOCTL to configure the sample rate for the audio configuration (both the audio device and the codec).