DSP/BIOS AIC31 codec driver

Architecture/Design Document

July 8, 2009 ***Platform Software Group***

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address:
Texas Instruments
Post Office Box 655303, Dallas, Texas 75265

About This Document

This document discusses the TI device driver architecture for Aic31 device Driver. The target audience includes device driver developers from TI as well as consumers of the driver.

Trademarks

The TI logo design is a trademark of Texas Instruments Incorporated. All other brand and product names may be trademarks of their respective companies.

This document contains proprietary information of Texas Instruments. The information contained herein is not to be used by or disclosed to third parties without the express written permission of an officer of Texas Instruments Incorporated.

Related Documents

- ❑ DSP BIOS user guide
- ❑ TLV320AIC31IRHBRG4_3960631.pdf

Notations

Terms and Abbreviations

Term	Description
API	Application Programming Interface
Audio Codec	Audio codec devices such as TI's AIC12 or AIC33.
Control Bus	Control bus connected to the audio codec device such as I2C or SPI.
Handle	A reference to the audio codec channel.
Instance Number	An identification of the instance of the audio codec driver.

Revision History

Document Version	Author(s)	Date	Comments
0.1	Imtiaz SMA	September 26 2008	Created the document
0.2	Imtiaz SMA	October 23 2008	Fixed the review comments given for the document.
0.3	Imtiaz SMA	December 8 2008	Updated the new IOCTL commands implemented for the driver
0.4	Imtiaz SMA	January 21, 2009	Updated the Sections for the IOM driver and IDriver contrast..

Table of Contents

Tables vi

1 System Context 1

1.1 Disclaimer 1

1.2 IOM driver Vs IDriver 2

1.3 Hardware 3

1.4 Software 3

1.4.1 Operating Environment and dependencies 3

1.5 Design Philosophy 3

1.5.1 The Module and Instance Concept 4

1.5.2 Component Interfaces 5

1.5.3 Design Goals 7

2 Aic31 Device Driver Software Architecture 8

2.1 Static View 8

2.1.1 Functional Decomposition 8

2.1.2 Aic31driver data structures 10

2.2 Audio codec driver data types 14

2.2.1 Aic31_Channel 14

2.2.2 Aic31_DataType 14

2.2.3 Aic31_OpMode 15

2.2.4 Aic31_ControlBusType 15

2.3 Dynamic View 16

2.3.1 The Execution Threads 16

2.3.2 Input / output using Aic31 driver 16

2.3 Functional Decomposition 17

3 APPENDIX A – IOCTL commands 25

Tables

Table 1 Aic31 Driver interfaces 7
Table 2 Instance_state 11
Table 3 Aic31_Channel_Object 12
Table 4 Dev_Params 13
Table 5 ChannelConfig 13
Table 6 Aic31_Channel 14
Table 7 Aic31_DataType 14
Table 8 Aic31_OpMode 15
Table 9 Aic31_ControlBusType 15

Figure 1 Instance and Channel 4
Figure 2 Aic31 driver Functional view 9
Figure 3 Instance$static$init( ) control flow 18
Figure 4 Aic31_Instance_Init() flow diagram 19
Figure 6 Aic31_Instance_finalize () control flow 20
Figure 6 Aic31_open () control flow 21
Figure 7 Aic31_Close control Flow 22
Figure 8 Aic31_Control Control flow 23

1 System Context

The Aic31 codec driver architecture presented in this document is situated in the context of DA830 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 DSP/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 BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IDriver 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 AIC31 5.x.

1.2 IOM driver Vs IDriver

The following are the main difference between the BIOS 5.x and BIOS 6.x driver. Please refer to the reference documents for more details in the IOM driver model.

1. All the references to the stream module should be treated as references to a module that provides data streaming. In BIOS 5.x the equivalent modules are SIO and GIO.

2. This document refers to the IDriver model supported by the BIOS 6.x.All the references to the IDriver should be assumed to be equivalent to the IOM driver model.

3. The BIOS 6.x driver uses a module specifications file (*.xdc) for the declaration of the enumerations, structures and various constants required by the driver. The equivalent of this xdc file is the header file XXX.h and the XXXLocal.h.

   Note: The XXXLocal.h file contains all the declaration specified in the “internal” section of the corresponding xdc file.

4. In BIOS 6.x creation of static driver instances follow a different flow and cause functions in module script files to run during build time. In BIOS 5.x creation of driver (for both static and dynamic instances) result in the execution of the mdBindDev function at runtime. Therefore any references to module script files (*.xs files) can be ignored for IOM drivers.

5. The XXX_Module_startup function referenced in this document can be ignored for IOM drivers.

6. IOM drivers have an XXX_init function which needs to be called by the application once per driver. This XXX_init function initializes the driver data structures. This application needs to call this function in the application initialization functions which are usually supplied in the tci file.

7. The functionality and behavior of the functions is the same for both driver models. The mapping of IDriver functions to IOM driver functions are as follows:

IDriver	IOM driver
XXX_Instance_init	mdBindDev
XXX_Instance_finalize	mdUnbindDev
XXX_open	mdCreateChan
XXX_close	mdDeleteChan
XXX_control	mdControlChan
XXX_submit	mdSubmitChan

8. All the references to module wide config parameters in the IDriver model map to macro definitions and preprocessor directives (#define and #ifdef etc) for the IOM drivers. e.g.

    a. XXX_edmaEnable in IDriver maps to –D XXX_EDMA_ENABLE for IOM driver.

    b. XXX_FIFO_SIZE in IDriver maps to #define FIFO_SIZE in IOM driver header file

9. In BIOS 6.x a cfg file is used for configuring the BIOS and driver options whereas in the BIOS 5.x the “tcf” and “tci” files are used for configuring the options.

10. In BIOS 5.x driver support for multiple devices is implemented as follows. A chip specific compiler define is required by the driver source files (-DCHIP_OMAPL138). Based on the include a chip specific header file (e.g soc_OMAPL138) which contains chip specific defines is used by the driver. In order to support a new chip, a new soc_XXX header file is required and driver sources files need to be changed in places where the chip specific define is used.

1.3 Hardware

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

1.4 Software

The Aic31 driver discussed here is intended to run in DSP/BIOS™ V6.10 on the OMAPL138 DSP.

1.4.1 Operating Environment and dependencies

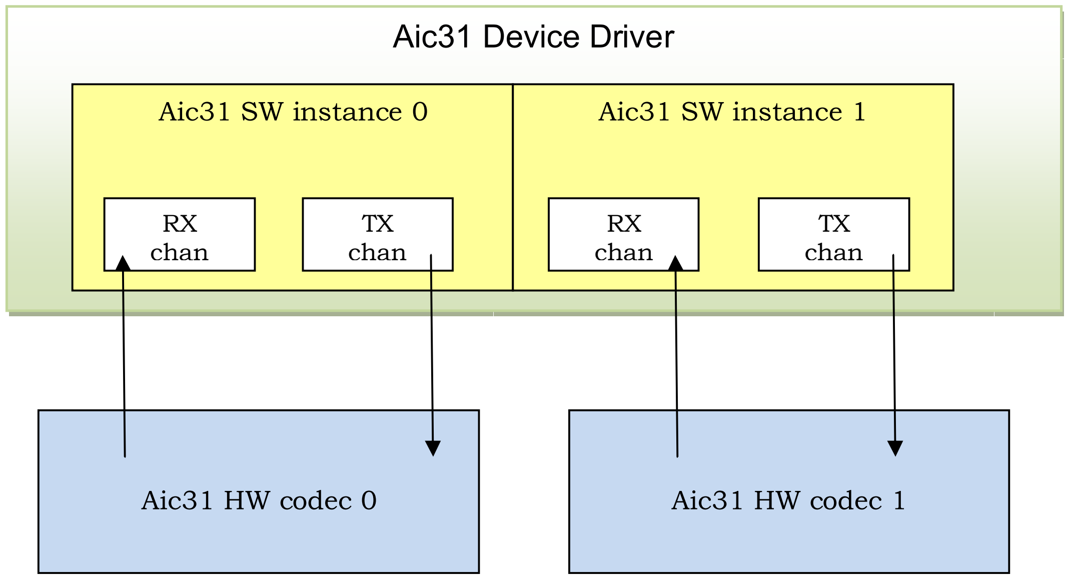
Details about the tools and their versions that the driver is compatible with, can be found in the system Release Notes.

1.5 Design Philosophy

This device driver is written in conformance to the DSP/BIOS IDriver 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 DSP BIOS based AIC31 IDriver intends to make the configuration of all AIC31 codecs simple. Using an Aic31 IDriver 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.

1.5.1 The Module and Instance Concept

The IDriver model, conforming to the XDC framework, 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.

Figure 1 Instance and Channel

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.

1.5.2 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 IDriver class one may need to refer the IDriver documentation to access the driver in raw mode or could refer the stream APIs to access the driver through stream abstraction. The structures and config params used would be documented in CDOC format as part of this driver development.

1.1.1.1 IDriver Interface

The Aic31 driver is intended to be an XDC module and this module would inherit the ICodec interface which in turn has inherited the IDriver interfaces. Thus the Aic31 driver module becomes an object of IDriver class. Please note that the terms “Module” and “Driver or IDriver” would be used in this document interchangeably.

As per xdc specification, a module should feature a xdc file, xs file and source file as a minimum.

ICodec interface (ICodec.xdc)

This XDC file is a common file for the audio codec drivers. It inherits the IDriver class thereby allowing all the drivers inheriting ICodec class to be of the IDriver class compatibile. This XDC file defines constants, enums, data structures and IOCTLs common to all the codec drivers.

Aic31 module specifications file (Aic31.xdc)

The XDC 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 XDC framework translates this into the driver header file (AIc31.h) and this header file shall be included by the applications, for referring to any of the driver data structures/components. Hence, XDC 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 the xdc file, all the variables (independent or part of structure) need to be initialized in xdc file itself.

Aic31 module script file (Aic31.xs)

The script file is the place where static instantiations and references to module usage are handled. This script file is invoked when the application compiles and refers to the Aic31 module/driver. The Aic31.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg file for the Aic31 module, the module use function in the AIc31.xs file is used to initialize the hardware instance specific details of the codec (device address on the control bus) depending on the SOC type. This data is stored for further use during instantiation. This gives the flexibility to design, to handle multiple SOC with single c-code base (as long as the IP does not deviate).

2. Handling static instantiation of the Aic31 instance

   When the Aic31 instance is instantiated statically in the application cfg file, (please note that dynamic instantiation is also possible from a C file) the instance static init function is called. If a particular instance is configured with set of instance parameters (fron CFG file) they are used here to configure the instance state. The instance state is populated based on the instance number, like the default state of the driver, channel objects etc.

The Aic31 IDriver module implements the following interfaces

Function Name	**Description**
Aic31_Instance_init	Function to create a new instance of the audio interface dynamically
Aic31_open	Function to open a channel for data communication
Aic31_submit	Function to transfer/receive data using a previously opened channel
Aic31_control	Function to pass control commands to the audio device and codecs
Aic31_close	Function to close a previously opened channel
Aic31_Instance_finalize	Function to delete the audio interface driver instance.

Table 1 Aic31 Driver interfaces

1.5.3 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 to the platform.xs file.
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.

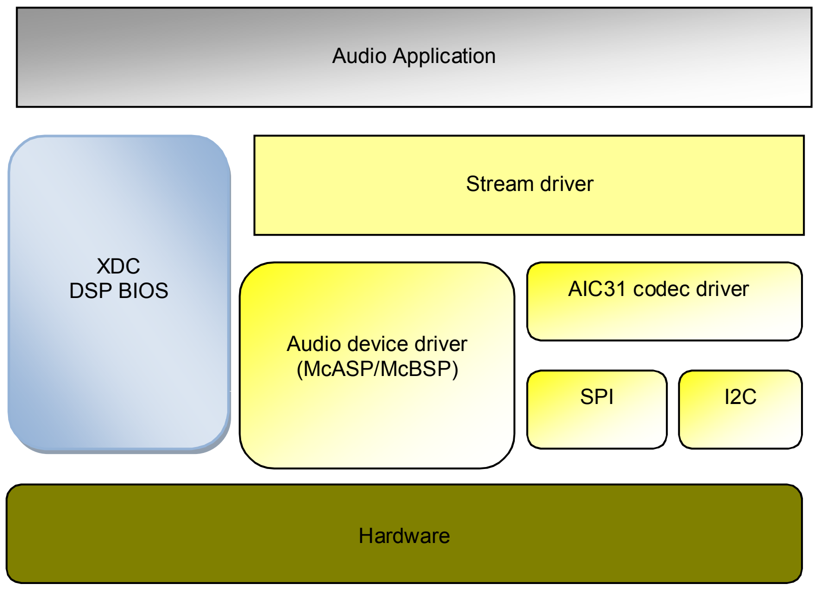
2.1 Static View

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 IDriver 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)

Figure 2 Aic31 driver Functional view



The above Figure 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.

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 "DriverTypes_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 IDriver 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 IDriver 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
String	acControlBusName	Name of the control bus driver in the driver table
UInt32	acCodecId	Slave Device address (I2c) 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.Handle	semHandle	Handle to the semaphore used to synchronise the reqad and write operations on the Control bus.

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)
UInt8	instNum	Instance number of the codec to use.(refer to platform.xs file)
Ptr	devHandle	Pointer to the driver instance object

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. For further information of these methods please refer xdc documentation. These parameters are preserved in the DevParams structure and are as explained below:

Element Type	Element Name	Description
AIc31_CodecType	acType	Type of the codec(for information only)
Aic31_Channel	channelMode	Current operational mode of the channel(RX/TX)
Aic31_CodecType	acType	Type of the codec
String	acControlBusName	Name of the control bus driver in the driver table
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

Table 4 Dev_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	codecId	Slave address of the codec or the chip select value
UInt32	numSlots	Number of TDM slots

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 Aic31_Channel

The “Aic31_Channel” specifies the type of audio codec channel being configured i.e. TX path or the RX path.

Enumeration Class	Enum	Description
Aic31_Channel	Aic31_Channel_INPUT	Audio codec RX channel.
	Aic31_Channel_OUTPUT	Audio codec TX channel.
	Aic31_Channel_MAX	Delimiter enum

Table 6 Aic31_Channel

2.2.2 Aic31_DataType

The “Aic31_DataType” enumerated data type specifies the data transfer mode to be used by the codec.

Enumeration Class	Enum	Description
Aic31_DataType	Aic31_DataType_LEFTJ	Audio data transfer mode is left justified
	Aic31_DataType_RIGHTJ	Audio data transfer mode is right justified
	Aic31_DataType_I2S	Audio data transfer mode is I2S
	Aic31_DataType_DSP	Audio data transfer mode is DSP
	Aic31_DataType_TDM	Audio data transfer mode is TDM

Table 7 Aic31_DataType

2.2.3 *Aic31_OpMode*

The “Aic31_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
Aic31_OpMode	Aic31_OpMode_MASTER	Codec works in master mode
	Aic31_ OpMode_SLAVE	Codec works in slave mode

Table 8 Aic31_OpMode

2.2.4 *Aic31_ControlBusType*

The “Aic31_ControlBusType” enumerated data type specifies the control bus that will be needed to configure the Aic31 codec.

Enumeration Class	Enum	Description
Aic31_ControlBusType	Aic31_ ControlBusType_I2C	Codec communicates using the I2c bus
	Aic31_ ControlBusType_SPI	Codec communicates using the Spi bus.

Table 9 Aic31_ControlBusType

2.3 Dynamic View

2.3.1 The Execution Threads

The Aic31 IDriver 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.

2.3 Functional Decomposition

The Aic31 driver, as seen in the RTSC framework, 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.

2.3.1.1 *module$use() of XS file*

One important feature in the RTSC framework design of this package is that we have designed to have a platform.xs capsule file, which will have the SoC specific information(for more than one SOC), e.g. codec device address for multiple codecs. This move is adopted to keep the driver C code free from compiler switches and everything of this sort in the form of either configuration variables for the module or loading the platform specific data from the soc.xs capsule. This loading of data is done in the module use function.

The module shall have an array of device instance configuration structures (deviceInstInfo), which shall contain the device address details. The length of this array is defined by the array member of this instance in the soc.xs file.

The default device information is populated for all the instance numbers in this function since this information is needed to later prepare the instances in instance static init and the instance init functions.

2.3.1.2 *instance$static$init of XS file*

This function context is the where the instance statically created is initialized. Please note that the instance params provided by the applications (from the CFG file) would override the default value of those parameters from XDC file.

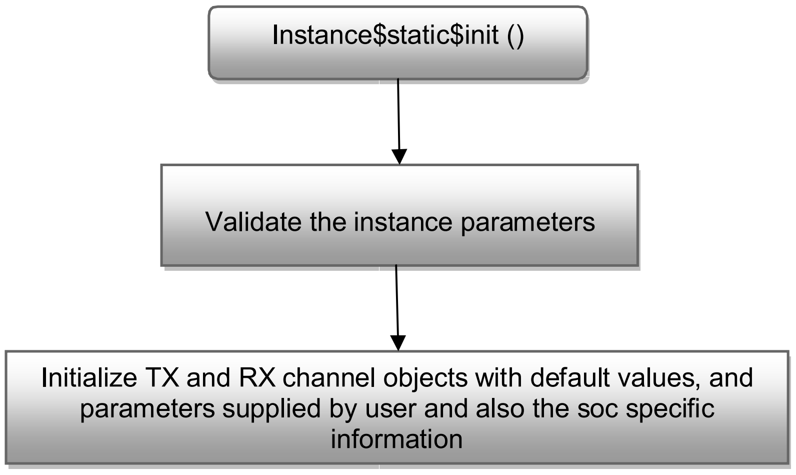

Instance$static$init ()
Validate the instance parameters
Initialize TX and RX channel objects with default values, and parameters supplied by user and also the soc specific information

Figure 3 Instance$static$init( ) control flow

2.3.1.1 Aic31_Instance_init()

The instance init function is called when the module is dynamically instantiated. This is the only context available for initialization per instance, when doing dynamic (application C file during run time) instantiation. Hence, this function should be including all the initialization done in the instance static init in the module script file and the module startup function if it is used. 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. This return value is preserved by the RTSC/BIOS framework and passed to the Instance_finalize function, which does a cleanup of the driver during instance removal accordingly.

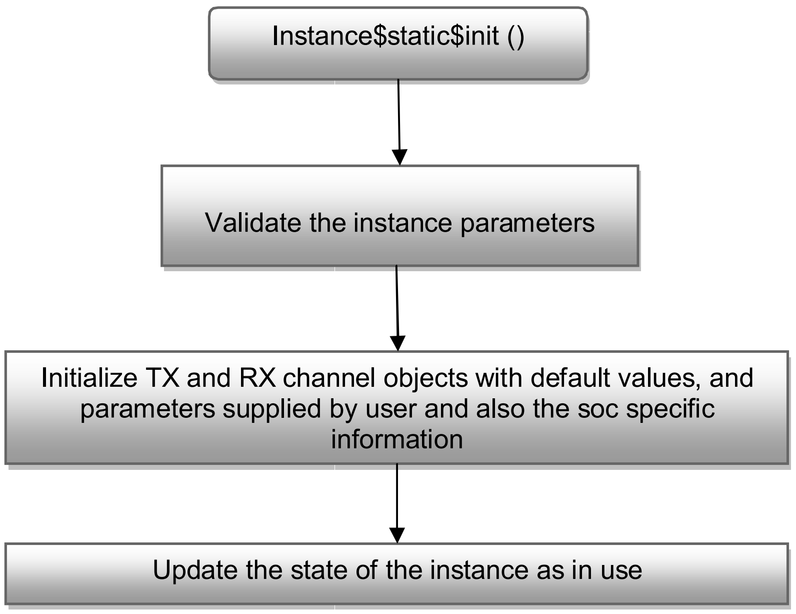

Instance$static$init ()
Validate the instance parameters
Initialize TX and RX channel objects with default values, and parameters supplied by user and also the soc specific information
Update the state of the instance as in use

Figure 4 Aic31_Instance_Init() flow diagram

2.3.1.2 Aic31_Instance_finalize()

The Aic31_Instance_finalize functions are called by the XDC/BIOS framework during the deleting of a driver instance

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 instance_init.

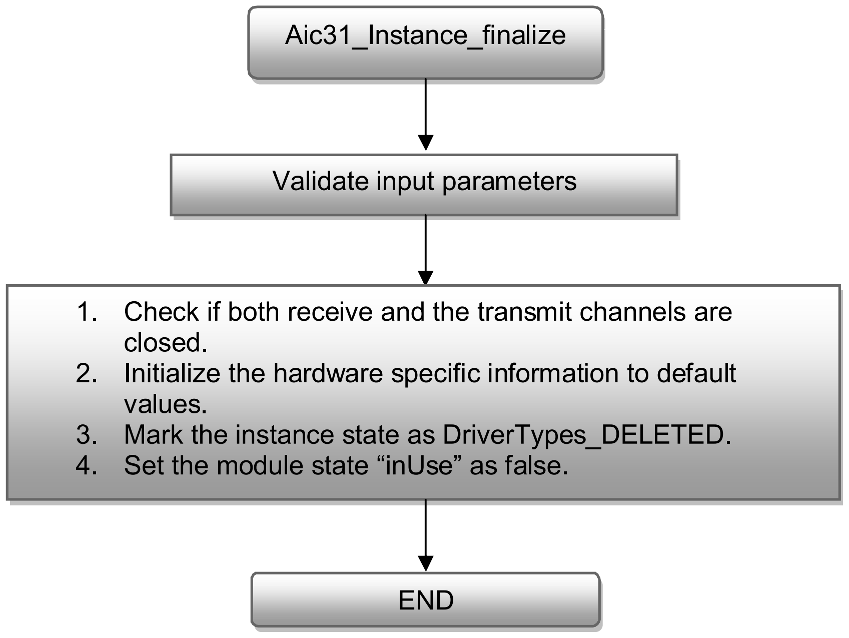

Aic31_Instance_finalize
Validate input parameters
1. Check if both receive and the transmit channels are closed.
2. Initialize the hardware specific information to default values.
3. Mark the instance state as DriverTypes_DELETED.
4. Set the module state "inUse" as false.
END

Figure 5 Aic31_Instance_finalize () control flow

2.3.1.3 *Aic31_Open()*

The AIc31_open function is called in response to the "Stream_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.

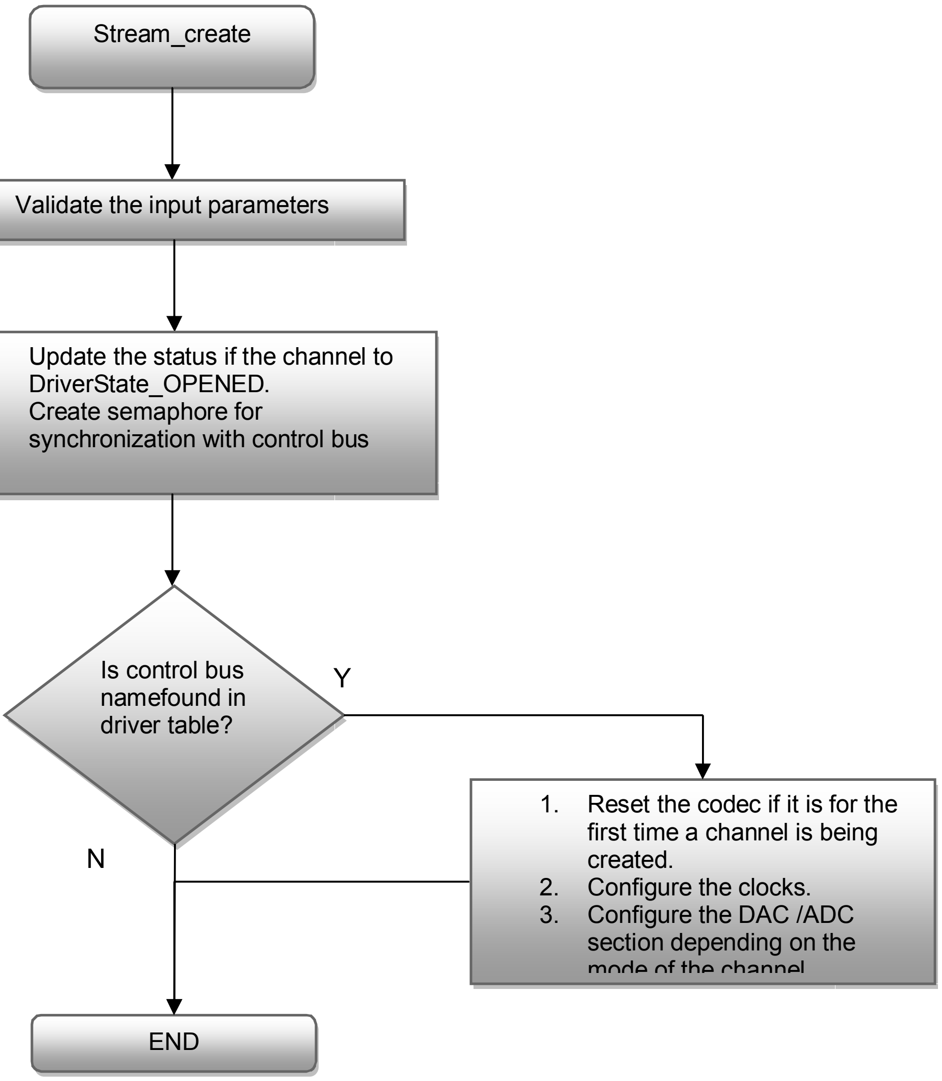

Stream_create
Validate the input parameters
1. Update the status if the channel to DriverState_OPENED.
2. Create semaphore for synchronization with control bus
Is control bus namefound in driver table?
Y
N
1. Reset the codec if it is for the first time a channel is being created.
2. Configure the clocks.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

Figure 6 Aic31_open () control flow

2.3.1.4 Aic31_Close()

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 “stream_delete” with the handle to the appropriate channel to close. This will invoke the Aic31_Close function provided by the Aic31 IDriver. 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

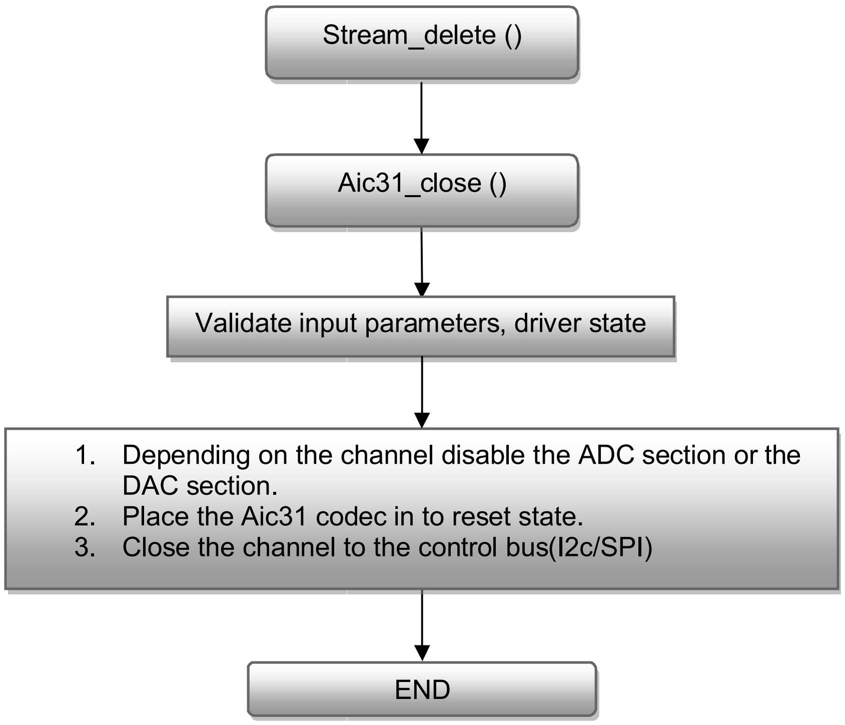

Stream_delete ()
Aic31_close ()
Validate input parameters, driver state
1. Depending on the channel disable the ADC section or the DAC section.
2. Place the Aic31 codec in to reset state.
3. Close the channel to the control bus(I2c/SPI)
END

Figure 7 Aic31_Close control Flow

2.3.1.5 Aic31_Control()

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 “stream_control” command to the IDriver.

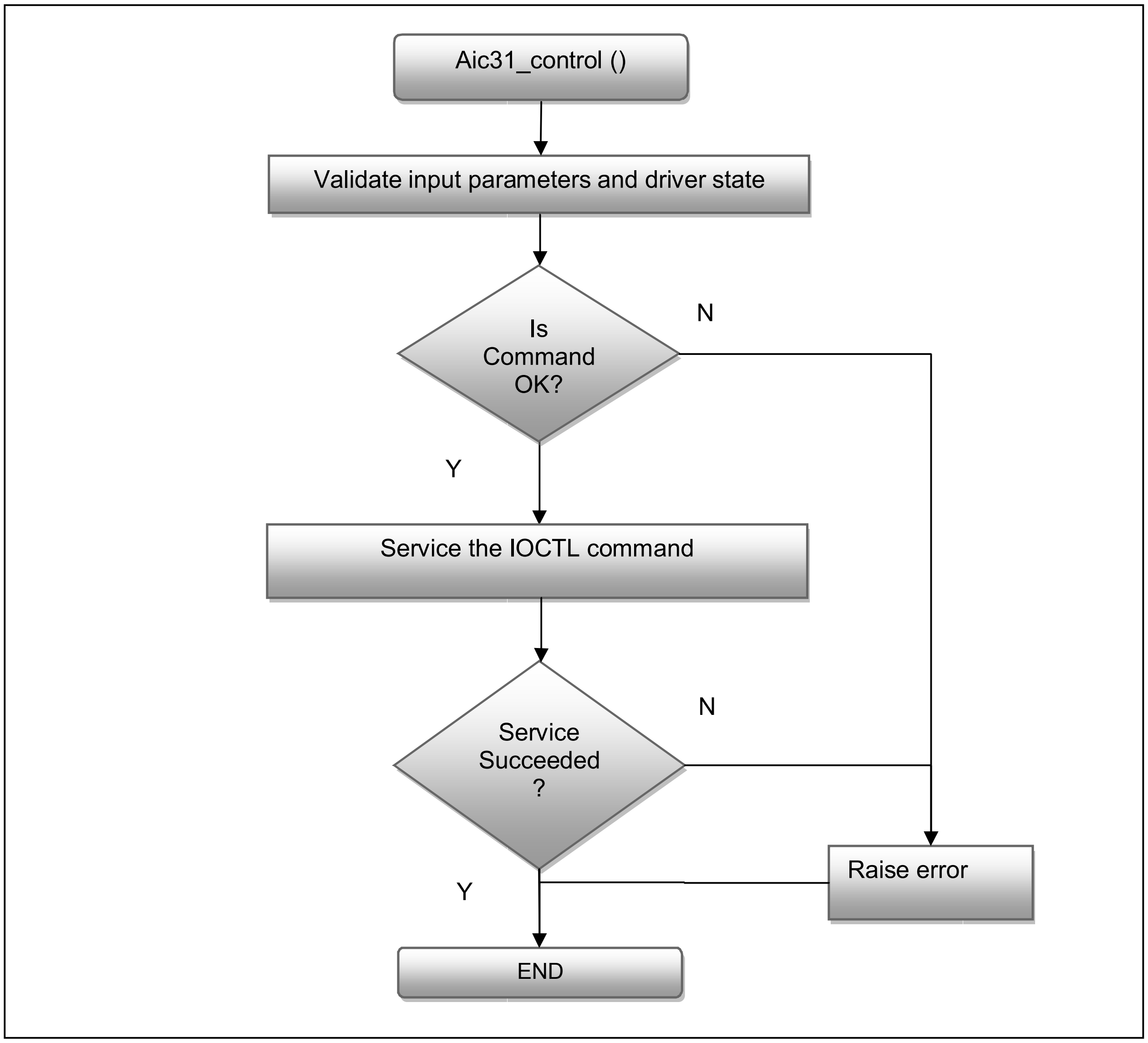

Aic31_control ()
Validate input parameters and driver state
Is Command OK?
N
Y
Service the IOCTL command
Service Succeeded ?
N
Y
Raise error
END

Figure 8 Aic31_Control Control flow

2.3.1.6 *Aic31_Submit()*

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 IDriver model the Aic31 driver will have a Aic31_Submit() function implemented. But, any IO transfer request from the application will cause an Error to be raised by the driver and return a DriverTypes_ERROR to the stream.

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.