TEXAS INSTRUMENTS REAL WORLD SIGNAL PROCESSING™

SOFTWARE ARCHITECTURE TEMPLATE

BIOSPSP

AIC31 Codec Driver Design Document

Rev No	Author(s)	Revision History	Date	Approval(s)
0.1	Sandeep		13th Mar- 2012	

TABLE OF CONTENTS

List of Tables

List of Figures

1 System Context

The Aic31 codec driver architecture presented in this document is situated in the context of DM8148 SOCs. But the design is relevant to other architectures as well, as the Aic31 codec driver is essentially designed to work on various hardware with no modifications.

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 Disclaimer

This is a design document for the AIC31 driver for the SYS/BIOS operating system. Although the current design document explain the AIC31 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 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 AIC31 5.x.

1.2 Hardware

The Aic31 codec driver is designed for the TLV320AIC3 codec from TI.

1.3 Software

The Aic31 driver discussed here is intended to run in SYS/BIOS™ on the C6748.

1.3.1 Operating Environment and dependencies

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

1.4 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

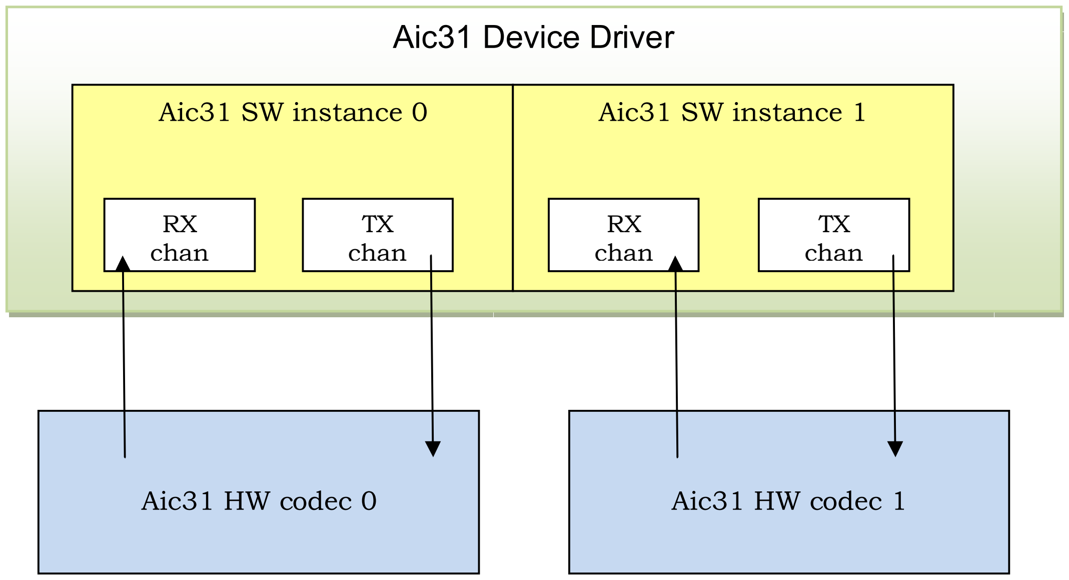


Figure 1 Instance and Channel

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.

1.4.1 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 (Aic31 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, the application can set the variable “paramCheckEnable” to FALSE to disable input parameter checking on every function call.

This instance word usage (Aic31 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 Aic31 module configuration file.

Hence every module shall only support the as many number of instantiations as the number of Aic31 hardware instances on the SOC.

1.4.2 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The Aic31 driver module is an object of IOM Driver class one may need to refer the BIOS IOM Driver model documentation to access the driver in raw mode or could refer the GIO APIs to access the driver through GIO abstraction layer.

1.1.1.1 IOM Driver Interface

The IOM Driver constitutes the Device Driver Manifest to application. This adapts the Driver to SYS/BIOS™. This Aic31 driver is intended to inherit the IOM Driver interfaces. Thus the Aic31 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.

ICodec interface (ICodec.h)

This file is a common file for the audio codec drivers. This file defines constants, enums, data structures and IOCTLs common to all the codec drivers.

Aic31 module specifications file (Aic31.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: if any enums are required.

STRUCTURES: data structure other than those inherited from the ICodec interface.

CONSTANTS: error ids and IOCTLs

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. The Instance object (the driver object) and channel object which contain all the information 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 (Aic31.h) shall be included by the applications, for referring to any of the driver data structures/components. Hence, this file contains everything that should be exposed to the application and also accessed by the driver.

Please note that by nature of the specification of this file, all the variables (independent or part of structure) need to be initialized in this file itself.

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

- Driver initialization function.
- IOM Function pointer table.

Hence theAic31 driver (because of the virtue of its IOM driver model compliance) exposes the following interfaces -

- Aic31_init()
- Aic31_IOMFXNS

The Aic31_init() is a startup function that needs to be called by the user (application) to initialize the module state structure of the Aic31 instance.

- The working of the Aic31 driver will be affected if this function is not called by the application prior to accessing the Aic31 driver APIs.

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

The Aic31 driver exposes IOM function pointer table which contains various APIs provided by the Aic31 driver. The IOM mini-driver implements the following API interfaces to the class driver**:**

Function Name	**Description**
aic31MdBindDev()	Function creates a new instance of the audio interface dynamically/statically
aic31MdCreateChan()	Function to open a channel for data communication
aic31MdSubmitChan()	Function to transfer/receive data using a previously opened channel
aic31MdControlChan()	Function to pass control commands to the audio device and codecs
aic31MdDeleteChan()	Function to close a previously opened channel
aic31MdUnBindDev()	Function to delete the audio interface driver instance.

Table 1 Aic31 Driver interfaces

1.4.3 Design Goals

The following are the key device driver design goals being factored by proposed Aic31 codec driver:

1. Simple unified interface for the application to access the various instances of Aic31 codecs through a set of well defined APIs.
2. Easy addition of new instances of codecs automatically.
3. Application can configure any codec with little knowledge about the underlying codec.

4. Codec driver is essentially platform independent. All the platform dependencies are incorporated in the Aic31Local.h file under the appropriate compiler definitions like number of codec instance, code address etc.

5. Controlling all the Aic31 drivers on a board inspite of different control buses using a single driver.

2 Aic31 Device Driver Software Architecture

This chapter deals with the overall architecture of TI AIC31 codec driver, including the device driver partitioning as well as deployment considerations. We'll first examine the system decomposition into functional units and the interfaces presented by these units. Following this, we'll discuss the deployed driver or the dynamic view of the driver where the driver operational scenarios are presented.

2.1 Static View

2.1.1 Functional Decomposition

The device driver is an abstraction layer between the application and the underlying AIC31 codecs. The device driver can handle multiple instances of the AIC31 codec.

The AIC31 IOM driver will help the application to have a uniform view of the underlying codec and not worry about the control bus configuration and configuring of the codec (unless default configuration needs to be modified)

The Figure 2 shows the Functional composition of the AIC31 codec driver. The codec driver usually can be configured using a control bus.

Two types of control buses are possible

1. SPI bus.

2. I2C bus.

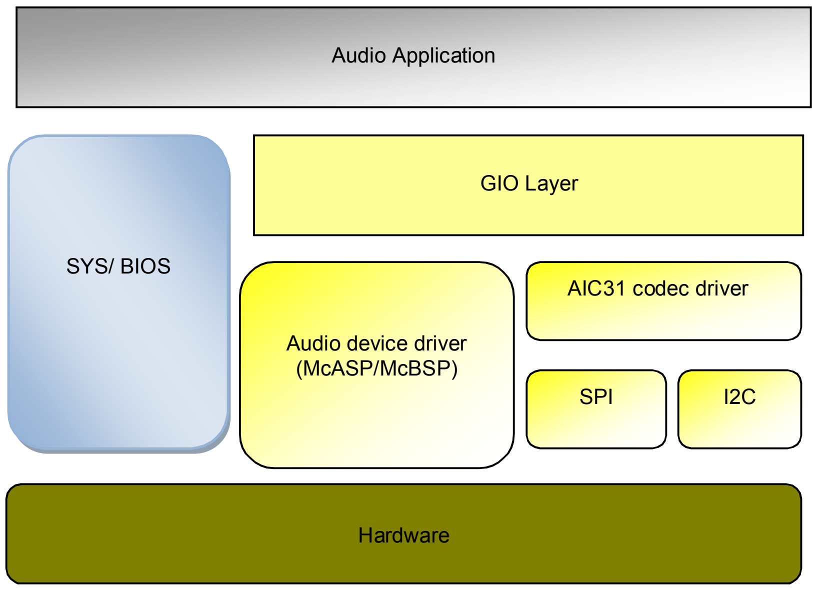


Figure 2 Aic31 driver Functional view

Note:

1. The usage of different control buses is dependent on the hardware support available and layout.

2. The current AIC31 driver only supports I2C control bus. Trying to configure through the SPI bus will return "IOM_ENOTIMPL" i.e. not implemented as error.

The audio interface layer is independent of the OS and the hardware platform. The static configurations are dependent on the Soc for which they are designed.

2.1.2 Aic31driver data structures

This section specifies the data structures used by the Aic31 driver. The IOM driver employs the Instance State (Aic31_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.

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

2.1.2.1 *Instance state(Aic31_Object)*

The instance state comprises of all data structures and variables that logically represent the actual hardware instance on the hardware. 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 codec to use.(refer to platform.xs file)
Aic31_DriverState	devState	State of the instance (Created/Deleted)
Aic31_CodecType	acType	Type of the codec
ICodec_ControlBusType	acControlBusType	Type of the control bus (SPI or I2C)
String	acControlBusName	Name of the control bus driver in the driver table
GIO_Device	* acCtrlBusHandle	Pointer to the driver object of the control bus
UInt32	acCodecId	Slave Device address (Aic31) or CS number (SPi)
Aic31_opMode	acOpMode	Mode of operation of the codec (Master or slaVe)
Aic31_DataType	acSerialDataType	Data format type
UInt32	acSlotWidth	Slot width
Aic31_DataPath	acDataPath	Mode to open the codec
Bool	isRxTxClockIndependent	Whether the RX and TX clocks are independent sections
Channel_Object	ChanObj[2]	TX and RX channel objects
Ptr	hCtrlBus	Handle to the control bus channel
Semaphore_Struct	semObj	Handle to the semaphore used to synchronise the reqad and write operations on the Control bus.
ICodec_Master_Clk_Src	masterClkSrc	Audio codec reference master clock source

Table 2 Instance_state

2.1.2.2 Aic31_Channel_Object

This configuration structure is used to specify the aic31 driver channel object. The driver can be opened in two modes (RX and TX). Hence there will be one channel object structures one for each channel (i.e. two channel object structures for each instance of audio interface).

The structure internally stores the information specific to the channel, the values of which may represent the current state of the channel and also contain information for using the channel.

Element Type	Element Name	Description
Aic31_DriverState	chanStatus	Status of the channel(opened/closed)
ICodec_Channel	channelMode	The channe mode. It could be transmit or recieve
Ptr	devHandle	Pointer to the driver instance object
Uint32	samplingRate	Current sampling rate of the channel
Uint32	chanGain	The current gain of the channel in percentage
Uint32	pllClkDivValue	Audio PLL clcok divider 'N' value

Table 3 Aic31_Channel_Object

2.1.2.3 Device Parameters (Instance parameters)

During module instantiation a set of parameters are required which shall be used to configure the hardware and the driver for that operation mode. This is passed via create call for the module instance. Please note that there are two ways to create an instance (static-using CFG file and dynamic using application c file) and each of these methods have different way of passing devparams. These parameters are preserved in the Aic31_Params structure and are as explained below:

Element Type	Element Name	Description
ICodec_CodecType	acType	Type of the codec(for information only)
UInt32	instNum	Instance number of the audio codec to be used.
ICodec_ControlBusType	acControlBusType	Control bus to be used for the configuring of the codec
String	acControlBusName	Name of the control bus driver in the driver table
ICodec_OpMode	acOpMode	Mode of operation of the codec (Master or slaVe)
ICodec_DataType	acSerialDataType	Data format type
UInt32	acSlotWidth	Slot width
ICodec_DataPath	acDataPath	Mode to open the codec
Bool	isRxTxClockIndependent	Whether the RX and TX clocks are independent sections
ICodec_Master_Clk_Src	masterClkSrc	Audio codec reference master clock source
Uint32	pllClkDivValue	Audio PLL clock divider ‘N’ value

Table 4 Aic31_Params

2.1.2.4 ChannelConfig(Channel parameters)

Every channel opened to the device may need some setting by the user. These parameters may be passed through to the driver module by using this structure during channel opening.

Element Type	Element Name	Description
UInt32	samplingRate	Sampling rate to be set for the codec
Uint32	chanGain	The intial gain for the channel
UInt32	bitClockFreq	Bit Clock speed
UInt32	numSlots	Number of TDM slots
Uint32	pllClkDivValue	Audio PLL clcok divider value (2 to 17)

Table 5 ChannelConfig

2.2 Audio codec driver data types

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

2.2.1 ICodec_Channel

The "ICodec_Channel" specifies the type of audio codec channel being configured i.e. TX path or the RX path.

Enumeration Class	Enum	Description
ICodec_Channel	ICodec _Channel_INPUT	Audio codec RX channel.
	ICodec _Channel_OUTPUT	Audio codec TX channel.
	ICodec _Channel_MAX	Delimiter enum

Table 6 ICodec_Channel

2.2.2 Aic31_DataType

The “ICodec_DataType” enumerated data type specifies the data transfer mode to be used by the codec.

Enumeration Class	Enum	Description
ICodec_DataType	ICodec_DataType_LEFTJ	Audio data transfer mode is left justified
	ICodec_DataType_RIGHTJ	Audio data transfer mode is right justified
	ICodec_DataType_I2S	Audio data transfer mode is I2S
	ICodec_DataType_DSP	Audio data transfer mode is DSP
	ICodec_DataType_TDM	Audio data transfer mode is TDM

Table 7 ICodec_DataType

2.2.3 ICodec_Master_Clk_Src

This enum specifies the Audio codec clock source. This can be used to derive reference clock to the code either from teh MCLK or from BCLK.

Enumeration Class	Enum	Description
ICodec_Master_Clk_Src	ICodec_MCLK	Codec reference clock is MCLK
	ICodec_BCLK	Codec reference clock is BCLK

2.2.4 ICodec_OpMode

The “ICodec_OpMode” enumerated data type specifies the operational mode of the audio codec i.e. whether the codec is operating in master mode or slave mode.

Enumeration Class	Enum	Description
ICodec_OpMode	ICodec_OpMode_MASTER	Codec works in master mode
	ICodec_ OpMode_SLAVE	Codec works in slave mode

Table 8 ICodec_OpMode

2.2.5 ICodec_ControlBusType

The "ICodec_ControlBusType" enumerated data type specifies the control bus that will be needed to configure the Aic31 codec.

Enumeration Class	Enum	Description
ICodec_ControlBusType	ICodec_ControlBusType_I2C	Codec communicates using the Aic31 bus
	ICodec_ ControlBusType_SPI	Codec communicates using the Spi bus.
	ICodec_ControlBusType_UNKNOWN	Delimiter Enum

Table 9 ICodec_ControlBusType

2.3 Dynamic View

2.3.1 The Execution Threads

The Aic31 IOM Driver module involves following execution threads:

Application thread: Creation of channel, Control of channel and deletion of channel.

Interrupt context: No interrupt context

Control bus call back context: The callback from control bus driver on the completion of the read or write request (this would actually be in the CPU interrupt context).

2.3.2 Input / output using Aic31 driver

Aic31 driver does not handle any IO transfer requests.

2.4 Functional Decomposition

The Aic31 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. 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

- Obtain the Base address for the instance.
- Initialization of Aic31_Instances.
- Sets the "inUse" field of the Aic31 instance module object to FALSE so that the instance can be used by an application which will create it.

Please refer to the figure below for the typical control flow during the initialization of the driver.

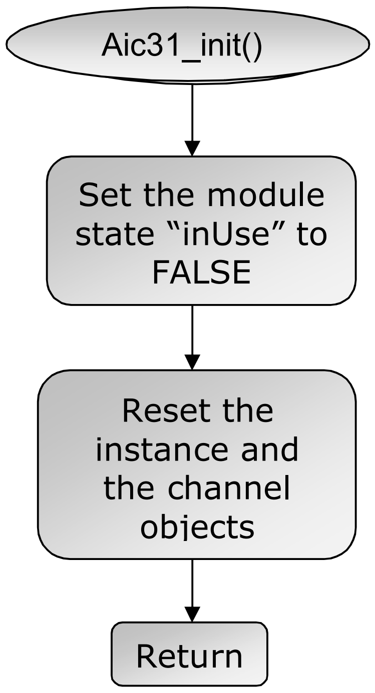

Aic31_init()
Set the module state "inUse" to FALSE
Reset the instance and the channel objects
Return

Figure 3 Aic31_init() control flow

2.4.1.1 aic31MdBindDev()

The instance MdBindDev function is called when the module is staticatically/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.

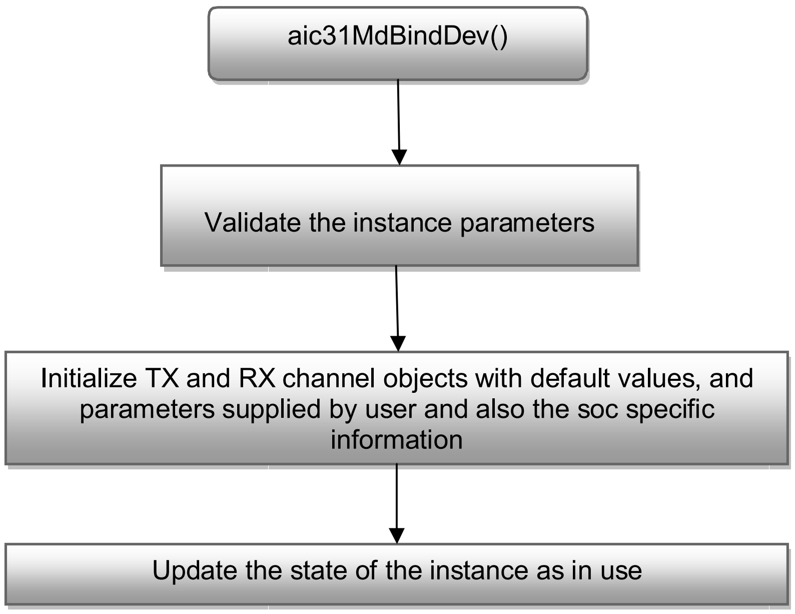

aic31MdBindDev()
Validate the instance parameters
Initialize TX and RX channel objects with default values, and parameters supplied by user and also the soc specific information
Update the state of the instance as in use

Figure 4 aic31MdBindDev() control flow

2.4.1.2 aic31MdUnBindDev()

The aic31MdUnBindDev functions are called by the IOM driver model during the deletion of a driver instance.

The instance MdUnBindDev 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.

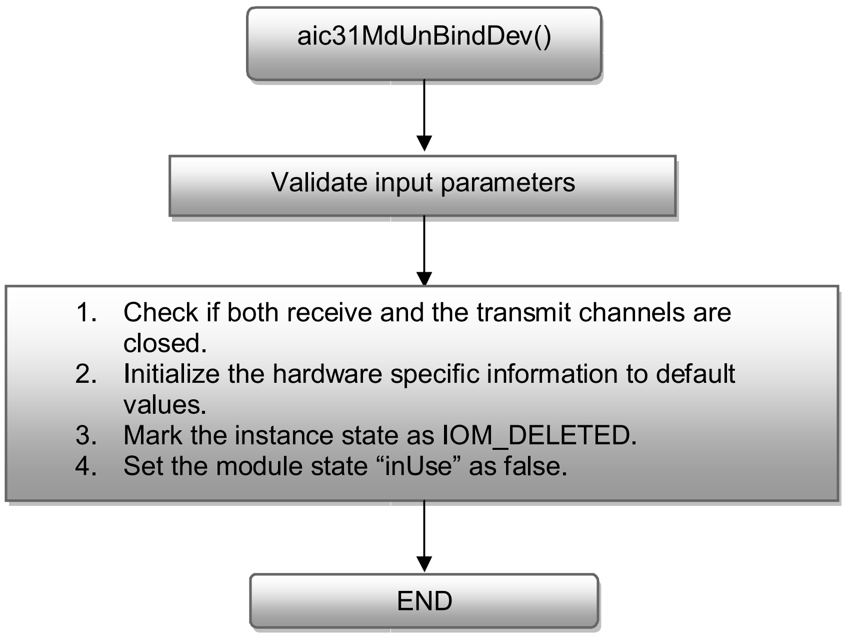

aic31MdUnBindDev()
Validate input parameters
1. Check if both receive and the transmit channels are closed.
2. Initialize the hardware specific information to default values.
3. Mark the instance state as IOM_DELETED.
4. Set the module state "inUse" as false.
END

Figure 5 aic31MdUnBindDev() control flow

2.4.1.3 aic31MdCreateChan()

The aic31MdCreateChan function is called in response to the "GIO_create" function called by the application. This function configures the ADC and DAC section of the codec depending on the channel specified.

This function checks the current state of the channel. If the channel is not under use then the channel is prepared for allocation else an error is raised. The application needs to specify the mode of creation of the channel. The application also supplies the channel parameters required for the creation of the channel.

This function returns a pointer which will be the handle to the channel for all the subsequent operations. In case of failure a NULL value is returned. This handle is required for control requests to be sent to the driver from the application.

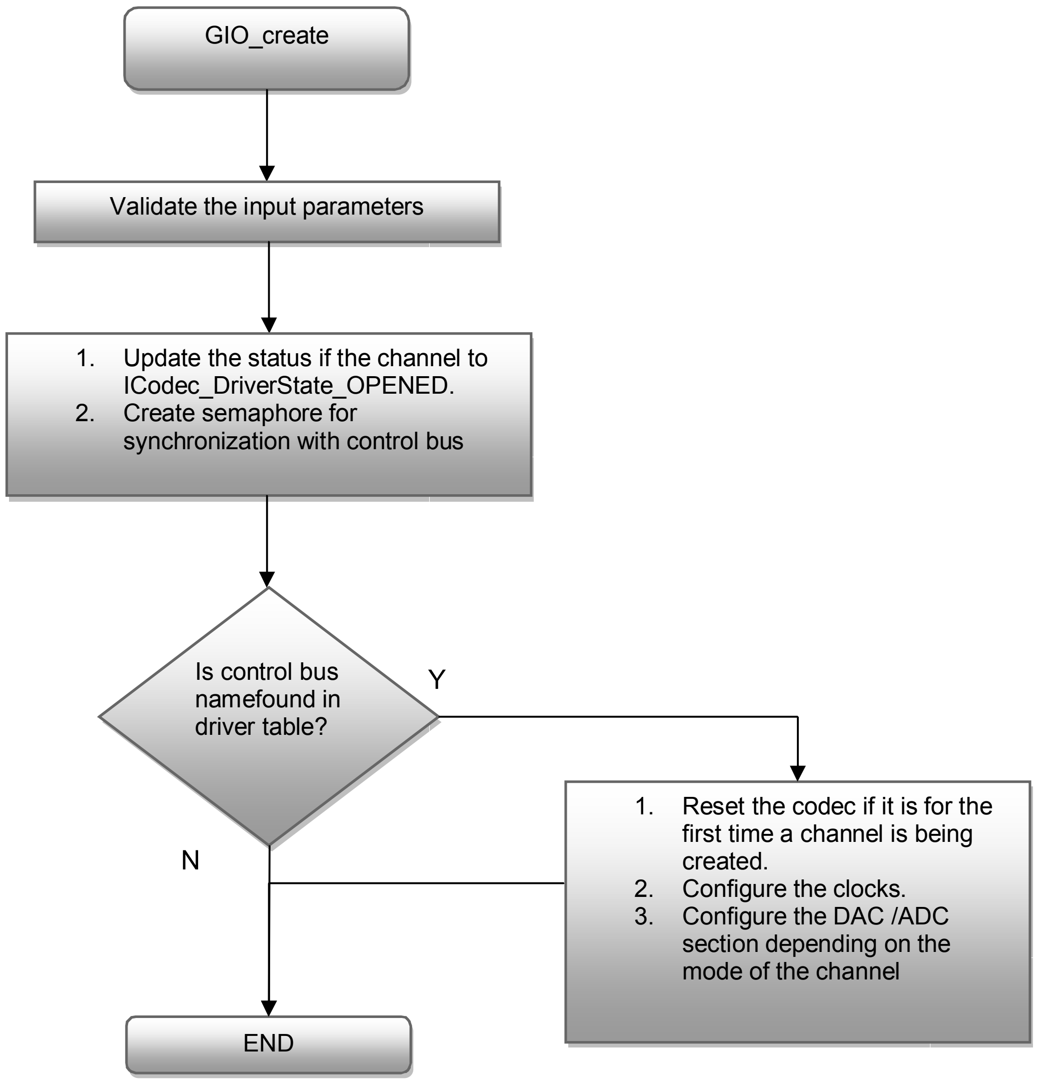

GIO_create
Validate the input parameters
1. Update the status if the channel to ICodec_DriverState_OPENED.
2. Create semaphore for synchronization with control bus
Is control bus namefound in driver table?
Y
N
1. Reset the codec if it is for the first time a channel is being created.
2. Configure the clocks.
3. Configure the DAC /ADC section depending on the mode of the channel
END

Figure 6 aic31MdCreateChan() control flow

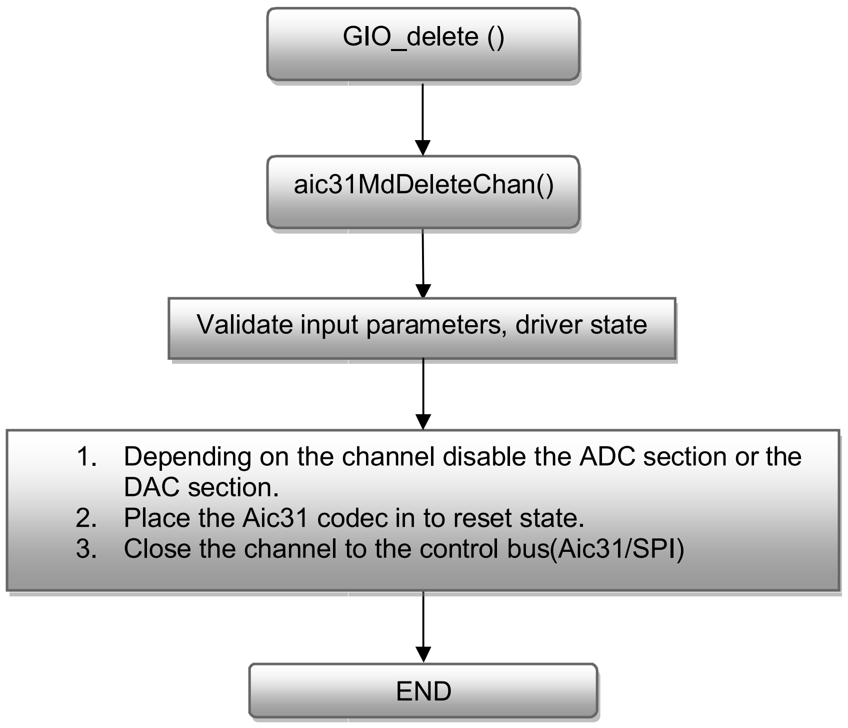
2.4.1.4 *aic31MdDeleteChan()*

Once the application has finished with all the transaction with the Aic31 device it can close the channel. After the channel is closed it is no longer available for further transactions and will have to be opened again if required to be used.

The Application will call the "GIO_delete" with the handle to the appropriate channel to close. This will invoke the aic31MdDeleteChan function provided by the Aic31 IOM Driver. The close function deallocates all the resources allocated to Aic31 device during the opening of the channel.

It marks the channel as in closed state .After this channel can be reused by any application again

Figure 7 aic31MdDeleteChan() control Flow


GIO_delete ()
aic31MdDeleteChan()
Validate input parameters, driver state
1. Depending on the channel disable the ADC section or the DAC section.
2. Place the Aic31 codec in to reset state.
3. Close the channel to the control bus(Aic31/SPI)
END

2.4.1.5 aic31MdControlChan()

The Aic31 driver provides the applications the interface to control the device using various IOCTL commands. The list of various IOCTL commands supported by the Driver is listed at the end of the document. The application can use the control commands to control the functionality of the Aic31 driver by issuing the “GIO_control” command to the IOM driver.

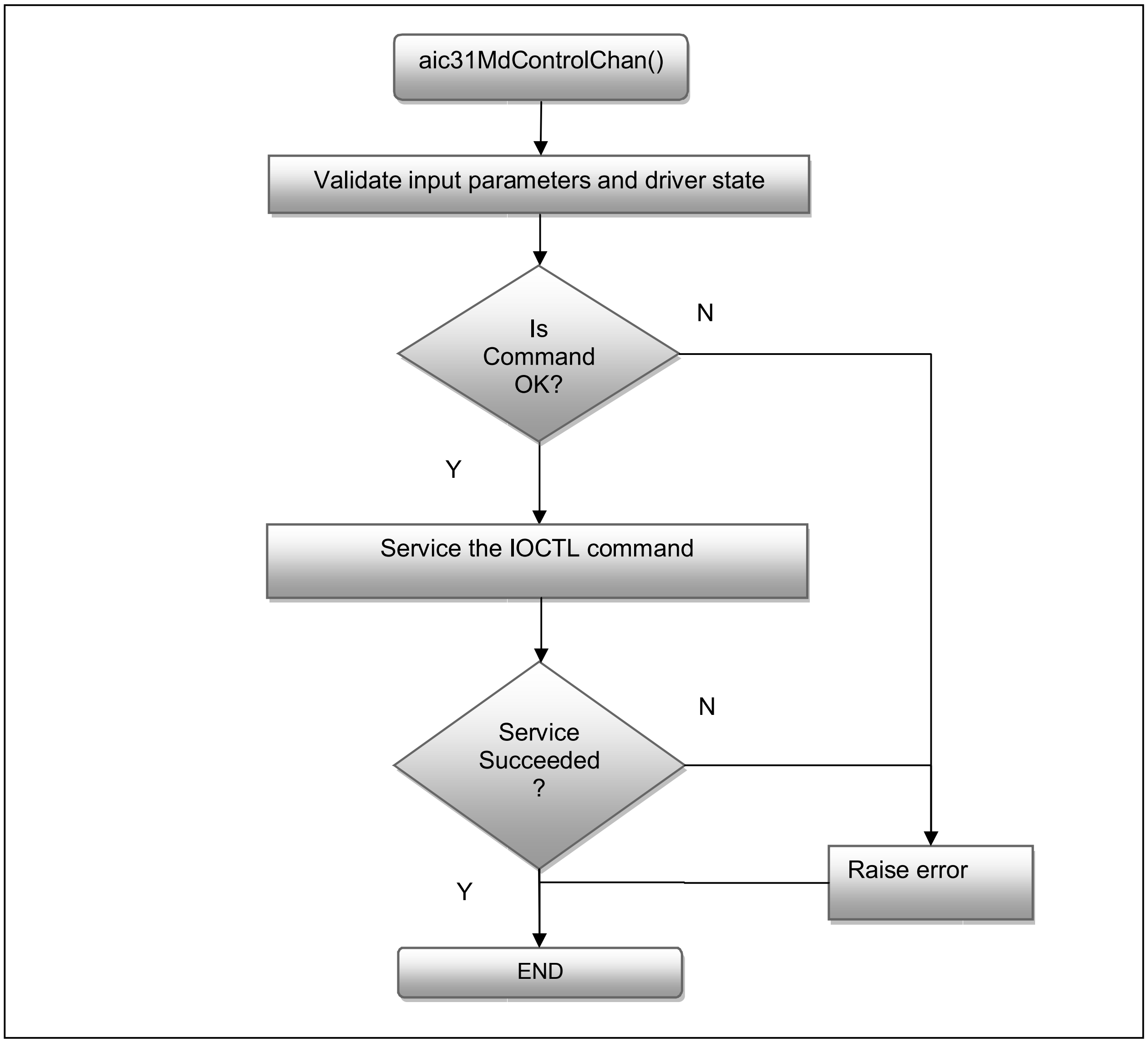


Figure 8 aic31MdControlChan() control flow

2.4.1.6 aic31MdSubmitChan()

The Aic31 device does not handle any data requests from the application directly. Hence the Aic31 driver does not support any IO request handling.

Note:

In order to be compatible with the IOM driver model the Aic31 driver will have an aic31MdSubmitChan function implemented. But, any IO transfer request from the application will cause an Error to be raised by the driver and return an IOM_ERROR to the GIO layer.

3 APPENDIX A – IOCTL commands

The application can perform the following IOCTL on the channel. All the IOCTL commands shall be sent through the individual channels.

S.No	IOCTL Command	Description
1	Aic31_AC_IOCTL_MUTE_ON	Set the codec Mute.
2	Aic31_AC_IOCTL_MUTE_OFF	Un-mute the codec
3	Aic31_AC_IOCTL_SET_VOLUME	Set the volume information for the codec
4	Aic31_AC_IOCTL_SET_LOOPBACK	Set the codec in to loop back mode
5	Aic31_AC_IOCTL_SET_SAMPLERATE	Set the sampling rate for the codec
6	Aic31_AC_IOCTL_REG_WRITE	Write to a specific codec register.
7	Aic31_AC_IOCTL_REG_READ	Read a specific codec register
8	Aic31_AC_IOCTL_REG_WRITE_MULTIPLE	Write to multiple codec registers
9	Aic31_AC_IOCTL_REG_READ_MULTIPLE	Read from multiple codec registers
10	Aic31_AC_IOCTL_SELECT_OUTPUT_SOURCE	Selects the output destination from HPOUT or line out or Both.
11	Aic31_AC_IOCTL_SELECT_INPUT_SOURCE	Selects the input source from Mic–in or Line in.
12	Aic31_AC_IOCTL_GET_CODEC_INFO	IOCTL to get the information about the codec instance.