Texas Instruments

DSP/BIOS McASP Device Driver

Architecture/Design Document

November 15, 2007

Platform Software Group

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

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

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 Driver Developer's Guide
- spru980.pdf

Notations

None

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

Revision History

Date	Author	Comments	Version
June 19, 2006	Pratik Joshi	Created the Document	1.0
September 6, 2006	Saloni Shah	Changes added for DMA mode	1.1
September 18, 2006	Saloni Shah	Changes for Release 0.1.4	1.2
September 21, 2006	Saloni Shah	BIOS version changed to 5.31	1.3
October 5, 2006	Saloni Shah	BIOS version changed to 5.31	1.4
November 30, 2006	Pratik Joshi	Changes for the release for 0.3.0	1.5
January 16, 2007	Pratik Joshi	Bios version changed to 5.31.02	1.6
January 29, 2007	Pratik Joshi	CCS version changed	1.7
April 23, 2007	Pratik Joshi	Updated for release 0.7.0	1.8
June 22, 2007	Anuj Aggarwal	Updated for GA Patch Release 1.00.01	1.9
July 2, 2007	Pratik Joshi	Updated for GA release 1.00.03.00	1.10
June 29, 2007	Amit Chatterjee	Modified Release Version	2.0
July 18, 2007	Maulik Desai	Modified Release Version	2.1
November 15, 2007	Nagarjuna K	Modified for DM648/C6452 and DM6437/C6424	2.2

Table of Contents

List Of Figures

1 System Context

1.1 Hardware

The McASP device driver architecture design is in the context of DSP/BIOS operating system running on DSP 64+ joule

1.2 Software

The McASP mini-driver discussed here is running DSP/BIOS on the 64x+ DSP. However the McASP driver can also be ported to any other OS, with minimal modifications in the OS specific section of the driver. More details can be found in the later part of this section.

1.2.1 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.

1.3 Design Philosophy

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

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.

1.3.1 The Channel Concept

The transmit and receive sections of the McASP are represented by channels in the mini-driver. The McASP device state is maintained in a *'Port Object'*. This contains placeholders for two channels and all requested serializers.

The lifetime of the channel is between its creation using mcasp_mdCreateChan and its deletion using mcasp_mdDeleteChan.

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

1) 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 mini-driver 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 if DMA mode of data transfer is requested. 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 mini-driver 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 if DMA mode of data transfer is requested. 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). The mini-driver allocates two channels only if they perform different I/O operations.

The application can delete a channel after it has finished with that operation. This channel can then be re-allocated to support a multiple serializers transfer. 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.

1.4 EDMA interaction with McASP mini-driver 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 EDMA event each to trigger transfers. The mini-driver also registers the dmaIsr function to service interrupts raised by this channel.

The contents of the data packet processed every time the dmaIsr is invoked are discussed in the next section under the IOM_Packet topic.

Note: The dmaIsr 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 mini-driver 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 EDMA PaRAM tables are also requested and filled up with default values. For a transmit channel the source of transmit is either taken as the 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 dmaIsr 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. It waits for the application to send another buffer to be processed.

If the channel's pending queue holds data packets to be processed, the dmaIsr 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 mini-driver 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 dmaIsr 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.

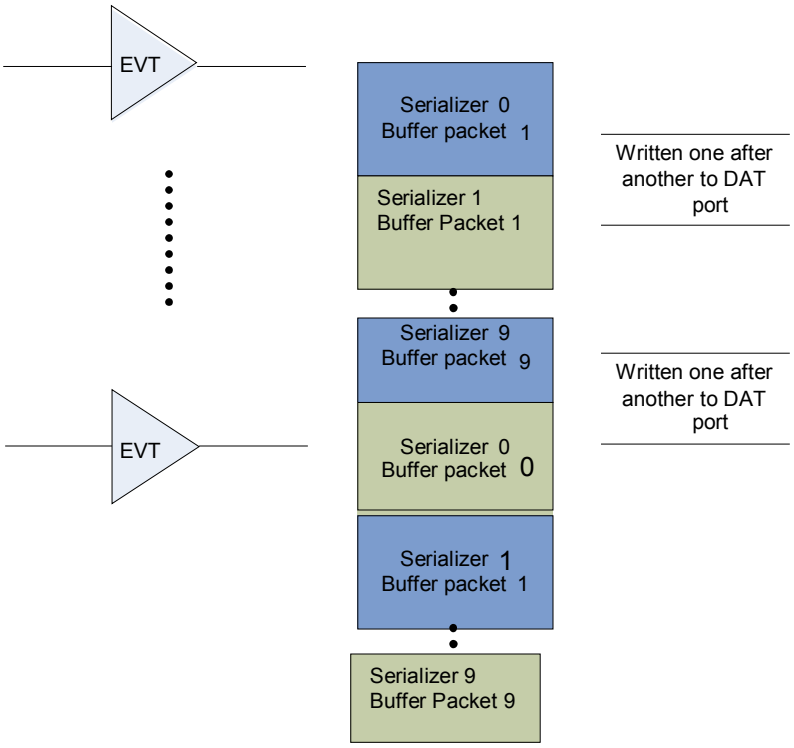

Figure 1 Device driver Layer

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. On 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.

1.5 Design Constraints

Due to hardware limitations, the McASP mini-driver 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 mini-driver, the IOM_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 dmaIsr 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 device driver is partitioned into distinct sub-components, consistent with the roles and responsibilities already discussed in section 1.3. In the following sub-sections, each of these functional sub-components of the device driver is further elaborated.

As per the design philosophy, the McASP driver shall be split into three layers in order to increase the reusability of the driver. The upper layer called the Device Independent layer responsible for buffer management and application synchronization. The middle layer is called the IOM layer and is specific to the McASP and EDMA devices, which exposes standard

interfaces to the device independent layer. The lower level layer is called Lower level controller constitutes a set of well-defined API that abstracts low-level details of the underlying SoC device so that user can configure, control (start/stop etc.) and have read/write access to peripherals without having to worry about register bit field details.

2.1.1.1 LLC Layer

Please refer section 1.2.2 of BIOS_PSP_User_Guide.doc for Hardware layer explanation.

2.1.1.2 Device Driver Core functionality (DDC)

Please refer section 1.2.3 of BIOS_PSP_User_Guide.doc for DDC layer explanation.

2.1.1.3 McASP driver's IO Mini Layer

Please refer section 1.2.4 of BIOS_PSP_User_Guide.doc for IOM Layer explanation.

The functions exported by the McASP IO Mini layer through the IOM_Fxns table "McASP_IOM_FXNS" are listed below.

- **mdBindDev**: Called by the application to bind the device to the IO mini layer.
- **mdCreateChan**: Used to create I/O channels.
- **mdDeleteChan**: Used to delete I/O channels.
- **mdSubmitChan**: Used by the upper layer for submitting I/O packets containing the information needed by the mini driver to program the EDMA channels for data transfer.
- **mdControlChan**:Used to perform device specific control operations.
- **mdUnBindDev**: Used to unbind the IO mini driver from the device.

2.1.2 Data Structures

The mini-driver employs the PortObj and ChannelObj structures to maintain state of the port and channel respectively.

In addition, the driver has two other structures defined – the devParams and chanParams. The devParams structure is used to pass on data to initialize the driver during DSP-BIOS initialization. The chanParams structure is used to specify required characteristics while creating a channel.

The data structures and their usage are explained in more detail in the API reference guide.

2.1.2.1 The McASP Port Object

```
#define PSP_MCASP_NUM_CHANS 2 //maximum 2 channels

#define PSP_MCASP_MAX_SERS  5 // FIVE serializers to the port
```

```
typedef struct _McASPPortObj_t{
        Bool                inUse;
        /**< Marks if port is currently in use                    */
        Uint16              InstNum;
        /**< Preserve  instance number in port                    */
        Bool                isDataBufferPayloadStructure;
        /**< Input request is a flat data buffer or a payload structure */
        McASPObj            XmtObj;
        /**< Holds transmit channel to the McASP.                 */
        McASPObj            RcvObj;
        /**< Holds receive channel to the McASP.                  */
} McASPPortObj , *McASPPortHandle;
```

2.1.2.2 The McASP Object

```
typedef struct  {
   PSP_McaspChannelStatus     status;
    /**< Keeps tab of whether channel is already in use  takes value of
    UNALLOCATED or ALLOCATED */

   Uint16                 mode;
   /**< mode for channel                                  */
   Uint16                 inUse;
   /**< mode for cache                                    */
   PSP_mcaspMode          channelOpMode;
   /**< Mode of operation: Transmit or Receive            */
   struct McASPPortObj_t      *portHandle;
   /**< Pointer to McASP device - Back pointer to device configuration structure   */
   PSP_Handle             ioHandle;
   /**< ddc channel Handle */
```

```
    IOM_Packet          *dataPacket;
    /**<  current active I/O packet        */
    IOM_TiomCallback    cbFxn;
    /**<  Notify client when I/O complete   */
    Ptr                 cbArg;
    /**< Callback Function                  */
} McASPChannelObj, *McASPChannelHandle;
```

2.1.2.3 The devParams structure

```
typedef   PSP_Mcasp_DevParams {
    /* Initial setup for the McASP */
    CSL_McaspHwSetup *mcaspHwSetup;
}  PSP_Mcasp_DevParams;
```

2.1.2.4 The chanParams structure

```
typedef struct PSP_Mcasp_ChanParams {
    Uint16      noOfSerRequested;
    /**< Serializer requested by channel. Channel can ask for both.     */
    Uint32      indexOfSersRequested[PSP_MCASP_MAX_SERS];
    /**< Multi Serializer numbers requested by channel
*/
    PSP_McaspHwSetupData *mcaspSetup;
    /**< Setup information for xmt/rcv sections of the McASP          */
    Bool        isDmaDriven;
    /**< This parameters determines whether channel operates in DMA mode
       All DMA parameters would be read only if this is TRUE         */
    Uint16      channelMode;
    /**< Specifies mode of operation (TDM or DIT) for transmit channel   or else
          just receive mode*/
    PSP_Handle          hEdma;
    /**< Edma handler */
```

```
    appCallback          prdCallback;
    /**< * callback required when the loop job is running -
         * must be callable  directly from the IOM layer      */
    Ptr                  prdBuf;
    /**< Buffer to be transferred when the loop job is running        */
    Uint16               prdCnt;
    /**< Number of frames to be transferred before calling the callback */
} PSP_Mcasp_ChanParams;
```

2.1.2.5 The IO_Packet structure

The IOM_Packet has an address field that is a void* pointer. This will be used to pass on information about both the EDMA start address to be programmed for the packet, as well as the Channel Status RAM and User Data RAM information for this packet. The Channel Status RAM and User Data RAM contain valid data only if the channel is transmitting in DIT mode. In other cases, these fields are not read by the channel.

```
typedef struct _PSP_Mcasp_PktAddrPayload {
    PSP_McaspChStatusRam *chStat;
    /**< Channel Status RAM info                          */
    PSP_McaspUserDataRam *userData;
    /**< User Data RAM info                               */
    Bool writeDitParams;
    /**< Determines whether Channel Status and User Data get
     *  written in case of interrupt mode                 */
    Uint32 *addr;
    /**< Actual address to program DMA with
        Address of data word if transactions are interrupt driven */
} PSP_Mcasp_PktAddrPayload;
```

2.2 Dynamic view of the DSP/BIOS McBSP driver

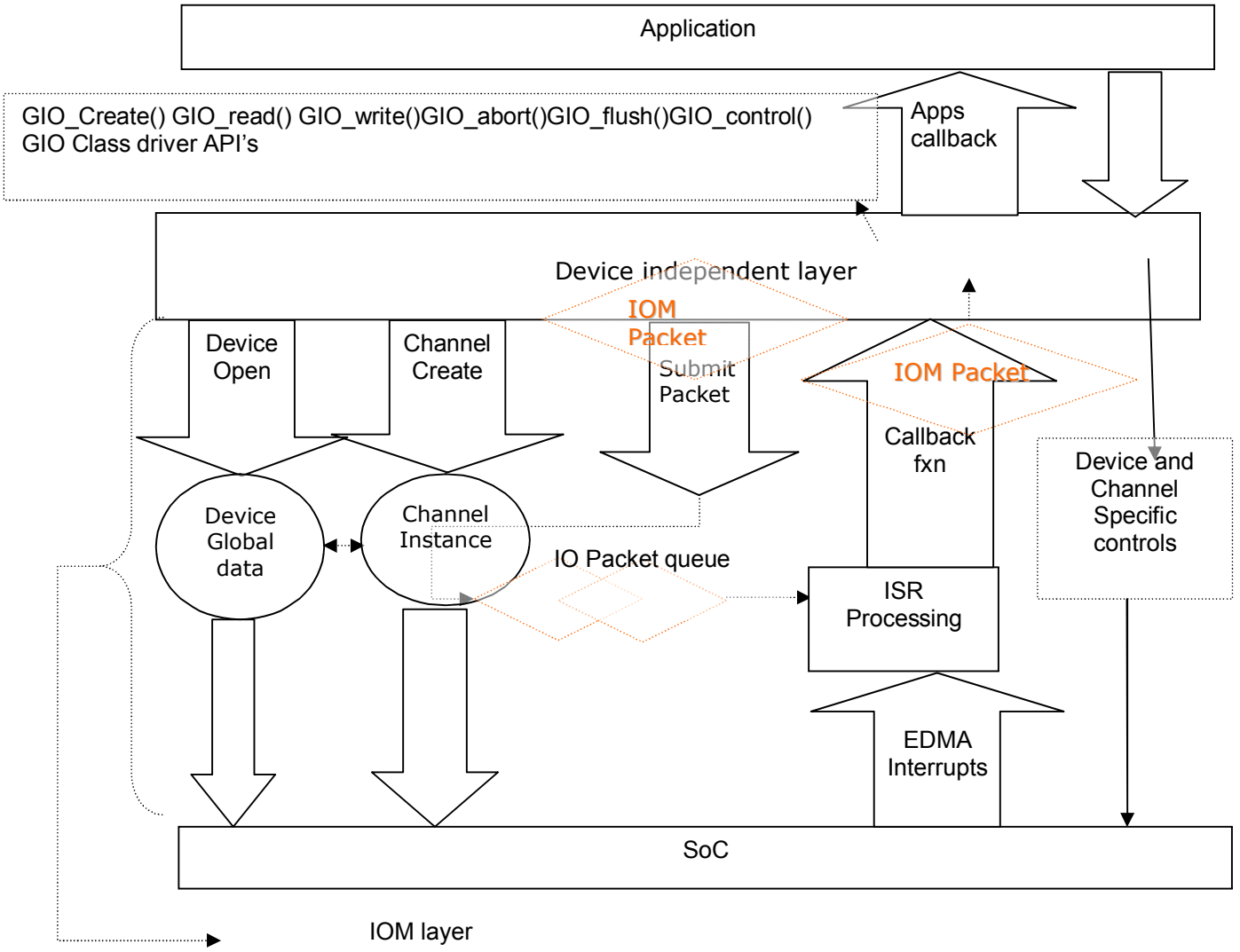

Figure 2: Dynamic view of the McASP driver

When the bios calls mdBindDev() of mini driver of McASP driver, the PSP_mcaspCreate () function of the IOM layer is invoked first and is responsible for creating device instance and initializing the device object and channel object structure of the McASP IOM driver.

Figure 2. shows the flow of data from the application to the driver to the underlying physical device. The IO packet shown in the Figure 2 is standard structure used to submit the I/O requests to the IOM layer of the McASP driver. It contains pointer to the data buffer, size of the buffer and the status of the request. The mode of the IO channel with which the packet

was issued decides whether it is a read or a write command, not the IO packet command field.

Before data communication between an application and a device can begin, a channel instance handle must be returned to the application by a call to GIO_create() API. The channel handle represents a unique communication path between the application and McASP device driver. All subsequent operations that talk to the driver shall use this channel handle. A channel object typically maintains data fields related to a channel's mode, I/O request queues, and possibly driver state information. Application should relinquish channel resources by deleting all channel instances when they are no longer needed through a call to GIO_close ().

Application shall call GIO_submit () API to submit read/write I/O request to driver. The Device Independent layer shall construct an I/O packet and submits the packet to the IOM layer to do the I/O operation. When a mini-driver completes its processing, usually in an ISR context, it calls its associated callback function to pass the IO packet back to the device independent layer of the McASP driver and the device independent layer of the driver in turn calls the application specified callback for that particular I/O request. The submit/callback function pair handles the passing of IO packets between the application and the McASP IOM layer of the driver. Before an IO packet is passed back to the upper layer driver, the mini-driver must set the completion status field and the data size field in the IO Packet. This status value and size are returned to the application call that initially made the I/O request.

2.2.1 Driver Open (Driver initialization and Binding)

The McASP IOM driver initializes the global data used by the McASP driver. The initialization function for the McASP driver is not included in the IOM_Fxns table, which is exported by the McASP driver; instead a separate extern is created for use by the DSP/BIOS. The initialization function is responsible for returning the “IOM” function table structure, which is needed by the device independent layer of the driver.

The initialization function sets the “inUse” field of both the McASP port object instance and the channel object instance to “FALSE” to make sure that the driver is not being used by any applications.

The binding function (mdBindDev) of the McBSP IOM mini-driver is called by application before using the driver. This function shall typically perform the following actions:

1. Set device defaults and perform setup based on the configured device parameters and optional global device data.
2. Acquire driver resources such as McASP.
3. Configure the McASP by default for the following operations.
    - Data output for audio playback
    - Data input for audio recording
    - Configure the McASP in DSP data format mode of the audio codec.
    - Configure the McASP to receive the Frame Sync and bit clock either externally or internally for both receiver and transmitter depending on the device parameter input.

The mdBindDev () of the McASP IOM driver expects device setup parameters in the "PSP_mcaspDevParams" structure defined in the "psp_mcasp.h" header file.

The parameters are explained in the following table

Table 1: Device setup parameters table

Device Parameters	Description
mcaspHwSetup	Initial setup for the McASP.

2.2.2 Channel Creation

The application can create communication channels by calling GIO_create() API which in turn calls McASP IO mini driver's mdCreateChan function. The application shall call mdCreateChan twice with change in mode parameter to create two logical channels one for input (for audio recording) and one for output (for audio playback). The mdCreateChan function should allocate a channel object and set the fields in the channel object to their initial values as needed. For each channel there will be a channel object and the mode field in the channel object specifies whether this is an input or output channel. McASP driver acquires the necessary EDMA channels used to transfer of data. Application has to pass the channel parameters in the "**PSP_mcaspChanParams**" structure exposed by the driver.

Table 2: Channel setup parameters table.

Channel Parameters	Description
"noOfSerRequested"	Serializer requested by channel. Channel can ask for both.
"indexOfSersRequested"	Multi Serializer numbers requested by channel
"mcaspSetup"	Setup information for xmt/rcv sections of the McASP
"isDmaDriven"	This parameters determines whether channel operates in DMA mode. All DMA parameters would be read only if this is TRUE
"channelMode"	Channel mode: TDM or DIT
"wordWidth"	The parameter informs the driver what is the width word (not slot) and this help driver indirectly to decided no. of bytes to be transferred into each serialer for each slot- This is very important parameter - in case of invalid value default value driver will assume is 32
"userLoopJobBuffer"	Buffer to be transferred when the loop job is running it should be

	noted that this buffer size should be n*userLoopjobLength where n is the no of serialisers configured in the direction of the channel we are creating
“userLoopJobLength”	Number of bytes of the userloopjob buffer for each serialiser Please note that this is no. of bytes and this should be pre-calcuated properly for word width of slot - Please refer the wordWidth of this structure
“edmaHandle”	Handle to the EDMA Driver
“gblCbk”	callback required when global error occurs must be callable directly from the ISR context
“noOfChannels”	No of channels of data to be transmitted after the frame sync. This input is valid only for TDM in DSP mode mode of communication E.g.--For Stereo data the value is 2 and for 6 channel dac taking all channel data through one serialiser the value of this member will be 6 Note: But for same 6ch dac taking stereo data though 3 seperate serialiser value should be 2

2.2.3 IO Access

Application invokes GIO_read () and GIO_write () APIs for data transfer using McASP. These APIs in turn creates and submits an IOM packet containing the all the transfer parameters needed by the IOM driver to program the underlying hardware for data transfer. The mdSubmitChan function of the McASP IOM driver must handle command code passed to it as part of the IOM_Packet structure. Depending on the command code, it either handles the code or returns the IOM_ENOTIMPL (not implemented) error code.
The command codes currently supported by the McASP IOM mini-driver are: IOM_READ, IOM_WRITE, IOM_ABORT, and IOM_FLUSH.

- **IOM_READ.** Drivers that support input channels must implement IOM_READ.
- **IOM_WRITE.** Drivers that support output channels must implement IOM_WRITE.
- **IOM_ABORT and IOM_FLUSH.** To abort or flush I/O requests already submitted, all I/O requests pending in the mini-driver must be completed and returned to the device independent layer. The mdSubmitChan function should dequeue each of the I/O requests from the mini driver's channel queue. It should then set the size

and status fields in the IOM_Packet. Finally, it should call the cbFxn for the channel.

- While aborting, all input and output requests are discarded.
- While flushing, all output requests are processed normally and all input requests are discarded. This requires the processing of each IOM_Packet in the original order they were queued up to the channel.

2.2.4 IO Control

McBSP IO Mini driver implements device specific control functionality which may useful for any audio codec driver, which internally uses the McBSP IOM driver. Application may invoke the control functionality through a call to GIO_control (). McBSP IOM driver supports the following control functionality.

- PSP_CHAN_TIMEDOUT: Send channel timeout command.
- PSP_CHAN_RESET: Send channel reset command.
- PSP_MCASP_CNTRL_SET_FORMAT_CHAN: Set channel format.
- PSP_MCASP_CNTRL_GET_FORMAT_CHAN: Get channel format.
- PSP_MCASP_CNTRL_SET_GBL_REGS: Set registers affecting McASP device.
- PSP_MCASP_SET_DLB_MODE: Set digital loopback mode.
- PSP_MCASP_SET_DIT_MODE: Set DIT mode.
- PSP_MCASP_STOP_PORT: Stops the data transfer operation after all the linked requests are complete.
- PSP_MCASP_ START_PORT: Re-starts the data transfer operation from the queued requests if they are queued or whenever a next requests are made from application
- PSP_CTRL_McASP_MODIFY_LOOPJOB: Issues a loopjob operation. When issued with non NULL loopjob buffer driver will enter in to loop job operation and when issued with loopjob buffer equal to NULL driver stops loop job and links loop parameter to zero data buffer.
- PSP_CTRL_McASP_MUTE_ON: Enable Mute command. This will mute the audio data by issuing requests with data buffer filled with zeros.
- PSP_CTRL_McASP_MUTE_OFF: disable Mute command. This will enable the current data to playback by replacing zero data buffer with the request buffer
- PSP_MCASP_DEVICE_RESET: Resets the entire the device and re-configure the device.

- PSP_MCASP_CNTRL_AMUTE: Enable\Disable Amute command.
- PSP_MCASP_GPIO_CONFIG: Enable\Disable GPIO.
- PSP_MCASP_QUERY_AMUTE: Query McASP Amute register.
- PSP_McASP_PAUSE: Stops the data transfer operation after all the linked requests are complete.
- PSP_McASP_RESUME: Re-starts the data transfer operation from the queued requests if they are queued or whenever a next requests are made from application
- PSP_CTRL_RCV_GPIO_INPUT: This IOCTL is specific for EVM. When McASP is in master mode (i.e. AIC33 is in slave mode) we need to configure receive pins as GPIO in input mode. This IOCTL configures AFSR and ACLKR pins in GPIO input mode

2.2.5 Channel Deletion

Application can free the resources held by the channel, if the channel is currently not in use, by calling GIO_delete() API. The corresponding "mdDeleteChan ()" function of the McASP IOM driver shall run from the application context and should de-allocate the specified channel object.

2.2.6 Driver Close

The "mdUnBindDev ()" shall free resources allocated by the "mdBindDev ()" function. If successful, "mdUnBindDev ()" function should return IOM_COMPLETED. If unsuccessful, this function should return a negative error code.

2.2.7 Asynchronous IO Mechanism

The McASP IOM driver supports asynchronous IO mechanism. In Asynchronous IO mechanism multiple IO requests can be submitted in one shot without causing the thread to block while waiting for resources. Application can submit multiple I/O requests using the GIO_read() or GIO_write () APIs and then callback function that was specified during the transfer request submission shall be called as a result of transfer completion by the driver for every transfer. The driver queues the IOM packets submitted internally to support the asynchronous I/O.

2.3 Usage scenario

The McASP mini-driver is so designed that each of the channels can be used independently. The following diagrams depict the flow of control during the course of operation.

In the DSP-BIOS model, the flush and abort are blocking calls, i.e. the GIO layer waits for the flush/abort packet to be returned marked as IOM_COMPLETED, and no new packets can be submitted to the channel till this is done. After a flush/abort is done, the application can submit packets using GIO_submit.

A similar scenario is presented if the application operates the channel in SYNC mode. In this case, the application submits a packet to the mini-driver and then waits for the packet to be returned, either marked as IOM_COMPLETED when the packet has been processed or IOM_ABORTED in case of a time-out condition.