TEXAS INSTRUMENTS

DSP/BIOS McASP Device 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 McASP Device. The target audience includes device driver developers from TI as well as consumers of the driver.

Trademarks

Notations

None

Revision History

Document Version	**Author(s)**	**Date**	**Comments**
0.1	Imtiaz SMA	September 23 2008	Created the document
0.2	Imtiaz SMA	October 23 2008	Fixed the review comments given for the document.
0.3	Imtiaz SMA	September 8 2008	Updating the document with the information regarding the new ISR handler and supported buffer formats.
0.4	Imtiaz SMA	January 21, 2009	Updated the document for the IOM driver and IDriver contrast sections.

Table of Contents

List Of Figures

Texas Instruments Proprietary

1 System Context

The purpose of this document is to explain the device driver design for McASP peripheral using DSP/BIOS operating system running on DSP OMAPL138 platform.

1.1 Terms and Abbreviations

McASP	Multi-channel Audio Serial Port
SPDIF	Sony/Philips Digital Interface
DIT	Digital Audio Interface Transmission
I2S	Inter-Integrated Sound protocol
TDM	Time Division Multiplexed
IOM	Input/Output Module

1.1 Disclaimer

This is a design document for the Mcasp driver for the DSP/BIOS operating system. Although the current design document explain the Mcasp 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 Mcasp driver 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 Related Documents

	DSP/BIOS Driver Developer’s Guide
SPRUFM1	McASP specification documents

1.4 Hardware

The McASP device driver architecture design is in the context of DSP/BIOS operating system running on DSP OMAPL138 core.

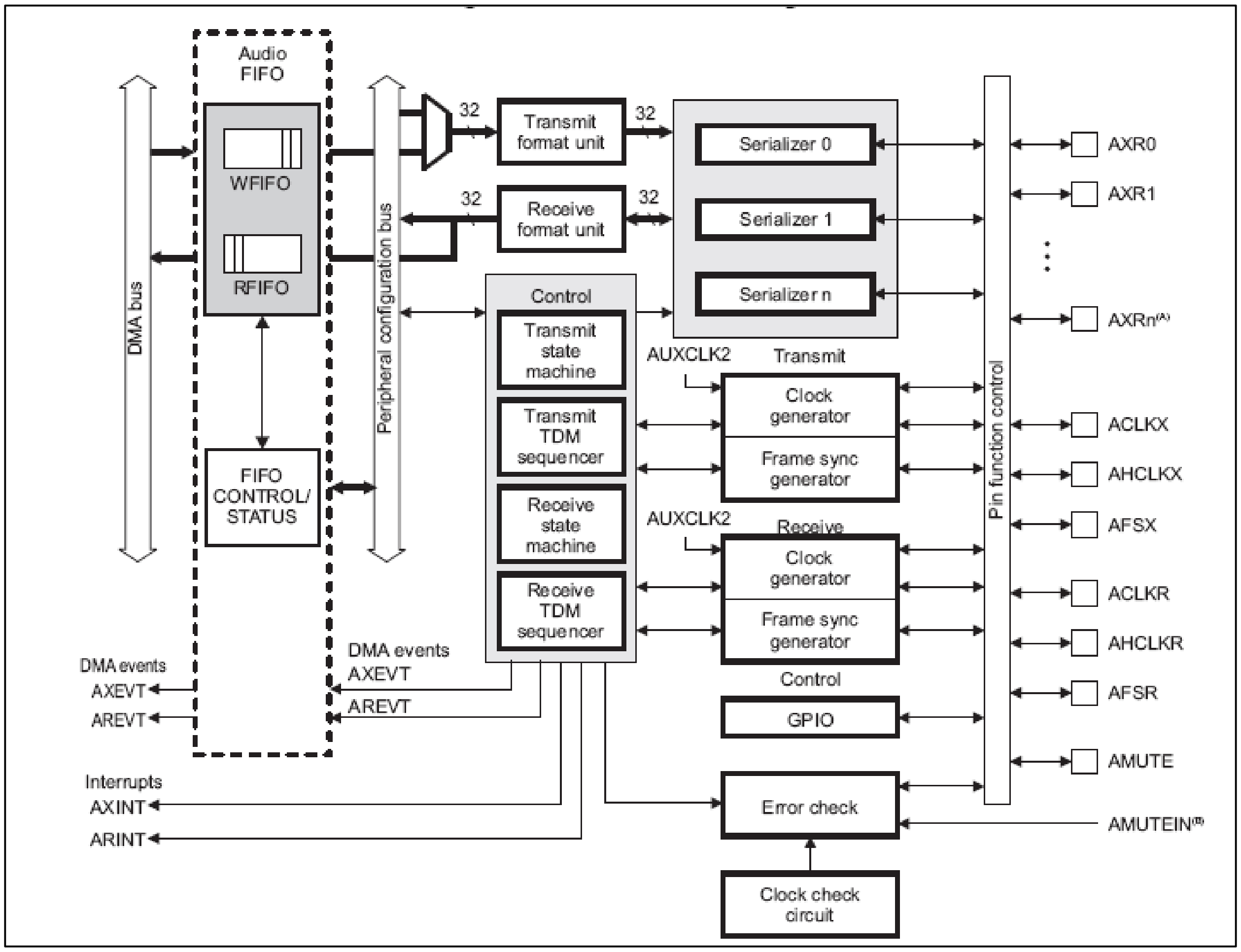


Figure 1 McASP Block Diagram

1.4.1 Features

1. The Mcasp is a general purpose serial port optimized for the need of the multichannel audio applications. It supports the TDM and DIT mode of data transfers.
2. The McASP consists of transmit and receive sections that may operate synchronized, or completely independently with separate master clocks, bit clocks, and frame syncs, and using different transmit modes with different bit-stream formats.

3. Extensive error checking and recovery
   a. Transmit under runs and receiver overruns due to the system not meeting real-time requirements
   b. Early or late frame sync in TDM mode
   c. Out-of-range high-frequency master clock for both transmit and receive
   d. DMA error due to incorrect programming.
4. Has hardware FIFO for additional buffering.
5. Has provision for generating the clocks internally or to be sourced from an external source.

1.5 Software

The McASP driver discussed here is running DSP/BIOS on the OMAPL138 DSP.

1.5.1 Operating Environment and dependencies

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

1.5.2 System Architechture

This device driver is written in conformance to the DSP/BIOS IDriver model and handles communication to and from the multi-channel audio serial port (McASP), and uses the EDMA mode to transfer the data.

The Application would invoke the driver routines through the Stream Calls APIs, which is an adaptation layer. The device drivers are accessed by the applications for performing I/O using Stream calls.

The transmit and receive sections of the McASP can be configured to operate independent of each other. All serializers of the McASP can be assigned to perform either operation; both acting synchronously as transmit or receive interface or any number of serializers can be set as transmit or receive mode.

The block diagram below shows the overall system architecture.

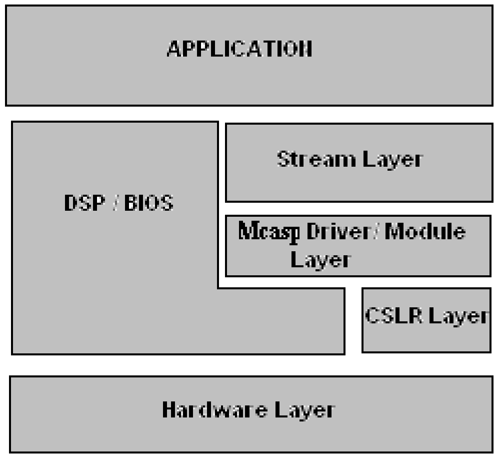


Figure 2 System Architecture

Driver module which this document discusses lies below the steam layer, which is an class driver layer provided by DSP BIOS™ (please note that the stream layer is designed to be OS independent with appropriate abstractions).The MCASP driver would use the rCSL (register overlay) to access the Hardware and would use the DSP BIOS™ APIs for OS services. Also as the IDriver is supposed to be an asynchronous driver (to the stream layer), any operation of the driver will not wait for the completion. (An exception is the driver control calls which will be executed synchronously). The Application would use the Stream APIs to make use of driver routines.

1.6 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The McASP driver module is object of IDriver class one may need to refer the IDriver documentation to access the McASP 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.

1.6.1 IDriver Interface

This McASP driver is intended to be an XDC module and this module would inherit the IDriver interface. Thus the McASP 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 an xdc file, xs file and source file as a minimum.

The McASP IDriver contains the following important files.

McASP module specifications file (Mcasp.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: Operation modes, Serializers setting, DIT/TDM mode etc

STRUCTURES: Communication status (no. of bytes transferred, errors etc.), Chanparms,

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 the data structures, enums, constants and module wide config variables. The Instance object (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 XDC framework translates this into the driver header file (Mcasp.h). 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.

McASP module script file (Mcasp.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 McASP module/driver. The Mcasp.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg for the Mcasp module, the module use function in the Mcasp.xs file is used to initialize the hardware instance specific details like base addresses, interrupt numbers, edma event numbers, number of serializers etc. 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 Mcasp instance

   When the Mcasp 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 (from 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

McASP source files (Mcasp.c, Mcasp_edma.c, and Mcasp_ioctl.c)

The Mcasp.c source file contains the IDriver API implementations. The Mcasp_edma .c contains the edma related functions required for the edma mode of operation and the Mcasp_ioctl.c contains the Ioctls supported for the Mcasp driver.please refer to the cdoc for the DSP BIOS for information pertaining to the IDriver interface.

The Mcasp IDriver module implements the following interfaces

S.No	IDriver Interface	Description
1	Mcasp_Module_startup()	Function called during the module startup. Places the Mcasp hardware in a reset state.
2	Mcasp_Instance_init()	Handles dynamic calls to module instantiation. This shall be a duplication of the tasks done by instance static init in the module script file.
3	Mcasp_Instance_finalize()	Function which deletes the Mcasp instance effectively.
4	Mcasp_open ()	Creates a communication channel in specified mode to communicate data between the application and the MCASP module instance.
5	Mcasp_close ()	Frees a channel and all its allocated resources.

6	Mcasp_control ()	Implements the IOCTLS for Mcasp IDriver module. All control operations go through this interface.
7	Mcasp_submit ()	Submit an I/O packet to a channel for processing. Used for data transfer operations like read and write operations

1.6.2 *Module Config Parameters*

This table lists the module wide configurable variables that are available to the application for configuring.

S.No	IOM Interfaces	Description
1	paramCheckEnable	Parameter that allows the user to enable/disable the input parameter checking in IDriver functions. Default value is TRUE

1.6.3 *CSLR Interface*

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the Mcasp IDriver module to configure Mcasp registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure which allows the access to the Mcasp registers.

1.7 Design Philosophy

This device driver is written in conformance to the DSP/BIOS IDriver model and handles communication to and from the Mcasp hardware.

1.7.1 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 (Mcasp module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be a module variable. For example, mode of operation (TDM/DIT) setting is a module wide variable. Default parameters for any channel opened shall be a module wide variable since, it should be available during module start up or use call, to initialize all the channels to their default values.

This instance word usage (Mcasp instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent the device directly by holding info like the TX/RX channel handles, device configuration settings, hardware configuration etc. Each hardware instance shall map to one Mcasp instance. This is represented by the Instance_State in the Mcasp module configuration file.

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

1.7.2 The Channel Concept

The transmit and receive sections of the McASP are represented by channels in the IDriver. The McASP device state is maintained in an ‘Instance Object’. This contains placeholders for two channels and all the requested serializers.

The lifetime of the channel is between its creation using Mcasp_open and its deletion using Mcasp_close.

The application can interface with the McASP peripheral in two ways:

Assign all serializers to the transmit/receive unit

In this case, the application requests for a single channel with all serializers to be created. The IDriver checks if all serializers are free to be allocated, and then allocates all to one channel. A single EDMA channel is allocated to service all serializers. The EDMA operation is dealt with in more detail in the next section. The other channel is invalid, and cannot be allocated, as there is no serializer is free to be assigned to this channel. Upon deletion of the active channel, all serializers are freed and can then be allocated to one/two channels as per the requirement.

2) Assign requested serializers per channel

In this case, the application presents the IDriver with the request for a channel with index of serializers to perform transmit/receive operation. If the resources are available, this channel is allocated, and EDMA channel is allocated for movement of data to requested serializers. The second channel can then be allocated on a request for another channel with rest of the free serializers.

Note: The driver does not support two concurrent channels performing the same operation (transmit/receive) i.e. any given channel has to operate in either the transmit or receive mode but not both simultaneously. The IDriver allocates two channels only if they perform different I/O operations.

The application can delete a channel after it has finished with all the transactions and no longer requires the channel for further operations. This channel can then be re-allocated to support a multiple serializers transfer subject to the availability of the serializers. In case all serializers are free, or both channels are deleted, the application can then request for a channel with all serializers allocated to it.

1.8 EDMA interaction with McASP IDriver channel

Each operation – transmit/receive – of the McASP will have EDMA channel allocated to it to service its data requirements. The two operations of the McASP are provided with individual EDMA event each to trigger transfers. The IDriver also registers the appropriate callback function to service interrupts raised by the individual channels.

The control flow of the callback function is discussed in the next section.

Note: The "Mcasp_localEdmaCallback" function services interrupts raised by DMA channels for both transmit and receive operations.

As a reference case, let us take up EDMA channel servicing the transmit section of the McASP. The receive operation would be similar in its treatment.

Depending on the channel configuration, two cases exist in channel configuration:

1) The transmit channel has one serializer assigned to it

While creating the transmit channel, the IDriver requests for a single EDMA channel to service this. The McASP transmit event is registered with this channel as a trigger for data transfer.

The EDMA interacts with the transmit section of the McASP using its data address on the DSP shared bus of VBUSP. During McASP channel create, the EDMA channel to service this is programmed with the destination address, element, frame and block sizes and to generate an interrupt after a frame/block is transferred. At this time two extra EDMA PaRAM tables are also requested and filled up with default values. For a transmit channel the source of transmit is taken as the application supplied buffer or a loop job buffer sent by the application or the default loop job buffer. Only if the application provides with a callback function for loop job that the interrupts are enabled for EDMA. The two EDMA PaRAM tables are used for linking. The moment when McASP does not have any buffers to send, it starts the loop job. The EDMA channel is then turned on and waits for the McASP to trigger it for data transfer. Each data transfer will happen at every event.

Note: The McASP supports only 32 bit transfers to the serializer buffers. Due to this limitation, the EDMA channel element size must necessarily be 32 bits.

When the McASP transmit section is taken out of reset, it triggers EDMA event. The EDMA channel transfers one element for every event triggered, which is done every time the McASP consumes the previous element transferred.

As per the programming of the EDMA, when the last element of the frame/block is transferred, a FRAMEIE/BLOCKIE interrupt is raised by this channel, which is serviced by the "Mcasp_localEdmaCallback"function.

This function checks the channels pending queue for further data packets. If the queue is empty, it links the loop job buffer (either passed from the main application or default loop buffer) to perform the loop job. The loop job buffer is transmitted until the application submits a request.

If the channel's pending queue holds data packets to be processed, the "Mcasp_localEdmaCallback" picks the next packet from the queue and programs the EDMA channel with its source address. This ensures a constant data flow to the McASP transmitter.

2) The transmit channel has multiple serializers assigned to it.

While creating the transmit channel, the IDriver requests for a single EDMA channel to service all the serializers. The McASP transmit event is registered with this channel as a trigger for data transfer.

Since the EDMA would have to send multiple elements for every event triggered – one for each serializer – the channel should be programmed to be frame synchronized. Each frame would contain multiple elements of data for multiple serializers. The EDMA channel should be programmed to trigger an interrupt for every block completion (BLOCKIE).

Note: The McASP supports only 32 bit transfers to the serializer buffers. Due to this limitation, the EDMA channel element size must necessarily be 32 bits.

Each data packet would contain data to service one block. The "Mcasp_localEdmaCallback" would be called on to service the BLOCKIE interrupt generated by the channel. This would then program the source address with the next data packet contents, as outlined in the earlier case.

Note: Since the EDMA channel supplies data for multiple serializers per event, the application should provide data that is interleaved to service each serializer alternately.

The figure below shows the data transferred for each event in case of a transmit channel.

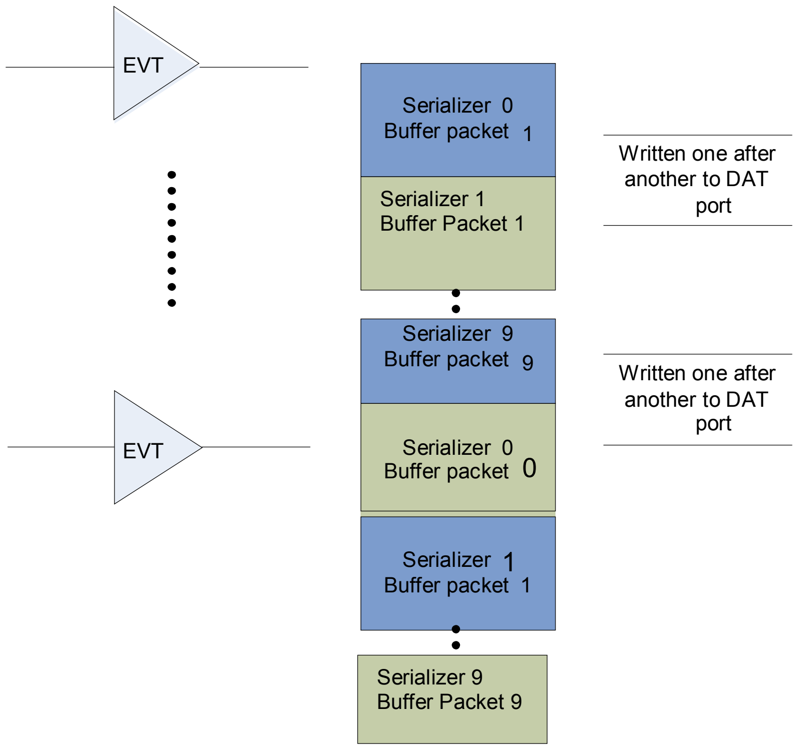

EVT
Serializer 0 Buffer packet 1
Serializer 1 Buffer Packet 1
Written one after another to DAT port
Serializer 9 Buffer packet 9
Written one after another to DAT port
EVT
Serializer 0 Buffer packet 0
Serializer 1 Buffer packet 1
Serializer 9 Buffer Packet 9

Figure 3 McASP Data transfer to Serializers

In case of receive data, the data read from the VBUSP port would be from alternate serializers for each event, and the application would have to sift through this to retrieve multiple data streams. It is required to program the McASP to change VBUSP from VBUS when it is operating in EDMA mode.

Also for receive data similar to the transmit data we request for two PaRAM tables used for linking. Loop job also operates in the similar way for receive. Once in EDMA callback ISR, if there are no other packets to be linked, it starts the loop job.

Note: EDMA channel fills the data to serializers which are configured as transmit mode for every transmit EDMA event. In the same manner it reads the data from all serializers when receive event is occurred.

1.9 EDMA Params and Linking

The Mcasp driver is used for transmission of audio data. Audio data requires that it should be handled with no or very little latencies. In case of latencies the user will be able to hear “breaks” or “noise” in the transferred data. Hence it is required that the Mcasp handles the next pending requests almost immediately as soon as the current request is completed.

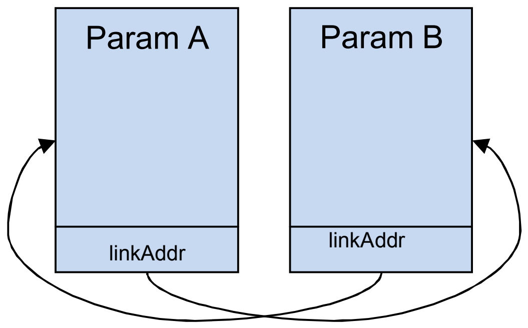
To handle these stringent timing constraints the Mcasp is programmed in the EDMA mode with the facility to link the next IO request in the queue immediately with the usage of EDMA channel linking facility.

The explanation of the linking scenario is as explained below.

Initially during the opening of the channel, the Mcasp IDriver will request for two spare EDMA PaRAM entries for the linking purpose.

The IDriver maintains two lists per channel for the handling of the requests.One is a queueFloatingList and the other is queueReqList.

The “queueFloatingList” will contain a maximum of two packets and these are the packets that are currently loaded in to the spare EDMA PaRAM sets. The remaining requests (other than the outstanding 2) will be queued up in the “queueReqList”.



1.10 Design Constraints

Due to hardware limitations, the McASP IDriver imposes certain constraints on how the data is routed to the device.

- All interactions with the McASP buffer registers take place only through the VBUS port with proper XBUF offset address when operating in interrupt mode.

  In case of transmit channel, the destination address should be derived from base address of McASP with offset of XBUF for which the data transfer is required for the lifetime of the channel.

  In case of receive channel, the source address should be programmed to a value derived from offset of XBUF for which the data transfer is required for the lifetime of the channel.

- All writes to the serializer buffer have to be of 32 bits length. The EDMA element size for all transfers should de declared as 32 bits.

- The EDMA cannot be used to fill in data to the DIT Channel Status RAM and User Data RAM.

  This has to be handled by the CPU or VBUS. In the IDriver, the DriverTypes_Packet contains pointers to the data to be filled in the Channel Status RAM and User Data RAM, and these are filled in before the EDMA is programmed with the corresponding packet during the "Mcasp_localEdmaCallback" interrupt service function.

2 McASP Driver Software Architecture

This chapter deals with the overall architecture of DSP/BIOS McASP device 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 Partition

The McASP driver is a single layer IDriver compliant implementation of the device driver as specified by the DSP BIOS operating system.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 Stream create call to the Mcasp IDriver module. The channel object is held inside the Instance State of the module. However, this instance state is translated to the Mcasp_Object structure by the XDC frame work. 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 IDriver employs the Instance State (Mcasp_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 (Mcasp_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. The handle to this is sent out to the application for access when the module instantiation is done via create statically (application CFG file) or dynamically (C file during run time). The parameters that are to be passed for this call is described in the section Device Parameters.

S.No	**Structure Elements *(Mcasp_Object)***	**Description**
1	*instNum*	Current instance number of the Mcasp
2	*devState*	Preserve the current state of the driver (Create/Deleted etc)
3	*isDataBufferPayloadStructure*	Whether the application buffer to be interpreted as payload structure.
4	*mcaspHwSetup*	Mcasp device hardware information for initializing
5	*hwiNumber*	Interrupt number used by the Mcasp
6	*enablecache*	Flag to enable/disable the usage of the cache
7	*stopSmFsXmt*	Flag to stop the transmit state machine
8	*stopSmFsRcv*	Flag to stop the receive state machine

9	*XmtObj*	Transmit channel object
10	*RcvObj*	Receive Channel object
11	*HwInfo*	Structure holding the instance specific information like base address etc
12	*serStatus*	Status of each serializers (e.g. Free/transmit/receive)

2.1.2.2 The McASP 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.

S.No	**Structure Elements *(Mcasp_Object)***	**Description**
1	*chanState*	Preserve the current state of the driver (Open/closed etc)
2	*mode*	Mode of operation of the channel
3	*devHandle*	Pointer to the Mcasp_Object structure
4	*cbFxn*	Callback function to be called on completion of an IO operation
5	*cbArg*	Argument to be passed to the callback function
6	*queueReqList*	List to handle all the pending IO packets
7	*queueFloatingList*	List to handle the currently processed IO requests
8	*noOfSerAllocated*	Number of serializers allocated for this channel
9	*channelOpMode*	Audio data transport protocol (DIT/TDM)

10	*isDmaDriven*	Option to check if the channel is DMA driven
11	*dataQueuedOnReset*	Data queued in the channel.
12	*intStatus*	Interrrupt status
13	*dataPacket*	Current IO packet pointer
14	*tempPacket*	Temporary IO packet pointer
15	*isTempPacketValid*	Flag to indicate whether the tempPacket field holds an valid packet.
16	*userDataBufferSize*	Size of the application given buffer
17	*submitCount*	Number of IO requests pending in the channel
18	*indexOfSersRequested*	Index of the serializers requested by the channel.
19	*edmaHandle*	Handle to the EDMA driver
20	*xferChan*	EDMA transfer channel
21	*tcc*	EDMA Transfer channel
22	*pramTbl[2]*	Spare channels of EDMA used for linking
23	*pramTblAddr[2]*	Physical address of EDMA channels used for linking
24	*nextLinkParamSetToBeUpdated*	Element holding the next paramset to be linked
25	*loopjobUpdatedinParamset*	Flag to check if the loop job is updated in the param set
26	*cpuEventNum*	Cpu interrupt number
25	*xferinProgressIntmode*	Flag to indicate that transfer is in progress in the interrupt mode
26	*loopJobBuffer*	Pointer to Loop job buffer to be used when the Mcasp is idle
27	*loopJobLength*	Length of the loop job to be used for each serializer
28	*roundedWordWidth*	Length of the word to be

		transferred in EDMA
29	*currentDataSize*	Current transfer size
30	*bMuteON*	Flag to indicate if the mute is on for this channel
31	*paused*	Flag to indicate if the channel is paused
32	*edmaCallback*	Pointer to the edma callback function
33	*gblErrCbk*	Pointer to the call back to be called in case of error
34	*nextFlag*	Flag used to check if the channel state machine can be stopped
35	*currentPacketErrorStatus*	Current packet Error status is maintained here

2.1.2.3 The Mcasp chanParams structure

Structure used to supply the channel parameters during the creation of the channel.

S.No	Structure Elements *(Mcasp_chanparams)*	Description
1	*noOfSerRequested*	Number of serializers requested
2	*indexOfSersRequested[]*	Index of the requested serializers
3	*mcaspSetup*	Pointer to the Mcasp hardware set up structure
4	*isDmaDriven*	Flag to indicate if the channel is DMA driven
5	*channelMode*	Audio transport protocol to be used(DIT/DTM)
6	*wordWidth*	Size of the data to transferred
7	*userLoopJobBuffer*	Loop job buffer to be used (optional)
8	*userLoopJobLength*	Loop job buffer length
9	*hEdma*	Handle to the EDMA driver

10	*gblCbk*	Pointer to the callback function to be called in case of Errors
11	*noOfChannels*	Number of channels to be transmitted (used only in case of TDM mode).
12		

2.1.2.4 *The Mcasp_ PktAddrPayload structure*

This is the format of the audio data to be sent by the application in case of Using DIT mode.

S.No	**Structure Elements** ***(Mcasp_PktAddrPayload)***	**Description**
1	*chStat*	Channel status ram info
2	*userData*	User information
3	*writeDitParams*	Whether the DIT params are to be written
4	*addr*	Actual address of the buffer to be transferred

2.1.2.5 *The Mcasp_ Params structure*

S.No	**Structure Elements** ***(Mcasp_PktAddrPayload)***	**Description**
1	*instNum*	Instance number of the Mcasp to use
2	*enablecache*	Whether cache has to be used.
3	*hwiNumber*	Hwi number to be used by the Mcasp device
4	*isDataBufferPayloadStructure*	Whether the buffer is to be interpreted as payload structure or normal buffer
5	*mcaspHwSetup*	The Mcasp hardware initialization strcuture

2.2 Dynamic view of the DSP/BIOS McASP driver

The McASP driver provides the application data paths to transport the audio data to and from the SOC. The serializers can be configured in transmit or receive mode depending on the requirement of the application.

The below shown block diagram depicts the dynamic view of the DSP/BIOS Mcasp driver.

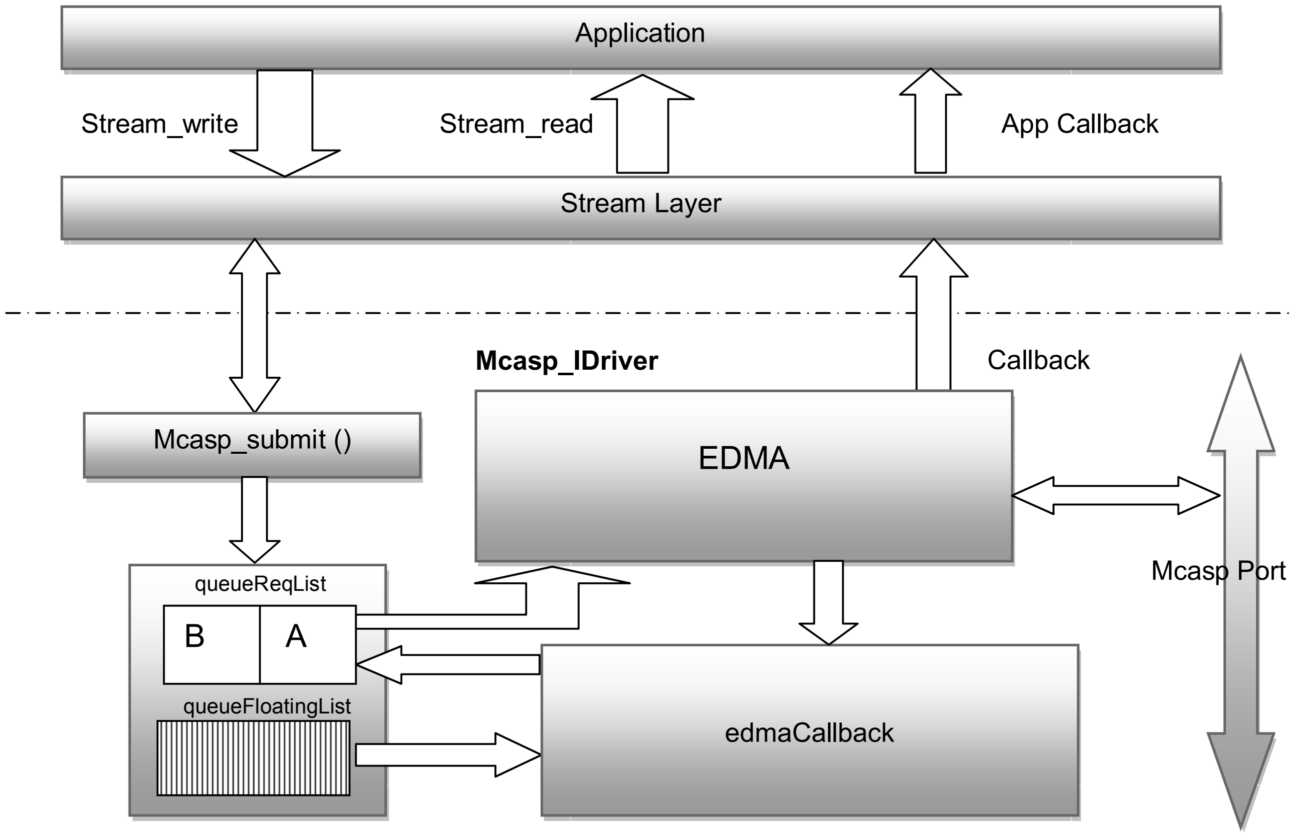


Figure 4 McASP Driver Dynamic View

The following sections describe in the detail the various steps in the creation of the Mcasp driver.

2.2.1 *Mcasp_Module_Startup*

This function is called by the DSP BIOS during the startup sequence. Every DSP BIOS module can specify a startup function to be called before the module is used. The “Mcasp_Module_Startup” is the Mcasp modules start up function. This function is called until it succeeds (i.e. return STARTUP_DONE) or until the “MAX_NUM_PASSES” is exceeded.

This function initializes the Mcasp hardware and places it in a known state.

2.2.2 *Mcasp_Instance_init*

This function is called during the initialization of an instance dynamically. Usually the Mcasp driver can be created in two ways.

1. Static creation
2. Dynamic creation

Static creation

In static creation the Mcasp module is initialized through a cfg file. This initialization is done during the compiling of the module.

Dynamic creation

In dynamic creation the Mcasp module is created dynamically during the runtime. This is achieved by calling the function “Mcasp_create ()”

This function basically initializes all the data structures and places them in a default state. it also initializes the Mcasp hardware and places the hardware in a known state.

2.2.3 Mcasp_open

The Mcasp_open function is called in response to the “Stream_create” function called by the application. This function creates a dedicated channel for data streaming between the device and the application.

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 then allocates the required resources like EDMA channels, registering of interrupts for error handling etc.

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 further IO and control requests to be sent to the driver from the application.

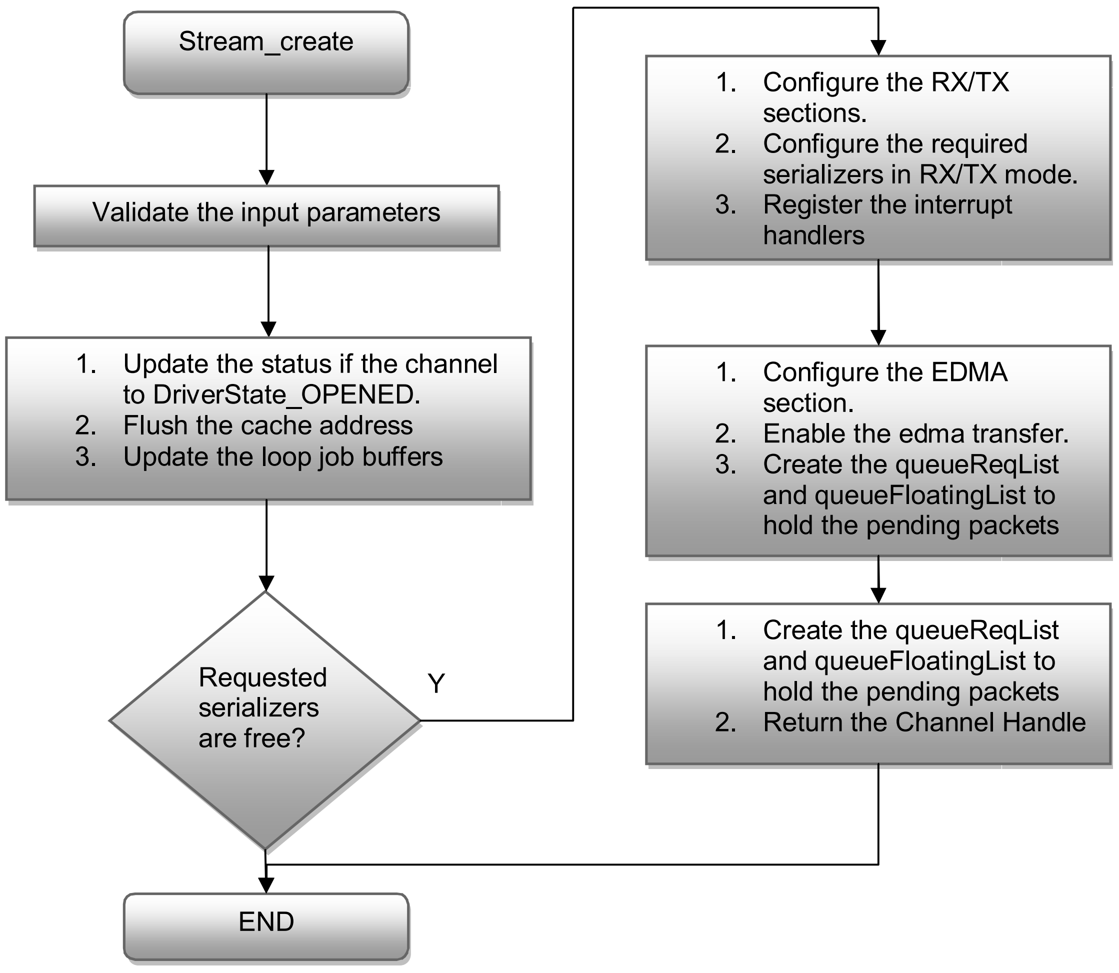

Stream_create
Validate the input parameters
1. Update the status if the channel to DriverState_OPENED.
2. Flush the cache address
3. Update the loop job buffers
Requested serializers are free?
Y
1. Configure the RX/TX sections.
2. Configure the required serializers in RX/TX mode.
3. Register the interrupt handlers
1. Configure the EDMA section.
2. Enable the edma transfer.
3. Create the queueReqList and queueFloatingList to hold the pending packets
1. Create the queueReqList and queueFloatingList to hold the pending packets
2. Return the Channel Handle
END

Figure 5 McASP Open API

2.2.4 Mcasp_Submit

The application invokes Stream_read () and Stream_write () APIs for data transfer using McASP. These APIs in turn submits a DriverTypes_packet containing all the transfer parameters needed by the IDriver to program the underlying hardware for data transfer.

The Mcasp IDriver provides the "Mcasp_submit" api to handle all the IO requests for the Mcasp driver from the application. The DriverTypes_Packet has the command code embedded inside it.

The Mcasp_submit handles the following commands

1. DriverTypes_READ – Read command request
2. DriverTypes_WRITE – Write command request

The McASP driver works essentially in the asynchronous IO mode.

Asynchronous IO Mechanism

A driver is said to be in asynchronous mode when multiple IO requests can be submitted to the driver by a thread without causing the thread to block till the completion of the IO request. Usually after the completion of the IO a callback function registered by the application will be called to notify the thread of the IO completion.

The McASP IDriver's async mode working can be broadly divided in to the following sections.

1. DriverTypes_Packet Queuing
2. EDMA callback and subsequent packet loading.
3. Loop job buffer usage.

2.2.4.1 *DriverTypes_Packet Queuing*

The most important aspect of the Mcasp IDriver for the async mode is the queuing of the IO requests while a packet is currently under processing.

When the IDriver submit function is called to submit a new IO request the following is the sequence of events that take place in the driver.

1. The command embedded within the IO packet is verified for validity.
2. The buffer address supplied by the application is validated.
3. Since, essentially the Mcasp driver works in EDMA mode the cache is flushed.

The McASP driver essentially uses two lists to maintain the IO packets

1. **queueFloatingList** is the list that contains the IO requests which are currently loaded in to the spare param sets of the EDMA. This should hold a maximum of two IO requests i.e. the one currently being processed and the other being the packet to be loaded next.
2. **queueReqList** is the lists that holds the pending IO requests after the floating list is FULL. Can hold any number of packets.

Now, there are two possible scenarios.

1. The “queueFloatingList” is empty i.e. there are no current requests under processing.
2. The “queueFloatingList” is full i.e. there are currently two request also queued in the driver.

queueFloatingList NOT FULL

When the queueFloatingReqList is not full, It means that the driver is currently having either no request or one outstanding request only. In this case the IO request is directly queued in to the “queueFloatingList”.

Now the new EDMA parameters pertaining to the IO packet are loaded in to the spare EDMA param sets (requested while opening the channel).

In case that this is the first packet to be loaded in to the “queueFloatingList” the EDMA param set has to be loaded to the currently running loop job buffer. Then the EDMA transfer needs to be enabled.

queueFloatingList FULL

When the queueFloatingReqList is full, it means that there are currently two outstanding requests in the EDMA driver and that no further requests can be submitted to the EDMA. Hence the IO packet is added to the pending IO list i.e. “queueReqList”.

The below figure shows the flow diagram in the Mcasp_submit context

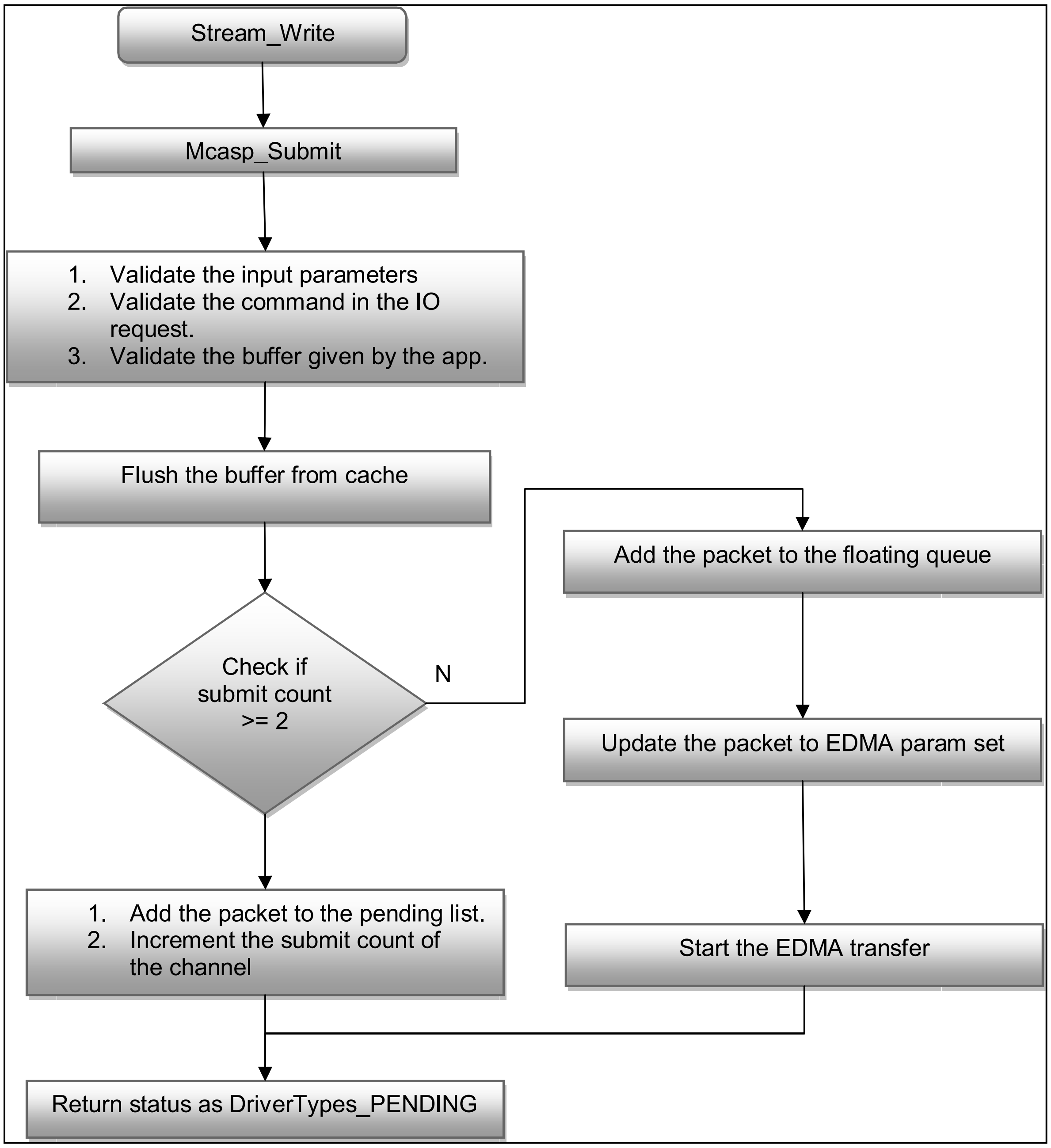

Stream_Write
Mcasp_Submit
1. Validate the input parameters
2. Validate the command in the IO request.
3. Validate the buffer given by the app.
Flush the buffer from cache
Check if submit count >= 2
N
Add the packet to the floating queue
Update the packet to EDMA param set
Start the EDMA transfer
1. Add the packet to the pending list.
2. Increment the submit count of the channel
Return status as DriverTypes_PENDING

Figure 6 McASP Submit

2.2.4.2 EDMA callback function

The Mcasp driver has a single callback function registered for both receive and transmit EDMA events. The callback function is invoked after the completion of the transfer.

The EDMA completion status is updated in to the DriverTypes_Packet error field as the request completion status. In case of Error the status is updated as “Mcasp_E_dmaError” else “DriverTypes_COMPLETED” is updated.

Once the status of the packet is completed, the “queueReqList” is checked to find if there are any packets pending.

queueReqList is EMPTY

In case that the”queueReqList" list is empty, it means that there are no outstanding requests in the Mcasp driver. Hence the callback then programs the loop job buffer (detailed in the next section). The EDMA param set is updated to take up the loop Job buffer.

queueReqList is NOT EMPTY

In case that the “queueReqList” is not empty, it means that there are pending requests in the McASP driver. The pending packet is taken from the “queueReqList” and is programmed in to the empty EDMA param set and the currently executing packet’s param set is linked with the newly loaded param set.

2.2.4.3 Loop job Buffer

When the Mcasp driver is not having any pending requests and the Mcasp driver is not stopped, the loop job buffer is loaded.

The loop job buffer is a buffer with all “zeros” data which will be used by the EDMA during the idle time of the Mcasp (no IO requests are present).The Application has the facility of providing the loop job buffer of its choice during the creation of the channel else a buffer internal to the Mcasp driver is used.

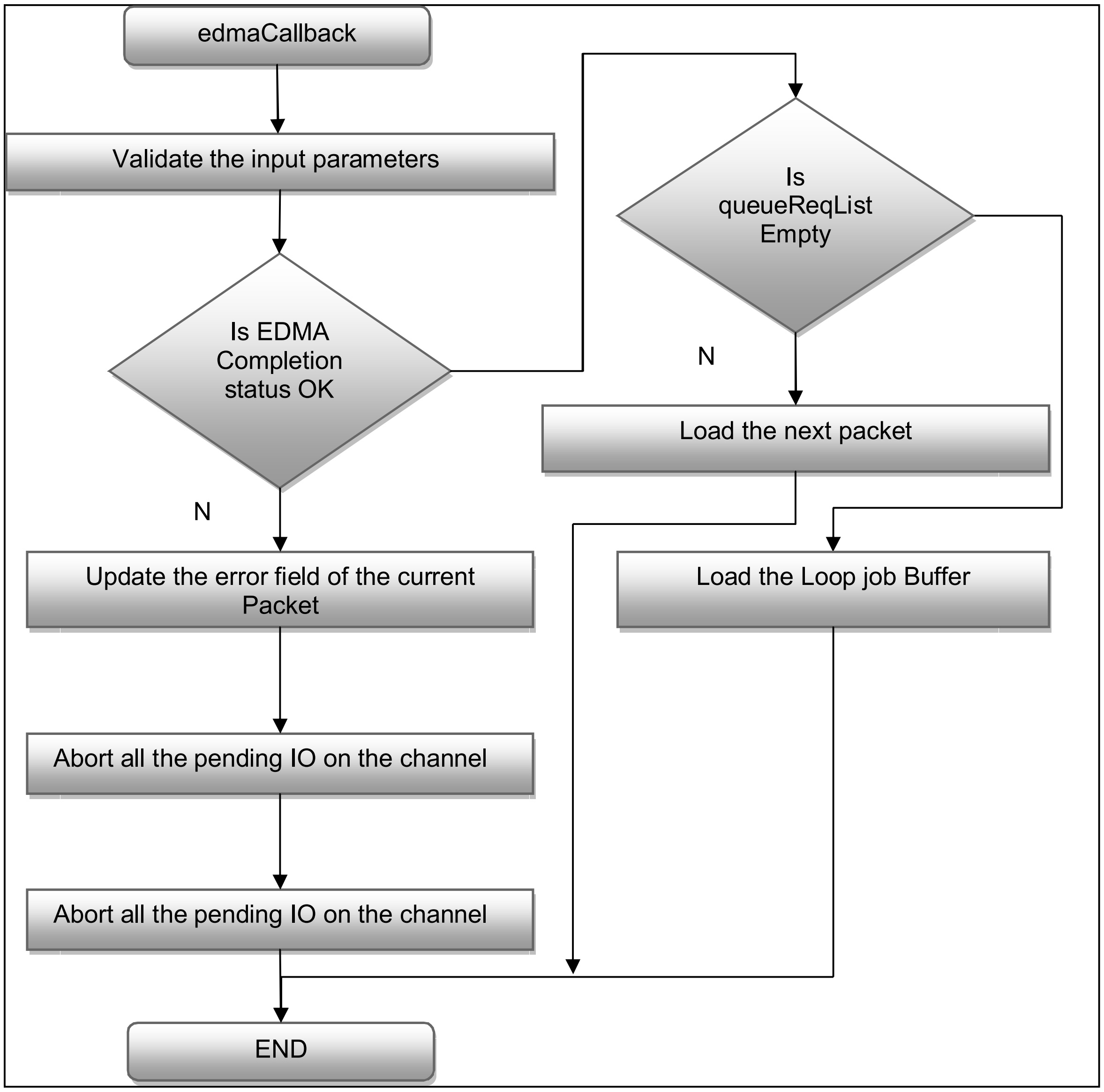

edmaCallback
Validate the input parameters
Is EDMA Completion status OK
N
Update the error field of the current Packet
Abort all the pending IO on the channel
Abort all the pending IO on the channel
END
Is queueReqList Empty
N
Load the next packet
Load the Loop job Buffer

Figure 7 McASP Edma Callback

2.2.5 Mcasp_Control

The Mcasp IDriver provides an interface for controlling the driver by issuing a set of specified control commands. The application can use the control commands to control the functionality of the McASP driver by issuing the "stream_control" command to the IDriver.

The following figure shows the flow diagram of the Mcasp_control command

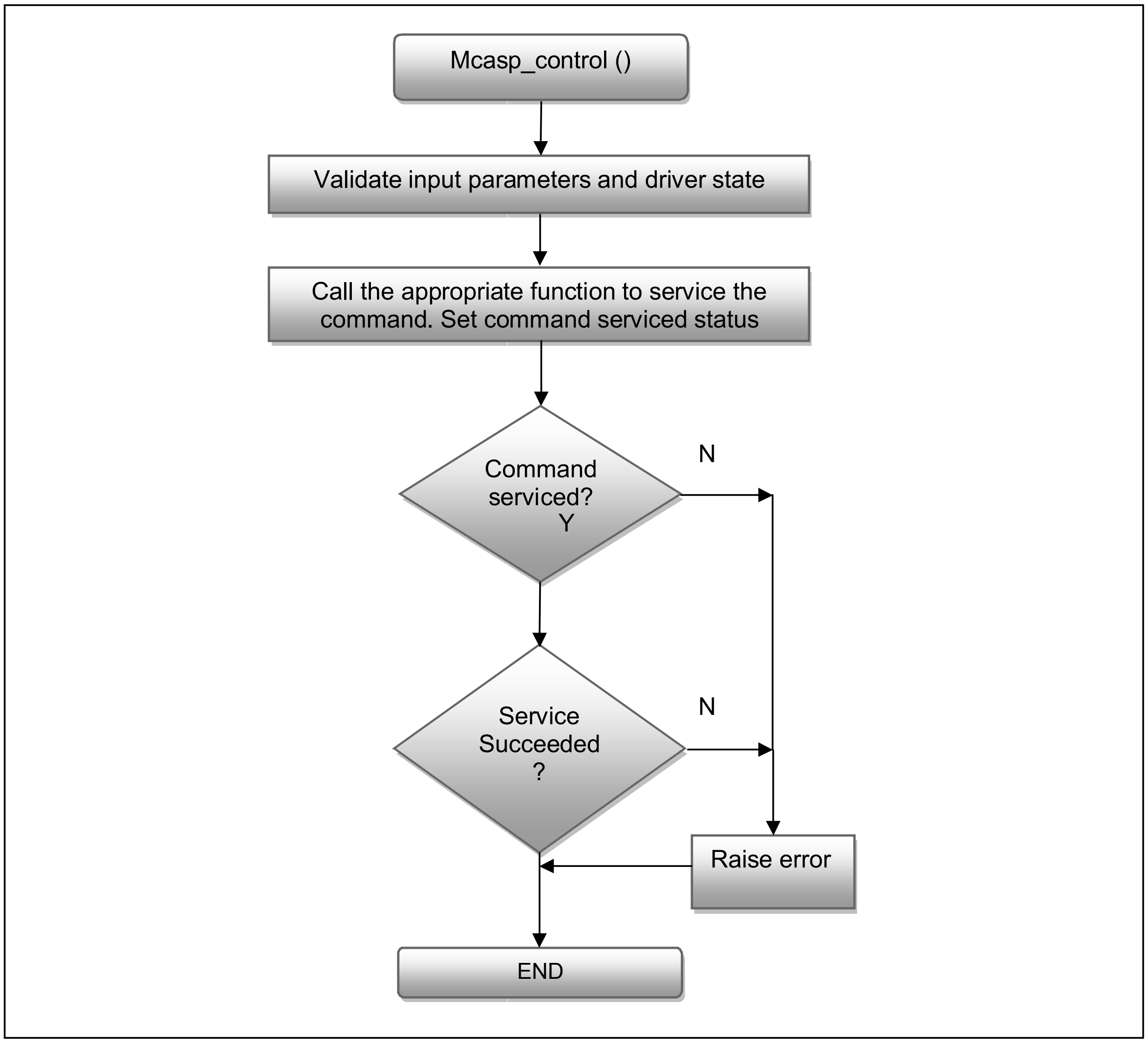

Mcasp_control ()
Validate input parameters and driver state
Call the appropriate function to service the command. Set command serviced status
Command serviced?
Y
N
Service Succeeded ?
N
Raise error
END

Figure 8 McASP control API

The following sections list the IOCTL commands supported by the McASP driver.

2.2.5.1 Mcasp_IOCTL_DEVICE_RESET

Command	Mcasp_IOCTL_DEVICE_RESET
Parameters	None

The application issues this command to reset the Mcasp device. On receiving this command the Mcasp aborts all the current pending IO requests and resets the Mcasp.

Note: This command resets the entire Mcasp instance irrespective of the channel to which the command is issued.

2.2.5.2 Mcasp_IOCTL_CNTRL_AMUTE

Command	Mcasp_IOCTL_CNTRL_AMUTE
Parameters	Value to write to AMUTE register

The application issues this command to control the AMUTE pin of the McASP device. The application also needs to specify the value to be written to the AMUTE register.

2.2.5.3 Mcasp_IOCTL_START_PORT

Command	Mcasp_IOCTL_START_PORT
Parameters	None

This IOCTL function allows the application to start the state machine of the required channel. The McASP driver has two channels receive and transmit channel. This command starts the state machine of the channel for which the start command has been issued.

2.2.5.4 Mcasp_IOCTL_STOP_PORT

Command	Mcasp_IOCTL_STOP_PORT
Parameters	None

This IOCTL function allows the application to stop the state machine of the required channel. The McASP driver has two channels, receive and transmit channels. This command stops the state machine of the channel for which the command has been issued.

2.2.5.5 Mcasp_IOCTL_QUERY_MUTE

Command	Mcasp_IOCTL_QUERY_MUTE
Parameters	Pointer to the variable to hold the AMUTE register value

This command is used by the application to query the value of the McASP AMUTE register. The application provides a pointer where the queried valued is updated

2.2.5.6 Mcasp_ IOCTL_CTRL_MODIFY_LOOPJOB

Command	Mcasp_ IOCTL_CTRL_MODIFY_LOOPJOB
Parameters	Pointer to the “Mcasp_ChanParams” structure

The application uses this command to modify the loop job buffer to be used by the EDMA driver. The loop job buffer is used by the EDMA when it has no valid packet to transmit. The application has to pass the new “Mcasp_ChanParams“ which contain the new loop job buffer address and the size parameters which will be used by the EDMA.

2.2.5.7 *Mcasp_IOCTL_CTRL_MUTE_ON*

Command	Mcasp_IOCTL_CTRL_MUTE_ON
Parameters	None.

This command mutes the Mcasp device i.e. only zeros will be sent instead of an audio stream.

2.2.5.8 *Mcasp_IOCTL_CTRL_MUTE_OFF*

Command	Mcasp_IOCTL_CTRL_MUTE_OFF
Parameters	None

This command is used to “un-mute” the previously muted Mcasp device. In case of trying to un-mute a channel which is not muted, an error is given by the IDriver.

2.2.5.9 *Mcasp_IOCTL_PAUSE*

Command	Mcasp_IOCTL_PAUSE
Parameters	None

This command sets the McASP device in to pause i.e. no more IO packets are processed by the McASP. All the new requests will be queued up in the IDriver.

2.2.5.10 *Mcasp_IOCTL_RESUME*

Command	Mcasp_IOCTL_RESUME
Parameters	None

This command is used to resume the McASP device which is paused previously. In the case that the Mcasp is not is a paused state the IDriver raises an error.

2.2.5.11 Mcasp_IOCTL_SET_DIT_MODE

Command	Mcasp_IOCTL_SET_DIT_MODE
Parameters	Value to write to DITCTL register

This command is used to modify the Mcasp audio data transport protocol to the DIT mode.

2.2.5.12 Mcasp_IOCTL_CHAN_TIMEDOUT

Command	Mcasp_IOCTL_CHAN_TIMEDOUT
Parameters	None

This command is to be called in case a timeout is encountered during a channel operation. This command aborts the timed out channel.

2.2.5.13 Mcasp_IOCTL_CHAN_RESET

Command	Mcasp_IOCTL_CHAN_RESET
Parameters	None

This command is used by the application to reset a McASP channel

2.2.5.14 Mcasp_IOCTL_CNTRL_SET_FORMAT_CHAN

Command	Mcasp_IOCTL_CNTRL_SET_FORMAT_CHAN
Parameters	Pointer to the new “Mcasp_HwSetupData” structure

This command is used to modify the channel settings of the McASP channel. It configures the Mcasp channel with the new hardware set up data sent by the application.

2.2.5.15 Mcasp_IOCTL_CNTRL_GET_FORMAT_CHAN

Command	Mcasp_IOCTL_CNTRL_GET_FORMAT_CHAN
Parameters	Pointer to the “Mcasp_HwSetupData” structure to hold the data

The application can use this command to get the information about the current channel. The application needs to provide the pointer to the “Mcasp_HwSetupData” structure to hold the data.

2.2.5.16 Mcasp_IOCTL_CNTRL_SET_GBL_REGS

Command	Mcasp_IOCTL_CNTRL_SET_GBL_REGS
Parameters	Pointer to the “Mcasp_HwSetup” structure

The application can use this command to set the global control register. The application needs to send the pointer to the new “Mcasp_HwSetup” data structure that needs to be programmed.

2.2.5.17 Mcasp_IOCTL_SET_DLB_MODE

Command	Mcasp_IOCTL_SET_DLB_MODE
Parameters	DLB mode enable or disable

This command is used to set the McASP in to the loopback mode.

2.2.5.18 Mcasp_IOCTL_ABORT

Command	Mcasp_IOCTL_ABORT
Parameters	None

The application issues this command to abort the pending requests of the channel. This IOCTL aborts all the pending request of the channel and stops the state machine. The EDMA transfer is also stopped.

2.2.6 Mcasp_close

Once the application has finished with all the transaction with the McASP 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 Mcasp_Close function provided by the McASP IDriver. The close function deallocates all the resources allocated to the McASP 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.

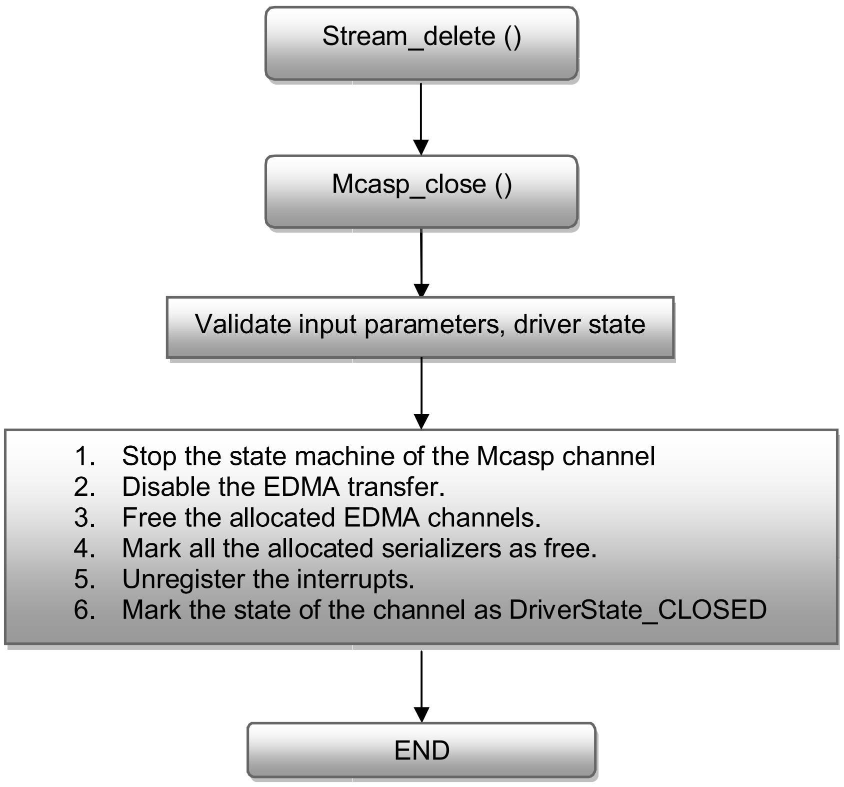

Stream_delete ()
Mcasp_close ()
Validate input parameters, driver state
1. Stop the state machine of the Mcasp channel
2. Disable the EDMA transfer.
3. Free the allocated EDMA channels.
4. Mark all the allocated serializers as free.
5. Unregister the interrupts.
6. Mark the state of the channel as DriverState_CLOSED
END

Figure 9 McASP close

2.2.7 *Mcasp_Instance_finalize*

The Mcasp_Instance_finalize is called during the closing of the instance. The function initializes the instance specific information to default values and then sets the state of the instance as DriverState_DELETED.

The below flow diagram shows the control flow of the function

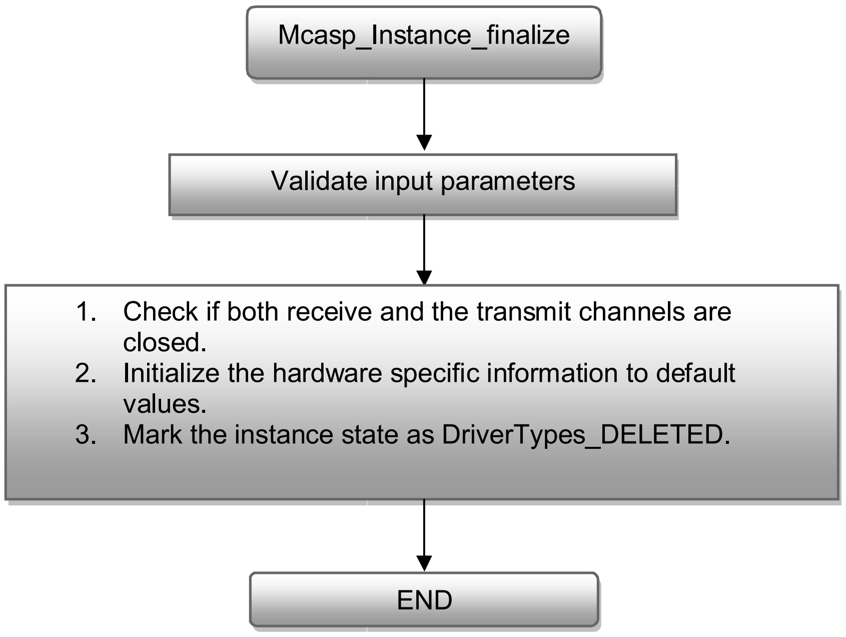

Mcasp_Instance_finalize
Validate input parameters
1. Check if both receive and the transmit channels are closed.
2. Initialize the hardware specific information to default values.
3. Mark the instance state as DriverTypes_DELETED.
END

Figure 10 McASP instance Finalize